

Lecture 9 — October 07, 2004

Lecturer: Peter Gacs

BOSTON UNIVERSITY

Scribe: Alexandra Stefan

9.1 Introduction

There are two different uses of randomness in computer science. The first one is complexity-related, namely randomness is used in computing the average running time of an algorithm, over many inputs of the same size n . The second one, called randomized computation, has a more practical aspect: it uses random numbers in computations leading to a solution of a problem. In this course we study the latter one.

Let us see how randomness can be useful. Consider sorting an array $A = (a_1, \dots, a_n)$ using the Quicksort algorithm. The average time complexity of Quicksort is very good, namely $O(n \cdot \log n)$. But what is the worst case complexity? Remember what *Quicksort* (algorithm) looks like:

1. 'Pick' an index k from the set of all possible indexes, $\{1, \dots, n\}$.
2. Partition A into A_1 and A_2 such that all the elements in A_1 are less or equal to a_k and all the elements in A_2 are greater or equal to a_k .
3. *Quicksort*(A_1), *Quicksort*(A_2).

In order to have a valid algorithm, we need to specify how to 'pick' k . Let's take $k = 1$. For this algorithm, the average case complexity is

$$\frac{1}{n} \sum_{\pi} \text{time}(\pi(A)) = O(n \cdot \log n), \quad (*)$$

where π denotes possible permutations of A .

The worst case occurs when A is already ordered. The number of comparisons performed in this case is:

$$(n-1) + (n-2) + \dots + 1 = \frac{n(n-1)}{2},$$

therefore the time complexity is $O(n^2)$.

Moreover, typically, the arrays we need to sort are not random, but of a particular kind. Since the algorithm is more efficient on random arrays we can try to force some randomness. There are two ways of doing it. The first one is to mix the array before calling *Quicksort* on it. The second one is to change the algorithm to pick k randomly. Both approaches render *randomized* algorithms. Lets see what happens when k is randomly generated. For a fixed input array, A , there are various choices of k . We will average over them. Let *choice* denote a set of choices for each index k_1, \dots, k_n . Let $\text{time}(\text{choice } i)$ and $\text{Prob}(\text{choice } i)$ denote the time needed and the probability to generate the *choice*. It follows that the expected running time is:

$$\text{Prob}(\text{choice } 1) \cdot \text{time}(\text{choice } 1) + \text{Prob}(\text{choice } 2) \cdot \text{time}(\text{choice } 2) + \dots \quad (**)$$

which is proved to be also $O(n \cdot \log n)$. Note that this result is more valuable than the one given by (*), because not only it is good on average, but also it was computed for the particular kind of array that you had.

The relation (**) is given by the definition of the expected value from in probability theory: Given a random variable X , with possible values x_1, x_2, \dots with probability $\text{Prob}(X = x_1) = p_1$, $\text{Prob}(X = x_2) = p_2$, \dots , $\text{Prob}(X = x_n) = p_n$, then the expected value of X is

$$EX = p_1x_1 + p_2x_2 + \dots + p_nx_n.$$

9.2 Polynomial Identity

Interesting problems from different fields are solved using randomized algorithms. For example, in probability theory, the Monte-Carlo method applies randomization to approximate the integral of complicated functions.

In computer science randomization is often used to check whether a given function, f , is identically zero or not. The idea is to pick a few random inputs, x_1, \dots, x_n , and check whether $f(x_i) = 0$ for all i or not. Note that this algorithm is good only for functions f such that if f is not identically zero ($f \neq 0$) it must be that $f(x) \neq 0$ for most values x on which f is defined. Let us take a particular type of function f , namely a polynomial over x :

$$f(x) = a_0 + a_1x_1 + \dots + a_nx^n.$$

Theorem 1. *If f is a polynomial in one variable with degree n , then f has at most n roots unless $f = 0$.*

Let us assume that all coefficients a_0, \dots, a_n are integers and that we can only substitute x from $\{1, 2, \dots, N\}$ for some natural number, N . Since f has at most n roots, there can only be at most n roots in the set $\{1, 2, \dots, N\}$ and thus we have the following probability:

$$\text{Prob}(f(x) = 0) \leq \frac{n}{N},$$

which means that for a large N the chance to pick a root of f is very small.

One may argue that this is a fake problem, since it is trivial to check whether a polynomial is zero or not: you just have to check that all the coefficients are zero. But there are various cases when you are given some way to compute the function (without having the coefficients).

Now that we know that it is not a fake problem, let us analyze a more complex form of it: polynomials on several variables. We first give some helpful definitions. Given a term $t = a_i x_1^{p_1} x_2^{p_2} \dots x_m^{p_m}$, the degree of t is $d_t = p_1 + p_2 + \dots + p_m$. Given a polynomial, $f(x_1, \dots, x_m)$, on m variables, let d be the maximum of degrees in each variable of terms of f . We have the following theorem.

Theorem 2 (Schwartz Lemma). *Let f be a polynomial in m variables with degree d . If we substitute random integers $x_1, \dots, x_m \in \{1, \dots, N\}$, then*

$$\text{Prob}(f(x_1, \dots, x_m) = 0) \leq \frac{d \cdot m}{N}.$$

Proof: We use induction on the number of the variables. For the base case, polynomials on one variable, the property holds (see above).

For the induction step, assume the property holds for polynomials on $m - 1$ variables. Let us rewrite f as follows:

$$f(x_1, \dots, x_m) = a_0(x_1, \dots, x_{m-1}) + a_1(x_1, \dots, x_{m-1})x_m + \dots + a_d(x_1, \dots, x_{m-1})x_m^d = f'(x_m).$$

Assume $f \neq 0$. Then it must be that there exists i such that $a_i(x_1, \dots, x_{m-1}) \neq 0$. Let i be the largest such number. We have:

$$\text{Prob}(f(x_1, \dots, x_m) = 0) \leq \text{Prob}(a_i(x_1, \dots, x_{m-1}) = 0) + \text{Prob}(f'(x_m) = 0).$$

But we know that

$$\text{Prob}(a_i(x_1, \dots, x_{m-1}) = 0) \leq \frac{d \cdot (m - 1)}{N},$$

and that

$$\text{Prob}(f'(x_m) = 0) \leq \frac{d}{N}.$$

Therefore

$$\text{Prob}(f(x_1, \dots, x_m) = 0) \leq \frac{d \cdot m}{N}$$

□

Example 1 Rigidity of a large graph in 3 dimensions connected with edges of fixed length. Asking questions about it gives a matrix.

Example 2 Find a complete matching in a bipartite graph.

Solution: use the matrix that defines the graph

Claim: the determinant of this matrix is zero iff there is no complete matching.

Proof sketch: note that permutations different from zero correspond to complete matchings and consider the following formula for computing the determinant:

$$\det A = \sum_{\pi} \pm \prod_i a_{i\pi(i)}.$$

We have seen before that the determinant of a matrix can be computed in polynomial time using the Gaussian elimination. Theoretically this method is slightly better, $O(n^{2.4})$, than the fastest known deterministic solution, $O(n^{2.5})$. But the main advantage of this solution is that it is suitable to parallel computation: Gaussian elimination can be done faster than we have discussed by using parallel processors.

There is a generalization of the above theorem for non-bipartite graphs as well. It can be found in the Lovasz notes.

9.3 Branching Programs(BP)

definition from Sipser:

A Branching Program is a program that computes a boolean function. It takes as input a fixed number, n , of variables and at each step queries one of them and uses the answer to decide how to proceed. Thus it branches at every step. Such a program can be easily represented as a directed acyclic graph, in which every node is labeled with variable, except for two output nodes labelled with 0 and 1 respectively. The nodes labeled by the variables are called query nodes. Every query node has 2 outgoing edges. One of the nodes in a branching program is designated the start node.

Now that we know what a **BP** is, we ask the following question. Given two **BP**'s, P_1, P_2 over the same variables, x_1, \dots, x_n , are they equivalent: $P(x_1 \dots x_n) = P(x_1 \dots x_n)$? This turns out to be an NP-complete problem. Therefore we will consider a more simplified version of a **BP**: Read-Once Branching Program (**ROBP**). A **ROBP** is a **BP** in which each variable has at most one occurrence in any path in the graph. Given two **ROBP**, B_1 and B_2 , the above question can be answered using a polynomial randomized algorithm. The idea is to turn each one of the two **ROBP** into a polynomial, P_1 and P_2 respectively, and then check whether the polynomial $P_1 - P_2$ is identically zero. We have already seen that for such a problem we have a good randomized algorithm. One may argue that we could have used a similar randomized algorithm to generate some random binary inputs and to check the output of the two programs, but such an approach is not efficient due to the limited range of the input, 2^n , while the polynomials can be tested on a wider range of inputs since we no longer have to restrict ourselves to binary inputs. Such a method of turning a boolean formula into a polynomial and then using the later to substitute values outside $\{0, 1\}$, is called *arithmetization*. We will see that the resulting polynomials have maximum degree 1 in any variable, therefore, using the Schartz lemma we know that, if $P_1 \neq P_2$ we may mistakenly conclude that $P_1 = P_2$ with a probability less than n/N , where N represents the range in which we choose the test values.

Transforming a BP into a polynomial. The first step is to turn the **BP** into a boolean formula. Then the boolean formula will be turned into a polynomial. We will first assign a formula to each node as follows:

- The input nodes will take the value of the variable.
- If a node labelled x_i is associated the formula $f(x)$, then the outgoing branches of that node will be associated with formulas: $f(x) \wedge x_i$ and $f(x) \wedge \neg x_i$ respectively.
- If a node receives edges that are assigned the formulas $f_1(x), \dots, f_n(x)$ the node will be assigned the formula $f_1(x) \vee f_n(x)$.

It is easy to check that the above formula expresses the boolean function computed by the **BP**.

To turn the formula into a polynomial we use the following transformations:

$$\begin{aligned} \neg x &\rightarrow (1 - x) \\ x \wedge y &\rightarrow x \cdot y \\ f_1(x) \vee \dots \vee f_k(x) &\rightarrow f_1(x) + \dots + f_k(x) \end{aligned}$$

Clearly the polynomial computed in this way is 0 exactly for those inputs x for which the **BP** computes 0, but we want more than that. We want:

$$P_1 \equiv P_2 \iff F_1 \equiv F_2.$$

This property does not hold for a **BP**, but it holds for a **ROBP**. Expanding all conjunctions gives a DNF, but every term/clause has the form $b_1 \dots b_n$, where $b_i = 1$ or $b_i = x_i$ or $b_i = \neg x_i$. The polynomial is the sum of such these terms.

Claim 3. For P_1, P_2, F_1, F_2 defined above, the following holds:

$$P_1 \equiv P_2 \iff F_1 \equiv F_2.$$

Proof: We give just a proof sketch. First of all, notice the following problem: two DNFs can look different but be equivalent. To eliminate such differences, for every term $b_1 \dots b_n$, if $\exists i \in 1 \dots n, b_i = 1$ we replace it with $(b_i + (1 - b_i))$.

Now in our polynomial each term is of the form: $b_1 \dots b_n$ where $\forall i \in 1 \dots n, b_i = x_i$ or $b_i = (1 - x_i)$. Therefore each term corresponds to a satisfying assignment of the program we started from. It follows that $P_1 \equiv P_2 \iff F_1 \equiv F_2$. \square

Complexity considerations: Given an input x , $F(x)$ is computed in the same way the polynomial was constructed. Since we just compute with numbers, the result is not as large as the formal polynomial would be.

9.4 Randomized classes

Definition 4. Let L_0 be a witness language for a language $L \in NP$. We say that $L \in RP$ if for all x , if there is a witness y then the majority of potential witnesses is a witness.

More precisely, if $p(|x|)$ is a bound on the size of the witnesses of x , then we can conclude:

$$\frac{|\{y \mid (x, y) \in L_0\}|}{|\Sigma|^{p(|x|)}} > \frac{1}{2}$$

There is another special class of languages called *BPP*. It contains the languages that are recognized by probabilistic polynomial time Turing machines with an error probability of $\frac{1}{3}$.

A *probabilistic Turing machine* M is a type of a non-deterministic Turing machine where each non-deterministic step is called a *coin-flip step* and has two legal next moves. We assign a probability to each branch b of M 's computation on input w as follows. Define the probability of branch b to be

$$\Pr[b] = 2^{-k},$$

where k is the number of coin-flip steps that occur on branch b . Define the probability that M accepts w to be:

$$\Pr[M \text{ accepts } w] = \sum_{\substack{b \text{ is an} \\ \text{accepting branch}}} \Pr[b].$$

When a probabilistic Turing machine recognizes a language, it must accept all strings in the language and reject all strings out of the language as usual, except that now we allow the machine a small probability of error. For $0 \leq \epsilon \leq \frac{1}{2}$ we say that M recognizes language L with error probability ϵ if:

- $w \in L$ implies $\Pr[M \text{ accepts } w] \geq 1 - \epsilon$
- $w \notin L$ implies $\Pr[M \text{ rejects } w] \geq 1 - \epsilon$.

The effectiveness of these probabilistic algorithms can be *amplified*. Let $\frac{1}{2} < \epsilon < 1$ be the error probability. If we repeat the test for t times and as a result output the answer that was given more often by these tests, then the probability that this answer is wrong is less than ϵ_1^t where ϵ_1 is a constant, smaller than 1, that depends only on ϵ . For sufficiently large t this can be made arbitrarily small and it changes the expected number of steps only by a constant factor.