CS 710: Complexity Theory 2/23/2010

Lecture 11: Randomness

Instructor: Dieter van Melkebeek Scribe: Brian Rice & Jake Rosin

Last time we discussed parallelism as an extension of our existing model of computation. Today we complete that discussion. Then, we will introduce randomness into our computational model. We will show the qualitative usefulness of randomness through efficient probabilistic algorithms for a few difficult problems, and then quantify its power in comparison to existing computational classes.

1 Last Bits of Parallellism

Last time we left off the last step in proving the equivalence of several computational classes.

Proposition 1. *Log-depth formulas can be simulated by bounded–width branching programs of polynomial size.*

Proof. The BP M we construct has the following properties:

- The width of M is 5.
- The label of each node depends only on its layer, that is, M is an *oblivious BP*.
- Between any two levels, all of the 0–branches have distinct end states and all of the 1–branches have distinct end states. Since the width of M is constant, this makes M a *permutation BP*.
- Overall, the effect of M is the identity permutation e if its input is rejected, or a given 5-cycle π if its input is accepted. We say that such a BP is *π–accepting*.

We will construct M recursively. That is, the proof is by induction on formula size. The base case is for individual input variables. This is easy, as we just let the 0–branches from the top layer to the second (bottom) layer form the identity permutation e and the 1–branches form the permuation π.

We will use the following results to recursively construct a polynomial size BP of the required type for a given log-depth formula.

Claim 1. *If there exists a π–accepting BP for the formula ϕ of size s, for* some *5-cycle π, then there exists a σ–accepting BP of size s for ϕ for* any *5-cycle $\sigma \neq e$.*

Proof. In the symmetric group, every 5-cycle σ is conjugate to π; that is, there exists a permutation γ such that $\sigma = \gamma^{-1}\pi\gamma$. So, to construct a σ–acceptor from a given π–acceptor, we need only topermute the machine's top-layer nodes by γ^{-1} and the bottom-layer nodes by γ. This does not change the size of the program. □

Corollary 1. *If we can decide the formula ϕ using a π–acceptor of size s, then we can decide the formula $\neg\phi$ using a π–acceptor of size s.*

Proof. Build a π^{-1}–acceptor that decides ϕ. We can do this by the previous claim. Apply π to the last row to obtain a π–acceptor that decides $\neg\phi$. Again, this does not change the size s. □

Claim 2. *If there exists a π–acceptor of size s_ϕ that decides the formula ϕ and there exists a σ–acceptor of size s_ψ that decides the formula ψ, and $\tau = \pi^{-1}\sigma^{-1}\pi\sigma \neq e$, then there exists a τ–acceptor of size $2(s_A + s_B)$ that decides the formula $\phi \wedge \psi$.*

Proof. Suppose that M_ϕ is the π–acceptor deciding ϕ, and M_B is the σ–acceptor deciding ψ. Let $M_{\phi\wedge\psi}$ be the machine formed by concatenating M_ϕ^{-1}, M_ψ^{-1}, M_ϕ, and M_ψ, in that order, start-to-finish. (M_ϕ^{-1} is a π^{-1}–acceptor deciding ϕ, and M_ψ^{-1} is analogous. The previous claim shows how to build these.)

Consider the fate of input x fed to $M_{\phi\wedge\psi}$. If $\phi(x) \wedge \psi(x)$ is true, then on input x we obtain the permutation π^{-1} from M_ϕ^{-1}, followed by σ^{-1} from M_ψ^{-1}, followed by π from M_ϕ, followed by σ from M_ψ. Because $\tau = \pi^{-1}\sigma^{-1}\pi\sigma$, the net permutation that x undergoes is τ.

If, instead, $\psi(x)$ is false, then M_ψ^{-1} and M_ψ perform the identity permutation on input x. Therefore, the net permutation experienced on input x is $\pi^{-1}\pi = e$ if $\phi(x)$ is true, and just e if not. In any case x is rejected. The case where $\phi(x)$ is false is similar. So, the machine $M_{\phi\wedge\psi}$ accepts precisely those x such that $\phi(x)\wedge\psi(x)$ is true – that is, it decides $\phi\wedge\psi$, and the size of the machine is $2(s_\phi + s_\psi)$. □

There exist 5-cycles π and σ so that $\tau \neq e$, for example, let $\pi = (12345), \sigma = (13542)$, and $\tau = (12534)$. Thus, we can combine branching programs to simulate AND gates; and since we've already seen how to simulate NOT gates, we can simulate OR gates by De Morgan's laws.

So, consider the standard post-order formula traversal that these constructions suggest. At each level, our branching program may increase by no more than a factor of two. If the given formula has depth d, then the branching program we construct has size $O(2^d)$. Since d is $O(\log n)$, the size of the branching program is polynomial in n. □

2 Motivation for Randomness

Randomness appears to have great value as a computational tool. It simplifies problems in a large number of settings. The use of randomness also seems essential to cryptography, where the use of a deterministic method to hide a secret means that it could then be deterministically found by an adversary. Randomness in a cryptographic setting will be covered in upcoming lectures.

Here we deal with randomness in the standard setting: that of realizing a mapping from inputs to outputs; for example, decision problems. Intuitively randomness may not seem useful in such a setting, but there are certain problems which may be solved more efficiently using randomness than by any known deterministic algorithm. However, it is not known whether randomness truly provides additional computing power, as we will see.

3 Concept

To make use of randomness we allow a Turing machine to flip coins and base decisions on the outcome of those flips. The configuration of the machine at a given time therefore becomes a random variable based on coin flips. As the outcome is dictated by the final configuration of the

machine, it too becomes a random variable. In the context of decision problems, this introduces the possibility of an incorrect result.

Our aim is that the probability of an error be made negligible, but useful results can be easily generated by a machine which errs with probability nontrivially less than $\frac{1}{2}$ by running the machine repeatedly and taking the majority vote of its outcomes. Given a machine which produces an erroneous result with probability $\epsilon = \frac{1}{2} - \delta$, we can take the majority vote of the results from a number of independent runs. This allows us to ensure that the error is as small as we wish. For k independent runs, we have

$$\Pr[\text{Majority vote is wrong}] = \sum_{i=\frac{k}{2}}^{k} \binom{k}{i} \epsilon^i (1-\epsilon)^{k-i} \leq 2^k (\epsilon(1-\epsilon))^{\frac{k}{2}} \leq (1-4\delta)^{\frac{k}{2}} \leq e^{-2k\delta^2} \tag{1}$$

This shows it is possible to produce an exponentially small probability of error through majority vote of a polynomial number of runs, provided that each run has probability of error bounded away from $\frac{1}{2}$.

There are different ways to allow error in a randomized computation, and which type of error is allowed may affect the class of languages that can be computed.

- 2-sided error: our algorithm can have both false positives (invalid input accepted) and false negatives (valid input is rejected).
- 1-sided error: error possible on only one side; typically we have false negatives but not false positives. Invalid input is always rejected, while valid input is rejected with small probability.
- 0-sided error: never provides an incorrect answer, but may answer "unknown" with small probability.

The computation above, using majority vote to make the error exponentially small, works for algorithms with 2-sided error.

Randomized quick sort is an example of a 0-sided algorithm. Quick sort is even guaranteed to produce the correct result: it will never answer "unknown." The behavior of quick sort is affected by randomness, however, as the time, space, and other aspects of program behavior become random variables.

4 Randomized Algorithms

The goal when randomness is introduced is to obtain solutions with more efficient algorithms than are known deterministically, with efficiency measured in time or space.

4.1 Sequential Time Efficiency

4.1.1 Polynomial Identity Testing

Polynomial identity testing is the following problem. Given an arithmetic formula φ composed of addition, subtraction, multiplication, brackets and variables, the problem is to determine if $\varphi \equiv 0$. Solving this problem deterministically appears difficult; one method involves expanding all terms

and comparing monomials, but the number of monomials may be exponential in the length of the formula. In fact, all known deterministic algorithms run in exponential time.

A randomized solution is formed by choosing values $\bar{x}$ for all variables at random and evaluating the formula. If the result $\phi(\bar{x}) \neq 0$ the formula is rejected; if $\phi(\bar{x}) = 0$, then we guess $\phi \equiv 0$, but may be in error with some probability. A bound on the probability of an error can be determined as below. Given that values for each variable x_i are chosen uniformly at random from some set I, with d being the degree of φ, we have the conditional probability:

$$\Pr[\varphi(x_1, \ldots x_n) = 0 | \varphi \not\equiv 0] \leq \frac{d}{|I|} \tag{2}$$

Provided that the presentation of our polynomial is given as a formula (rather than as an arithmetic circuit; in that case, see the next example), the degree of a formula is bounded by its length, so $d \leq N$. The interval I should be chosen such that $\frac{d}{|I|}$ is sufficiently small. We choose I so that $|I|$ is some polynomial in N, thus ensuring that $\frac{d}{|I|}$ is small, but that each value in I is still specified with $O(\log N)$ bits. Evaluating the formula at worst raises the variables to the power N, meaning the resulting numbers have $O(N \log N)$ bits. All the arithmetic operations involved may be performed in polynomial time in the bit length of these numbers. This forms a 1-sided randomized algorithm for polynomial identity testing, which errs on nonmembers with small probability (producing false positives).

As we noted, no efficient determininstic algorithm for this problem is known. In fact, a subexponential deterministic algorithm would imply nontrivial lower bounds for circuit complexity.

If our polynomial is presented via an arithmetic circuit, much of the above can be made to work, but it is not easy to determine whether $\phi(\bar{x}) = 0$, since ϕ may have degree exponential in the length of the input. Thus, in order to solve the problem in that case, we need an efficient algorithm for the arithmetic circuit value problem.

4.1.2 Arithmetic Circuit Value

Arithmetic circuit value is the following problem: given an arithmetic circuit and inputs, does it evaluate to zero? Because the degree of the corresponding polynomial can be exponential in the depth of the circuit, exact calculation uses $2^{\text{poly } N}$ bits, which prevents evaluation in polynomial time. The solution is to perform all calculations modulo some random number m of at most poly N bits. This allows evaluation in polynomial time, since we need only work with a number of bits equal to the number of bits in m but introduces another source of error; namely, when our result is zero modulo m, but not in fact equal to zero.

If the value is in fact nonzero, there are polynomimally many prime numbers p which produce a zero incorrectly when the computation is performed modulo p. These are the prime factors of the actual value of the circuit. In order to reduce the probability of an error, we would therefore like to test modulo polynomially many different primes p. This could easily be achieved if we had some way of choosing random primes; however, to do this is not an easy problem. Instead, since working modulo composite numbers still cannot give us false negatives (that is, if the value is zero, it must be zero modulo every m), we simply choose enough random numbers to ensure that, with high probability, sufficiently many of them are primes. We choose numbers with number of bits polynomial in N at random. From the prime number theorem, we have that the prime counting

function $\pi(x) := \{p < x \ : \ p \text{ prime}\}$ satisfies

$$\pi(x) \sim \frac{x}{\ln x}. \tag{3}$$

This tells us that a randomly chosen M-bit number is prime with probability approaching $\frac{1}{M}$. Hence we can choose polynomially many numbers with poly(N) bits at random and, with high probability, have polynomially many of them be prime (for appropriate choices of the polynomials). Thus, if the value of our arithmetic circuit is zero modulo all of our m, then it is equal to zero with high probability.

The result is a 1-sided algorithm with Pr[correct]$= \frac{1}{\text{poly}}$ which errs on nonmembers. Running the algorithm multiple times can increase the confidence in the result. As was the case for polynomial identity testing, the best known deterministic algorithm for arithmetic circuit testing runs in exponential time.

4.2 Parallel Time Efficiency

Some problems have polynomial time determinisitve algorithms, but these algorithms are inherently sequential, and there may be no known efficient parallel algorithms for solving the problems. In these cases, we can sometimes improve the efficiency by finding an efficient parallel algorithm using randomness.

The problem of determining the *existence of a perfect matching in bipartite graphs* is such a problem. An inherently sequential polynomial time algorithm exists, but there is no known efficient deterministic parallel algorithm. However, we can find one using randomness.

A bipartite graph G can be represented as an $N \times N$ adjacency matrix. We then replace every '1' in the adjacency matrix with a random variable unique to its location. For example:

$$M = \begin{bmatrix} x_{11} & 0 & 0 & x_{14} & \cdots \\ 0 & x_{22} & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{4}$$

Claim 3. *G has a perfect matching iff Det(M)$\not\equiv 0$.*

Proof. One term in the determinant exists for every possible permutation, and permutations are in a 1-to-1 correspondence with potential perfect matchings. Thus the determinant contains exactly one term for every potential perfect matching; the coefficient of that term is zero if that matching is not realizable in the graph, and 1 if it is. Different perfect matchings lead to different monomials, which if they have a non-zero coefficient are realizable for the graph. Since all the monomials are distinct (and so no cancellation can occur, the polynomial is not identically zero if and only if G has a perfect matching. □

The determinant is a multi-variate polynomial of degree at most N. Checking for a perfect matching can be performed by checking if that polynomial is $\equiv 0$, which can be accomplished using the randomized algorithm above. Unfortunately, this would not be a parallel algorithm. However, this is not just any polynomial; its values are the values of the determinants of the matrices obtained by substituting in numbers for the variables in the matrix. Computing the determinant is a linear algebra problem, which, as we noted in the last lecture, is in NC^2. We use this fact to devise an NC^2 algorithm:

- Replace all '1's in the adjacency matrix with a random value in a suitable interval $I = [0, \text{poly}(N)]$.
- Compute the determinant in parallel.

If the result is non-zero, then a perfect matching exists. If it is zero, the algorithm can be repeated until we become sufficiently confident in the result.

4.3 Space Efficiency

The *undirected path problem* was until recently most efficiently solved by a randomized algorithm. We now have an efficient deterministic Ł solution, but the problem is included as a well-known example. Note that the related directed path problem is NL-complete.

Given an undirected graph G and two vertices s and t, the problem is to determine if a path exists from s to t. The randomized algorithm is a random walk beginning from s. This walk ends after a certain number of steps, or immediately if t is reached. Performing a polynomial number of steps reduces to a small probability the chance of not reaching t if s and t are connected.

This algorithm can be performed in logarithmic space, storing the following:

- Current location
- Destination t
- The number of steps taken - at most polynomially large, and so representable in logarithmic bits.

This gives us a logarithmic space randomized algorithm with 1-sided error.

The efficient deterministic solution to this problem uses expanders, which will be discussed in a future lecture.

5 Next Lecture

In the next lecture we will continue our examination of randomness by formally expanding our Turing machine model, defining randomized complexity classes, and examining their relation to deterministic and nondeterministic complexity classes.