CS 710: Complexity Theory 10/25/2011

Lecture 14: Randomized Classes

Instructor: Dieter van Melkebeek Scribe: David Guild

1 Introduction and Examples

Recall that randomness is a feature we can add to a computational model that provides a sequence of random bits for the algorithm to use. However, randomness can also introduce errors. Since we can trivially get 50% accuracy with a randomized computation, we will only be interested in randomized algorithms that perform somewhat better than a simple coin flip.

1.1 Polynomial Identity Testing

One problem that is hard to solve with previously discussed methods, but that becomes much easier with randomness, is polynomial identity testing. We are given an arithmetic formula over several variables, and wish to know if the formula is identically zero. As far as we know there is no efficient way to solve this problem, but there is an easy and simple randomized algorithm. Namely, we can randomly choose some values and evaluate the formula. If the result is non-zero, then we know for sure that the formula is not identically zero; conversely, if the formula is non-zero, the probability that our randomly chosen point evaluates to zero is bounded. By repeating this process, we can reduce the probability of a false positive to below any desired threshold.

This algorithm only works when the polynomial is expressed as a formula. If we are instead given an arithmetic circuit, the degree of the polynomial might be as large as 2^n. We can still pick a random input to test, which will only require polynomially many bits, but the evaluation of the circuit might involve numbers that are exponentially large.

1.2 Arithmetic Circuit Evaluation

Another example problem is arithmetic circuit evaluation. Here we have a multivariate arithmetic circuit C and an input x, and wish to know if $C(x) = 0$. While this seems easier than the previous example, note that because the expression is given as a circuit, its formula expansion (and thus its evaluation time) might be exponentially large. The key difference is that a circuit can use the output of one node as input to many other nodes; a formula would have to repeat the corresponding expression instead. We will again construct an algorithm with one-sided error.

Note that we can quickly evaluate $C(x) \mod p$ for small values of p. The random algorithm for this problem is to randomly choose $p \in [2, N]$ and then evaluate mod p. If this result is non-zero, then the full evaluation must also be non-zero. We also know that if $C(x) \neq 0$, then $C(x) \bmod p$ will only be zero if $p|C(x)$. In order to finish this algorithm, we need to determine that the probability of error is not too large, and decide a value for N.

Consider the case when p is prime. Clearly there are only a few distinct prime factors of $C(x)$ that are less than N. Specifically,

$$Pr[C(x) = 0 \mod p \mid C(x) \neq 0] = \frac{\#\text{ of prime divisors of } C(x) \leq N}{\#\text{ of primes} \leq N}$$

If $|\langle C, x\rangle| = n$, then $x \leq 2^n$ and the degree of C is $\leq 2^n$. So $C(x) \leq (2^n)^{2^n}$. Further, the number of distinct prime factors of a is less than $\log a$. Recall that the prime number theorem states that $\pi(x) \approx \frac{x}{\ln x}$, where $\pi(x)$ is the number of primes $\leq x$. Then the number of primes $\leq N$ is roughly $N/\ln N$, and

$$Pr[C(x) = 0 \mod p \mid C(x) \neq 0] \leq \frac{n2^n \ln N}{N}$$

If we choose $N = c \cdot n^2 \cdot 2^n$ this can be made $\leq 1/3$. The exact numbers will be explained below.

Note that this only holds if we chose p to be prime. While we could pick a random p and test its primality, there is an easier way. Notice that even for composite numbers, there is no chance of false negatives: if $C(x)$ is zero, then it will also be zero modulo any number, prime or composite. Then we simply need to choose enough random numbers p so that enough of them are prime. Since the probability of a randomly chosen number being prime is roughly $\frac{1}{\log N} \approx O(\frac{1}{n})$, we can pick polynomially many random numbers and end up with a good enough error.

The probability that we will get a false positive is at most the probability that we will pick either a composite number or a prime divisor of $C(x)$. Then the error ϵ_0 of a single guess is

$$\epsilon_0 \leq \frac{(N - N/\ln N) + n \cdot 2^n}{N} = 1 - \frac{1}{\ln N} + \frac{n \cdot 2^n}{N} = 1 - \frac{1}{dn} + \frac{1}{cn} = 1 - \frac{1}{2dn}$$

where d is some constant and $c = 2d$. Then we can repeat this process k times to get an error bounded by $(1 - \frac{1}{2dn})^k \leq e^{-k/2dn}$, which is small for $k > 2dn$.

1.3 Perfect Matching in a Bipartite Graph

The idea here is to reduce this problem to polynomial identity testing, which we know how to solve. Given a graph G with n vertices, create n^2 variables $x_{i,j}$ for each of the potential edges, construct an adjacency matrix M filled the $x_{i,j}$s, and define $P_G(x) = \det(M)$.

Recall that

$$\det(M) = \Sigma_{\sigma \in S_m} (-1)^{\text{sign}(\sigma)} \Pi_{i=1}^{n} M_{i,\sigma(i)},$$

so the determinant is non-zero if and only if there exists a permutation that corresponds to a matching. Then $P_G(x) \not\equiv 0 \iff \exists$ perfect matching for G.

The evaluation of P_G is slightly tricky, but can be expressed as matrix multiplication, which we can parallelize in NC^2. The overall algorithm is then a one-sided randomized NC^2 algorithm.

1.4 ST-connectivity

For a long time, there was no known deterministic log-space algorithm for ST-connectivity in undirected graphs. The standard breadth-first or depth-first algorithms both require linear space. However, there is an easy log-space randomized algorithm: perform a random walk from the start node, and return yes if the end node is reached. This algorithm has one-sided error, since if a random walk arrives at the destination then the two nodes must be connected.

Recently, a deterministic log-space algorithm for this problem was found. However, it is significantly more complicated, so the above randomized method is still used more often.

2 Modeling a Random Computation

Formally, we consider a randomized Turing Machine to have an extra tape containing a sequence of perfectly random bits. This tape is one-way and read-only, so the machine cannot use it for anything other than making random coin flips. The extra random tape does not count towards the machine's space usage, but we otherwise define time and space bounds in the normal way.

The class BPTIME($t(n)$), or Bounded Error Probabilistic Time, is defined as the set of languages that can be decided by a randomized machine in time $t(n)$ with two-sided error at most $1/3$. Note that our choice of $1/3$ is somewhat arbitrary; any language with two-sided error bounded by $1/2 - \epsilon$ is only a constant factor away from this. Nonetheless, we will use $1/3$ as the standard error bound for languages in this class. The class BPP is defined as $\cup_{c>0}$ BPTIME(n^c); in other words, the randomized equivalent of P. The log-space class BPL is defined similarly.

Randomized Time RTIME($t(n)$) is the class of languages with one-sided error at most $1/2$, and Zero Probabilistic Time ZPTIME($t(n)$) is the class of languages with zero-sided error at most $1/2$. Again, the choice of $1/2$ is simply for consistency; any error bounded by $1 - \epsilon$ would suffice. We also define the respective poly-time and log-space classes RP, RL, ZPP, and ZPL.

For space-bounded settings, we additionally require that a randomized Turing Machine *always* halts, regardless of the random bits. (This restriction is unnecessary for time-bounded settings.) Note that this is different from halting with probability 1. Consider a machine that halts when it sees a 0 on the random tape; while the probability of halting is 1, there is a sequence of bits for which the machine never halts, so we cannot say that this machine always halts. Without this restriction, we could use directed ST-connectivity to show that ZPL = NL.

Theorem 1. ZPP = $\{L \mid L$ *can be decided by a RTM that halts with probability 1, is correct with probability 1, and has a polynomial expected running time*$\}$.

Proof. Given a language $L \in$ ZPP, construct a potentially non-halting RTM M that runs the zero-sided error algorithm for L. If it receives a yes or no result, return that answer; otherwise start over. The expected number of times M will have to run the zero-sided algorithm is at most 2, so the expected running time is polynomial. Similarly, this RTM halts and outputs the correct answer with probability 1.

Given an RTM M for language L that halts and is correct with probability 1, and has expected running time n^c, we can construct an always-halting machine with zero-sided error at most $1/2$. Run M with a clock, and halt after $2n^c$ steps. If M returned an answer, then output that answer, otherwise output "unknown". Since the probability of M running for more than $2n^c$ steps is at most $1/2$, the error of this machine is also at most $1/2$, so $L \in$ ZPP. □

3 Relationships between Classes

$$\text{P} \subseteq \text{ZPP} = (\text{RP} \cap \text{coRP}) \subseteq \text{BPP} \subseteq \text{EXP}$$

The inclusions P $\subseteq$ ZPP $\subseteq$ BPP should be obvious. For the equality ZPP = (RP $\cap$ coRP), note that a machine that returns "unknown" can be easily modified to instead return yes or no and thus have one-sided error. In the opposite direction, if there is both an RP and a coRP machine, we can run both of them and thus get zero-sided error. Finally, an algorithm that is allowed exponential time can simulate all possible sequences of random bits, and use the majority answer.

It is conjectured that P = BPP, but we cannot yet prove this. If true, then randomness does not actually increase the power of a computation, although as we have seen it can greatly simplify certain algorithms.

$$\mathrm{L} \subseteq \mathrm{ZPL} = (\mathrm{RL} \cap \mathrm{coRL}) \subseteq \mathrm{BPL} \subseteq \text{uniform } \mathrm{NC}^2 \subseteq \mathrm{DSPACE}((\log n)^2$$

The inclusion BPL $\subseteq$ uniform NC^2 can be shown by modeling the BPL machine as a Markov process. This results in a matrix of transition probabilities $p \in \{0, 1/2, 1\}$, where the $1/2$ represents a transition that depends on a random bit. The simulation is then an instance of iterated matrix multiplication, which we saw last time is in NC^2.

It can be shown that BPL $\subseteq$ DSPACE$((\log n)^{3/2}$. It is conjectured that BPL = L, but once again we do not know how to prove this.

3.1 Related Theorems

The following theorems are known to be true, but their full proofs are left as an exercise for the reader.

Theorem 2. $\mathrm{BPP}^{\mathrm{BPP}} = \mathrm{BPP}$
$\mathrm{NP} \subseteq \mathrm{BPP} \implies \mathrm{PH} \subseteq \mathrm{BPP}$
$\mathrm{NP} \subseteq \mathrm{RP} \implies \mathrm{NP} = \mathrm{RP}$

The first statement is somewhat equivalent to the theorem that $\mathrm{P}^{\mathrm{P}} = \mathrm{P}$. The general observation here is that an oracle for a BPP language can be simulated by a randomized machine without too much additional error. Note that an oracle always returns the correct yes/no answer, with no probability of error; oracles are for languages, not methods of computation.

The last statement is equivalent to RP $\subseteq$ NP. Because the RP class is one-sided, a non-deterministic machine can simply guess a sequence of random bits and not have to worry about false positives.

The middle statement will be a homework exercise.

Theorem 3. $\mathrm{BPP} \subseteq \mathrm{P}/poly$
$\mathrm{BPL} \subseteq \mathrm{L}/poly$

For these theorems, note that repeating an algorithm with two-sided error $\leq 1/3$ roughly n times will reduce the total error to $\epsilon < 1/2^n$. Consider the probability that a randomly chosen bit sequence errs on *any* input of length n. Since the probability of erring on a certain input is at most ϵ, by union bound this probability is at most $2^n\epsilon < 1$. Then there must be at least one bit sequence that correctly decides all input of length n, and this will be our advice.

3.2 Hierarchy for BPTIME

There is no known hierarchy for BPTIME. The general proof for other time hierarchies does not seem to apply here: although we can enumerate random Turing Machines, there is so guarantee that they will obey any bounds on their error. A diagonalizing machine would need to have bounded error as well, and there is no easy way to ensure that.

Next Time

In the next lecture, we will show that BPP $\subseteq \Sigma_2^p \cap \Pi_2^p$. We will then begin discussing expanders.

Acknowledgments

In writing the notes for this lecture, I perused the notes by Brian Rice & Jake Rosin for lecture 11 and the notes by Beth Skubak & Nathan Collins for lecture 12 from the Spring 2010 offering of CS 710.