IITM-CS6840: Advanced Complexity Theory 23 Jan, 2012

Lecture 09 : Probabilistic TMs and Randomized Algorithms

Lecturer: Jayalal Sarma M N *Scribe: Balagopal*

1 Review of Branching machines

A branching machine is a machine that is allowed to make non-deterministic guesses while computation. This is a generalization of NTMs where the branching machine accepts iff at least one path accepts. Similarly we can think of various definitions of acceptance each of them leading to a (possibly) new complexity class. For example, PP is defined as the class which has a branching machine where more than half of the paths accept. $\oplus\mathsf{P}$ is defined as the class where each language has a branching machine in which an odd number of paths accept if $x \in L$. Similarly we can think of every branching machine as computing a function $f(x)$ where $f(x)$ is defined as the number of accepting paths of M on x.

Question. How is $\oplus\mathsf{P}$ related to P , NP , PSPACE , and PP ?

We will come back to answer this question, and show that NP can be *almost* solved in P if $\oplus\mathsf{P}$ can be solved in P. The *almost* here will refer to possibility of error by the algorithm in the decision. We will now do a systematic formal study of randomized algorithms (algorithms which may make an error but with low probability). We introduce these from the branching machine perspective that we have seen so far.

2 Characterizing Randomized Algorithms

We are going to characterize randomized algorithms using branching machines that we defined previously.

Consider a branching machine M for the language L. If $x \in L$ call all paths of M that reject as erroneous. If $x \notin L$ call all paths of M that accept as erroneous. Intuitively, we want the erroneous paths to be as small as possible. We use the following notations for counting paths with specific properties for a branching machine M.

$\#paths_M(x)$	Total number of paths of M on x
$\#acc_M(x)$	Total number of accepting paths of M on x
$\#rej_M(x)$	Total number of rejecting paths of M on x

$\#err_M(x)$ Total number of erroneous paths of M on x

We defined class PP as the set of languages L with branching machines satisfying the following property.

$$\begin{aligned} x \in L &\Rightarrow P(A \text{ accepts } x) > 1/2 \\ x \notin L &\Rightarrow P(A \text{ rejects } x) \geq 1/2 \end{aligned}$$

To connect the notion of a branching program and a randomized algorithm, we have to make sure that all paths of the branching machine are of the same length (i.e., the computation tree is a full binary tree). This can be done by a construction similar to the one used to do this while defining #P. However, when a path is extended by branching it yields multiple paths. We must ensure that the error probabilities are not increased during this process. Note that this is different from what we did while defining #P. The goal there was to preserve the count (Number of accepting paths). The goal here is to keep the error probability (which depends on the relative number of accepting and rejecting paths) the same.

Now consider a randomized algorithm A that chooses one path of M uniformly at random and executes it (This shows that there is a randomized algorithm corresponding to every branching machine). Clearly

$$x \in L \iff \#err_M(x) \leq \frac{1}{2}\#paths_M(x) \tag{1}$$

We see that the probability of A making an error is at most 1/2. But this can be achieved by a trivial randomized algorithm that flips a coin and determines the result according to the outcome of the coin flip. So the class PP is not a good candidate for formally capturing "good" randomized algorithms.

What we want is the error probability to be bounded away from 1/2. If $\#err_M(x) < \frac{1}{4}\#paths_M(x)$, then certainly the corresponding randomized algorithm is better than a trivial one.

We will now work towards defining a problem for which there is an efficient (poly time) randomized algorithm but for which no poly time deterministic algorithm is known. This gives reason to study randomized algorithms formally.

3 Polynomial Identity Testing

This problem has its roots in the simple high school arithmetic. Suppose we are given a polynomial in a complicated form where the monomials may repeat with arbitrary coefficients etc. We want to find out if the coefficient of the monomials cancel out to zero. This in effect is testing whether the polynomial is the zero polynomial, and equivalenty it is testing if the polynomials evaluates to zero on all substitutions of the variable from the underlying field $\mathbb{F}$.

How are we given the polynomial? This indeed is going to have effect on the complexity of the problem. Let us start with the high school arithmetic again. Suppose we are given it in the monomial form (though some monomials may repeat) along with their coefficients. To solve the problem, it suffices to check, for each monomial whether the coefficient in its various appearences is adding up to zero. Given the explicit representation at the input, this is very easy to do by simply going over the input for each monomial. Hence this can be done in time polynomial in the input.

What if the polynomial is not given that explicity. What is the most implicit form that we can think of? A black box which evaluates the polynomial. That is, we have an oracle p when given input a returns $p(a)$, the value of polynomial at a.

Assume that we are also given an upper bound on the degree of the polynomial $deg(p) \leq d$. Indeed, we do not have access to the actual polynomial except through the blackbox. We have to use some property of the degree d polynomials. The most obvious one is the number of points in which they can evaluate to zero. Based on this thought, the following deterministic algorithm solves the problem.

```
1.  Choose $d+1$ different points $a_1, \ldots, a_{d+1}$.
2.  Call the oracle $d+1$ times to evaluate $p(a_1), \ldots, p(a_{d+1})$.
3.  If all calls returned 0 accept else reject.
```

Figure 1: A deterministic algorithm for univariate polynomial identity testing

If p were really 0 then all calls will return 0 and we will definitely accept. If p were not 0, then at most d calls can return 0 since a polynomial with degree at most d has at most d roots. Hence if $p \neq 0$, then our algorithm will definitely reject.

Now let us think about the problem when p is a multivariate polynomial. The previous assertion that a degree d polynomial has at most d roots no longer holds. To see this, consider the degree 2 polynomial $p(x_1, x_2) = x_1 x_2$. This has an infinite number of roots $x_1 = 0, x_2 \in \mathbb{F}$, where $\mathbb{F}$ is the (possibly infinite) field over which p is defined. We can work around this problem by considering a finite subset of the field, say $S = \{0, \ldots, 10\}$. The polynomial p has 19 zeroes. So if x_1, x_2 is chosen uniformly at random from S there is at most 19/100 chance that we will get a false result. As can be seen from the above example, by making the size of S arbitrarily large, we can make the error probability arbitrarily small. But then the disadvantage is that we will need more random bits in order to choose an element at random from the set $|S|$, and the running time of our algorithm will also increase.

In the next lecture, we will show that this intuition is correct by exhibiting a low error polynomial time randomized algorithm for the multivariate case. The question of finding a deterministic algorithm for this problem is open. Although it looks like a simple algorithmic problem from algebra which only mathematicians might be interested in, there are several computational problems that can be encoded into this form and hence can be solved efficiently if this algorithmic problem can be solved efficiently.