CS 6810: Theory of Computing **Fall 2021**

Lecture 12: Oct 5, 2021

Lecturer: Eshan Chattopadhyay *Scribe: Mark Schachner*

1 Randomness in Computation

So far, we've seen multiple extensions of the Turing Machine construction, such as allowing machines multiple transition functions, specially labelled states, and access to oracles or advice strings. Today, we'll look at how the recognizable complexity classes change when we allow machines access to randomness. We will refer to machines with this access as *Probabilistic Turing Machines* (PTMs).

Before we give the precise definition of a PTM, it is worth philosophizing for a moment about what in our treatment should constitute randomness. Many concrete processes that are commonly considered random are in fact very possible to predict accurately with enough information; indeed, if you understand well the mechanics of how a die falls, rolls, and bounces, you can deduce which side lands face-up from a given set of initial conditions. Some processes, such as those in chaotic systems or the probablistic world of quantum mechanics, can more plausibly be considered random. For our purposes, it will suffice to black-box a "source of randomness" as a sample from a uniform distribution of binary strings.

On to the definition:

Definition 1.1. [Probabilistic Turing machines] A *probabilistic Turing machine*, abbreviated *PTM*, is a standard multitape Turing machine M with the following augmentation: M takes as input a *random string* $r \in \{0,1\}^*$ as well as the usual input string $x \in \{0,1\}^*$. Additionally, the machine comes equipped with a function $R : \mathbb{N} \to \mathbb{N}$ such that for each input (x, r) we have $|r| = R(|x|)$.

Observe that this definition bears some similarity to the definition of a non-deterministic Turing machine, in that it opens multiple possible paths of computation for the machine. We will examine this connection in more detail later. For now, recall that we defined the recognition of a language L by an NDTM M as the existence of at least one computational path which correctly decides L. In the probabilistic case, we require something stronger, a lower bound on the "success rate" of the machine. There are multiple possible notions of the computation of a language by a PTM, depending on the types of error we allow the machine to make:

Definition 1.2. [PTM Computation] Let M be a PTM, $L \subseteq \{0,1\}^*$, and suppose that for any input (x, r), M halts in time $T(|x|)$ regardless of r.

(a) We say M *computes* L *in bounded-error probabilistic time* $T(n)$ if for all $x \in \{0,1\}^*$ we have

$$\Pr[M(x, r) = L(x)] \geq 2/3.$$

We denote the class of such languages by $BPTIME(T(n))$.

(b) We say M *computes* L *in randomized time* $T(n)$ if for all $x \in \{0,1\}^*$ we have

$$x \in L \Rightarrow \Pr[M(x, r) = 1] \geq 2/3,$$
$$x \notin L \Rightarrow \Pr[M(x, r) = 1] = 1.$$

We denote the class of such languages by $RTIME(T(n))$.

A few remarks about these definitions:

- The constant 2/3 in both definitions is nearly arbitrary; what matters is that the success rate of the machine is bounded away from 1/2. Note that we can always achieve a success rate of 1/2 by simply guessing randomly.
- The key difference between the classes $BPTIME$ and $RTIME$ is their treatment of "false positive" results. For a PTM M to compute L in randomized time, it must always correctly identify strings which are not in L. However, bounded-error probabilistic time permits what is called *two-sided error*, i.e., false positives and false negatives– so long as these are not too common.
- We can form the class $coRTIME(T(n))$; it is not too hard to see that this corresponds to languages which are computable by PTMs as defined above, with the exception that we allow false positives and disallow false negatives. Under the definition below, this will lead to the class $coRP$ which corresponds to languages recognizatble in randomized polynomial time with no false negatives.

With these notions made precise, we can define complexity classes analogous to those we have seen in other cases:

Definition 1.3.

$$BPP := \bigcup_{c \geq 1} BPTIME(n^c), \qquad RP := \bigcup_{c \geq 1} RTIME(n^c).$$

2 Randomness vs. Nondeterminism

As we hinted previously, there is a connection between probabilistic computation and the more familiar nondeterministic computations we have already seen. Intuitively, we can impart probabilistic behavior to an NDTM by allowing it to choose randomly which of its two transition functions to employ at each step of the computation, and requiring some lower bound on the success rate of the resulting randomized process. This is summed up in the following proposition:

Proposition 2.1. *Let M be an NDTM with transition functions δ_0, δ_1. Define a PTM M' such that, on input (x, r), M' computes M on x, where on the n-th step of computation M transitions via δ_0 if the n-th bit of r is 0, and via δ_1 if the n-th bit of r is 1. Then:*

- *M' computes a language L in bounded-error probabilistic time if and only if the proportion of computation paths of M which successfully decide each $x \in \{0,1\}^*$ is at least 2/3, and*
- *M' computes a language L in randomized time if and only if the proportion of computation paths of M which successfully decide each $x \in L$ is at least 2/3, and all computation paths successfully decide each $x \notin L$.*

Thus we have made explicit the equivalence between the nondeterminism afforded by the two transition functions δ_0, δ_1, and the randomness provided by r.

3 Examples of Random Algorithms

The discussion in Section 2 shows that the definitions of randomized computation we have seen are in fact simply a different perspective on nondeterministic computation. So what does this shift in perspective afford us? It turns out that randomized computation is incredibly useful for designing simple algorithms that efficiently solve difficult problems with high success rate. The purpose of this section is to showcase a few examples, to demonstrate the power of randomness in computation.

Example 3.1. [Perfect matchings in bipartite graphs]
Let G be a graph; we identify the edges of G with the nonzero entries of its incidence matrix M. Under this identification, recall G is *bipartite* if there exists a partition $\{V_1, V_2\}$ of its vertex set such that all nonzero entries of M lie in the minor M_{V_1,V_2}; a *perfect matching* on G is a subset of the nonzero entries of M which intersects each row (and thus column) of M exactly once. Therefore a perfect matching on a bipartite graph is a matrix M' such that $M'_{ij} \leq M_{ij}$ for all i, j and M_{V_1,V_2} is a permutation matrix. This reduces the search problem for perfect matchings in bipartite graphs to the problem of whether a given irreflexive symmetric $n \times n$ matrix M "contains" a permutation matrix, in the sense that any nonzero entry of the permutation matrix is nonzero in M as well.

Now, recall the "permutation definition" of the determinant: for a matrix M, $\det(M)$ is the sum over all permutations σ of the product of $M_{i,\sigma(i)}$ over all i. With this definition in mind, we define a matrix X with a random variable $x_{ij} \in \{0, 1\}$ in each position where the incidence matrix for G has a 1. Then we have $\det(X) \neq 0$ if and only if G has a perfect matching; simply choose the x_{ij} to be 1 at each entry of the permutation matrix. Since there exist efficient algorithms to compute the determinant, this gives a simple random algorithm for determining if G has a perfect matching: choose values for the x_{ij} at random, and compute $\det(X)$. If the determinant is nonzero, we halt and accept, and otherwise we repeat until sufficiently confident.

Example 3.2. [Primality testing with the Miller-Rabin test]
Any odd integer greater than 2 is of the form $2^s \cdot d + 1$ for odd d. We say such an integer n is a *probable prime to base* a, where $a < n$ is positive if either $a^d \equiv 1 \bmod n$ or $a^{2^r d} \equiv -1 \bmod n$, for some nonnegative $r < s$. As the name suggests, this is an accurate but imperfect characterization of prime numbers. The *Miller-Rabin primality test* exploits this by testing n with multiple bases. Specifically, the algorithm proceeds as follows. Choose some parameter k which will determine how accurate the test is. Then run the following four steps k times:

(i) Choose a random integer a in $[2, n-2]$, and let $x = a^d \bmod n$.

(ii) If $x = \pm 1 \bmod n$ then go to step (i).

(iii) Otherwise, repeat the following $r - 1$ times:

- Let $x = x^2 \bmod n$.
- If $x = -1$ then return to step (i).

(iv) If none of the $r - 1$ repetitions results in a loop, then halt and output "composite".

If the algorithm runs k times without halting, then the test concludes n is probably prime. The algorithm runs in polynomial time for any fixed k, and the success rate is exponential in k; therefore the existence of this algorithmm places primality testing in BPP.

Example 3.3. [Polynomial Identity Testing]
Let $\mathbb{F}$ be a finite field, let $n \in \mathbb{N}$, and suppose $p, q \in \mathbb{F}[x_1, \ldots, x_n]$ are polynomials of degree $\leq d$. For this example, we are concerned with determining whether $p = q$. We note two facts:

- Since $p = q$ if and only if $p - q = 0$, it suffices to consider the decision problem of whether a single polynomial is equal to zero. For this reason, we refer to this problem as *Polynomial Identity Testing* (PIT).
- Representation matters greatly here. If a polynomial p is given by its coefficients, PIT is trivial since $p = 0$ if and only if each coefficient is zero. In complexity theory, polynomials are often represented by *algebraic circuits*, which are similar to Boolean circuits except with $\mathbb{F}$-valued inputs and gates replaced by the operations $\cdot, \times$. We will abstract this away by instead allowing "black-box access" to p; that is, we will only assume that we can efficiently compute p on any input.

The question of resolving PIT is central to algebraic complexity theory. Many problems, including finding perfect matchings in bipartite graphs, are reducible to PIT. Moreover, it is conjectured, but not known, that there exists a deterministic polytime algorithm to decide PIT. However, we have an easy random algorithm which decides PIT: choose $x_1, \ldots, x_n$ at random and compute $p(x_1, \ldots, x_n)$. If the result is zero, output that $p = 0$, and otherwise output $p \neq 0$. This algorithm works well because of the following result, named for Jack Schwartz and Richard Zippel but also commonly attributed to Richard DeMillo and Richard Lipton:

Lemma 3.4 (Schwartz-Zippel)**.** *Let $p \in \mathbb{F}[x_1, \ldots, x_n]$ be of degree d, and let $r_1, \ldots, r_n$ be randomly and independently chosen from the uniform distribution on $\mathbb{F}$ Then*

$$Pr[p(x_1, \ldots, x_n) = 0] \leq \frac{d}{|\mathbb{F}|}.$$

Other than these three examples, many combinatorial problems and graph-theoretic problems such as the max-cut problem can be efficiently solved using access to randomness; furthermore, these algorithms often use randomness in very simple ways (e.g., choose a random partition of a graph and test if it determines a maximal cut). We will conclude our examples of randomness in computation with a closer look at an algorithm which uses randomness in a more nontrivial way: a randomized algorithm which decides 3SAT.

4 A closer look: Schönig's Algorithm

To conclude, we present a randomized algorithm which decides 3SAT. The algorithm is due to Uwe Schöning, and is based on the concept of *local search:* that is, given a nonsolution of a formula, we look to formulas very close to the original guess until we are successful. The algorithm runs as follows. We are given a CNF φ in n variables.

- Choose an assignment at random.
- Repeat the following n times:
  - Test φ on the assignment. If it is satisfied, stop and accept. Otherwise,
  - Choose an unsatisfied clause in φ at random, and
  - Flip one of the literals in the unsatisfied clause.
- Repeat the above two steps $1/p$ times, where p is the probability that the algorithm stops and accepts on a single repetition.

The third step means that the complexity of the algorithm is polynomial in $1/p$. We will show that $p \geq (2/3)^n$; this can in fact be improved to $(3/4)^n$. This bounds the runtime of the algorithm below by $(4/3)^n$. In order to show our lower bound on p, suppose $x^* \in \{0,1\}^n$ is a satisfying assignment for φ. Denote by $\Delta(x,y)$ the *Hamming distance* between x and y, i.e., the number of positions at which x, y differ. We have

$$Pr[\Delta(x,x^*) = i] = \frac{\binom{n}{i}}{2^n}.$$

Now, say a step in the algorithm is "good" if it decreases the Hamming distance from x to x^*. The probability of a good step is at least $1/3$, so the probability of i consecutive good steps is at least $(1/3)^i$. Thus

$$Pr[\Delta(x,x^*) = i, i \text{ good steps}] = \frac{\binom{n}{i}}{2^n} \cdot (1/3)^i.$$

Summing over all Hamming distances, we obtain

$$p \geq \sum_{i=0}^{n} \frac{\binom{n}{i}}{2^n} \cdot (1/3)^i = \frac{1}{2^n}(1+1/3)^n = (2/3)^n.$$

Therefore, the running time of the algorithm is $(3/2)^n$.