CS 810: Complexity Theory 2/14/2007

Lecture 11: Randomness

Instructor: Dieter van Melkebeek Scribe: Jake Rosin

Last time we discussed parallelism as an extension of our existing model of computation. Today we introduce randomness and extend the Turing machine model to make use of it. We 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 Motivation

Randomness appears to have great value as a computational tool. It simplifies problems in a large number of settings, and is essential in others; for example the Dining Philosophers Problem, which cannot be solved by any deterministic solution. 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 seems like a hindrance in such a setting, but there are certain problems which may be solved more efficiently using randomness than by any known deterministic algorithm. Unfortunately it is not known whether randomness truly provides additional computing power, as we will see.

2 Concept

To make use of randomness we allow a TM 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 also becomes a random variable. In the context of decision problems this introduces the possibility of an incorrect result.

Obviously the probability of an error should 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 $\epsilon = \frac{1}{2} - \delta$, and the results from k independent runs:

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

This shows it is possible to produce an exponentially small probability of error through majority vote of a polynomial number of runs if the original probability of error is not too close to 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: error possible on both the membership and non-membership sides.
- 1-sided error: error possible on only one side; typically the membership side. 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.

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

3 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.

3.1 Sequential Time Efficiency

Polynomial identity testing is a problem where we know of an efficient randomized algorithm but do not know of an efficient deterministic algorithm. 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 for all variables at random and evaluating the formula. If the result $\neq 0$ the formula is rejected; if it $= 0$ then the formula may $\equiv 0$ 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:

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

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 use $|I|$ some polynomial in N thus ensuring $\frac{d}{|I|}$ is small and still each value is 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).

Arithmetic circuit testing can also benefit from this approach. The relevant decision problem is the following: 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 $\text{poly}\, N$ bits. This allows evaluation in polynomial time but introduces another source of error to be dealt with.

Polynomially many prime numbers produce a zero incorrectly (these numbers are the prime factors of the true result). To reduce the probability of an error we must pick our prime numbers from a set so that picking such a prime number from the set happens with small probability. To achieve this, we pick a prime number at random from $[0, \mathrm{poly}(N)]$ for a large enough polynomial. For this to work, we must have some guarantee that for a randomly chosen poly(N) there are a nontrivial number of primes in the interval: the prime number theorem provides this assurance. If $\pi(x)$ is the prime counting function that gives the number of primes less than or equal to x, then:

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

The result is a 1-sided algorithm with Pr[correct]= $\frac{1}{\mathrm{poly}}$ 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.

3.2 Parallel Time Efficiency

The *existence of a perfect matching in bipartite graphs* problem has no known deterministic parallel algorithm. An inherently sequential polynomial time algorithm exists, but finding a parallel algorithm would greatly improve the efficiency with which this problem can be solved. Such an algorithm can be generated by using randomness.

The graph G can be represented as an $N \times N$ adjacency matrix. We replace every '1' in the graph with a random variable unique to that location:

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

Claim 1. *G has a perfect matching iff Det(M) is not $\equiv 0$*

Proof. One term in the determinant exists for every possible permutation; permutations are in a 1-to-1 correspondence with perfect matchings. The determinant contains exactly one term for every possible perfect matching; the coefficient of that term is zero if that matching is not possible in the graph. Different perfect matchings lead to different monomials, which if they have a non-zero coefficient are valid for the graph. □

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. However, this would not be a parallel algorithm. Computing the determinant is a linear algebra problem, which as we saw last lecture is in NC^2. We use this fact to device an NC^2 algorithm:

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

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

3.3 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 will be demonstrated in an upcoming lecture.

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

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

4 Model

We model randomness by equipping a standard Turing machine with a random bit tape, to which the finite control has one-way read access. We assume this tape is filled with bits from a perfectly uniform distribution before computation begins.

The machine is given one-way read-only access because the random bit tape is not counted as used space. To reuse a random value it must be stored on some work tape, while using a random value takes a single computational step to read an entry from the random bit tape.

4.1 Time and Space

The complexity classes BPTIME(t) and BPSPACE(s) are analogous to their deterministic counterparts. They represent the classes of problems solvable by a randomized machine under some time or space bound, with the requirement that:

$$\Pr[\text{error}] \leq \frac{1}{3}$$

Specifically we refer to the classes BPP (bounded-error probabilistic polynomial time) and BPL (log space).

Complexity classes also exist for 1-sided and 0-sided machines. RTIME(t) and RSPACE(s), and the specific classes R (or RP) and RL, refer to problems solvable by randomized machines which meet the following criteria:

$$\Pr[\text{error}|x \in L] \leq \frac{1}{2}$$
$$\Pr[\text{error}|x \in \bar{L}] = 0$$

ZPTIME(t) and ZPSPACE(s), and specifically the classes ZPP and ZPL, refer to problems solvable by 0-sided randomized machines. These machines must meet these criteria:

$$\Pr[\text{error}] = 0$$
$$\Pr[\text{“Unknown”}] \leq \frac{1}{2}$$

Claim 2. *The class* ZPP *is equivalent to the class of problems solvable by randomized machines with no chance of error that run in expected polynomial time (call this class* ZPP2*)*

Proof. A ZPP machine can be run repeatedly until it outputs a definite answer. By the definition of ZPP each run halts in polynomial time, and outputs an answer with probability $\geq \frac{1}{2}$. If N^c is the running time of our ZPP algorithm, then the expected running time of our modified algorithm is:

$$\text{E(run time)} \leq \sum_{i=1}^{\infty} (N^c \cdot i) \cdot (\frac{1}{2})^i \tag{5}$$
$$= N^c \sum_{i=1}^{\infty} i \cdot (\frac{1}{2})^i = 2N^c \tag{6}$$

This new machine has expected running time polynomial and thus satisfies the definition of ZPP2.

Similarly a ZPP2 machine can be run on a clock for $t\cdot$ E(run time), then terminated with output “Unknown”. This modified algorithm outputs “Unknown” with probability at most:

$$\Pr[x \geq t \cdot E(x)] \leq \frac{1}{t} \text{ (Markov's inequality).} \tag{7}$$

To ensure the probability of outputting “Unknown” is at most 1/2, we set $t = 2$. As this new algorithm satisfies the error criterion of ZPP and runs in polynomial time, it is a ZPP algorithm for the language. □

Claim 3. ZPP = RP ∩ coRP

The proof of this claim is left as an exercise.

4.2 Space Complications

In a deterministic setting any log-space algorithm terminates in polynomial time (if it terminates at all), due to the polynomial bound on the number of possible machine configurations. Obviously a repeated configuration signifies a loop, which occurs iff the machine does not halt.

Things become more complicated when randomness is involved. *When discussing space bounds in this setting we assume machines that always halt.* As in the deterministic setting this requires that the machine never repeats a configuration, because doing so would introduce a non-halting loop. This restriction gives us the nice property that a randomized log-space machine runs in polynomial time (and in general, a space s machine runs in $2^{O(s)}$ time).

Note that machines of this type are distinct from those which halt with probability 1. We define a separate class, $\lfloor$ZPL$\rceil$, for that type of machine. An example of the addition power this model has over the one stated above is given by the following claim, which we leave as an exercise.

Claim 4. $\lfloor$ZPL$\rceil$ = NL.

For more on how these definitional issues affect the power of randomness in the log-space setting, see the survey [1].

5 Relation to Deterministic Classes

Relating these randomized classes to the known deterministic classes provides a measure of their computational power.

$$\begin{aligned} &\mathrm{RP} \subseteq \mathrm{NP} \\ &\mathrm{P} \subseteq \mathrm{ZPP} \subseteq \mathrm{RP} \subseteq \mathrm{BPP} \subseteq \mathrm{EXP} \end{aligned}$$

All but the last inclusion are by definition. BPP $\subseteq$ EXP follows from the fact that exponential time is sufficient to exactly compute the probability of acceptance by a randomized machine, by exhaustively generating all possible coin flip results.

$$\text{Ł} \subseteq \mathrm{ZPL} \subseteq \mathrm{RL} \subseteq \mathrm{BPL} \subseteq \mathrm{UniformNC}^2 \subseteq \mathrm{DSPACE}(\log^2 N) \tag{8}$$

Uniform $\mathrm{NC}^2 \subseteq \mathrm{DSPACE}(\log^2 N)$ follows by a proof similar to that of $\mathrm{NC}^1 \subseteq$ Ł given in the last lecture. BPL $\subseteq$ Uniform NC^2 takes a bit more work.

Claim 5. *BPL* $\subseteq$ *Uniform* NC^2

Proof. A BPL computation can be viewed as a Markov chain of machine configurations. For a log-space machine the size of the set of configurations is polynomial. The Markov chain can be represented as a matrix, with each entry M_{ij} representing the probability of transitioning from state j to state i. Multiplying this matrix against itself N times gives the probability matrix for state transitions taking exactly N steps. This can be used to determine the probability of reaching an accepting configuration:

$$\begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1m} \\ p_{21} & p_{22} & \cdots & p_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ p_{m1} & p_{m2} & \cdots & p_{mm} \end{bmatrix}^N \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{9}$$

This is iterated matrix multiplication, which as discussed last time is in Uniform NC^2. □

In fact there is a tighter bound known for the class BPL:

Claim 6. $\mathrm{BPL} \subseteq \mathrm{DSPACE}\left(\log^{3/2} N\right)$

The proof for this claim will be presented in a future lecture, along with evidence for the conjectures $BPP = P$ and $BPL = L$. As with all complexity classes we ignore constant-factor increases.

The following facts are also known (proving the second is the final problem on Homework 1).

Proposition 1.

$$\begin{aligned} &\mathrm{BPP}^{\mathrm{BPP}} = \mathrm{BPP} \\ &\mathrm{NP} \subseteq \mathrm{BPP} \implies \mathrm{PH} \subseteq \mathrm{BPP} \\ &\mathrm{NP} \subseteq \mathrm{BPP} \implies \mathrm{NP} = \mathrm{RP} \end{aligned}$$

The results given above relate randomized complexity classes among each other and with deterministic and nondeterministic complexity classes. The following result relates the randomized class BPP to the non-uniform class $P/poly$, showing that in general randomness can be replaced by non-uniformity.

Theorem 1. BPP $\subseteq$ *$P/poly$ and* BPL $\subseteq$ *$L/poly$*

Proof. Consider a BPP algorithm. We first make the probability of error smaller than the number of inputs of a given length N: Pr[error] $< \frac{1}{2^N}$. Recall from section 2 that running a BPP algorithm with error $1/2 - \delta$ for k times and taking the majority vote results in a BPP algorithm computing the same language but now with error at most $e^{-2k\delta^2}$. We assume original error at most $1/3$, so $\delta \geq 1/6$. We need to pick k large enough so that $e^{-2k(1/6)^2} < 1/2^N$. Then a large enough $k = \Theta(N)$ suffices, meaning the resulting BPP algorithm still runs in polynomial time.

Because we have reduced the error to less than $1/2^N$, given any distinct setting for the random bit tape, the probability that there exists some input x of length N on which the machine makes an error is less than one. Therefore there exists at least one coin flip sequence for which the machine gives the correct result on all inputs of length N. This sequence of coin flips is the advice given. As the amount of randomness used is some polynomial, this is a polynomial amount of advice. □

We stated above that it is conjectured that BPP=P, but we have been unable to prove this. In fact, we cannot yet even prove BPP $\subseteq$ NP, but it has been shown that BPP lies within the polynomial hierarchy.

Theorem 2. BPP $\subseteq \Sigma_2^P \cap \Pi_2^P$

The proof for this theorem will be presented in the next lecture.

6 Next Lecture

In the next lecture we will look at *expanders*, a useful tool which allows error reduction using fewer random bits. Reducing the number of coin flips used will improve efficiency of the trivial derandomization of evaluating every possible series of flips. Expanders are analogous to pseudorandom number generators, in that from a short random input they deterministically produce a series of pseudorandom bits which can be used for many randomized functions as if they were truly random.

We will also address the issue that while our model relies on a series of bits whose randomness is unbiased and uncorrelated, this is functionally impossible to achieve in the real world. Expanders also allow transformation from a series of bits with some randomness to another with essentially perfect randomness.

References

[1] Michael Saks. Randomization and Derandomization in Space-Bounded Computation. *Annual Conference on Structure in Complexity Theory*, 1996.