Course on Pseudorandomness

(Lecture Notes)

JAYALAL SARMA

Department of Computer Science and Engineering
Indian Institute of Technology Madras (IITM)
Chennai, India

Last updated on : February 13, 2021

Table of Contents

Todo list

Part I

General Introduction and Tools

Power of Randomization and Derandomization Problem

Lecturer : Jayalal Sarma

WEEK 1

Randomized algorithms play a very powerful role in algorithm design. We will concentrate on the randomized algorithms for decision problems in this course. So all of our computational problems can be abstractly represented as given a string $x \in \Sigma^*$ in an alphabet, does x have property $\mathcal{P}$ or not?

Informally, a randomized algorithm running in time t is an algorithm that that on input x is allowed to perform at most t instances of random experiment of tossing unbiased coins during its computation and uses the outcome of the experiment in the computation, but however, provides a guarantee that the answer of the algorithm is the *correct* answer for the input x in a good fraction of the possible outcomes of the experiment. [1]

1.1 Randomness helps in Matrix Multiplication Verification

Consider the task of multiplying two $n \times n$ matrices over the field $\mathbb{F}_2$. The trivial algorithmic solution to the problem takes $O(n^3)$ time and the trivial lower bound for the problem is $\Omega(n^2)$. It has been a long standing question which is the right complexity bound for this important problem (improvement to which will lead to improvements even in practice !). The exponent of matrix multiplication is the smallest constant ω such that two $n \times n$ matrices may be multiplied by performing $O(n^{\omega+\epsilon})$ for every $\epsilon > 0$.

Indeed, one of the basic ideas that we learn in algorithms courses for demonstrating the power of divide and conquer is the Strassen's multiplication which gives a running time bound of $O(n^{2.,73})$ which went through a sequence of improvements and to the current best of $O(n^{2.31})$.

The question that we address now is something closely related - that of verifying whether a given multiplicaiton is correct.

PROBLEM 1.1.1. *Given three matrices $A, B, C \in \mathbb{F}_2^{n \times n}$, check whether $AB = C$ or not.*

Indeed, the trivial method would be to mutliply the two matrices and check if the result is equal to C. But then this requires, $O(n^{2.31} + n^2)$ time using the best matrix multiplication algorithm that we know currently. Since we just require verification, it is conceivable that we might be able to do better if we are allowed to make a small some error in the process. We show that this indeed possible to be done in $O(n^2)$ with error probability at most 2^{-k} for any constant k (independent of

[1] Notice that if the guarantee for the algorithm is not saying "*strictly more than half fraction of the coin tosses*", then essentially the algorithm is useless since we can always replace it with a random experiment of tossing an unbiased coin and returning the answer to be YES if we get the heads and NO if we get tails. Note that we have at least a $\frac{1}{2}$ probability of success $\frac{1}{2}$.

n).

Trivial Approach using randomization: A natural first cut attempt is to choose an entry $(i, j) \in [n] \times [n]$ uniformly at random from the n^2 entries of C and checking if :

$$\sum_{k=1}^{n} A_{ik} B_{kj} = C_{ij}$$

This runs in time $O(n)$. And we can choose constant k more entries to amplify the success probability. However, the probability of correctness is very small. Suppose, in the worst case input, there was only one $(i, j) \in n \times n$ where there was an error in the multiplication. The probability that we will choose that particular (i, j) for verification is as small as $\frac{1}{n^2}$. Amplifying this to a success probability of $\frac{1}{2^k}$ takes more than $\Omega(n^{1+\epsilon})$ iterations and hence the overall algorithm will take $\Omega(n^{2+\epsilon})$ time which is beyond what we can afford to spend time on.

Freivalds' Approach: The idea is to check a randomly chosen "linear combination" of entries rather than a single entry of the matrix C. If we choose this to be a random linear combination of rows of the matrix C, then the combinatorics helps to achieve a much better probability of error. We now formally write down the algorithm and analyse it.

Algorithm 1.1 : Frievald's Algorithm for Verification of Matrix Multiplication - $\mathcal{F}(A, B, C)$

1: Choose a vector $r \in \mathbb{F}_2^n$ uniformly at random.
2: If $[A(Br) = Cr]$ then output YES else output NO.

The computation of $(A(Br))$ and Cr are done using $O(n^2)$ time algorithms since computing a linear transformation result Ax for an $n \times n$ matrix can be done in $O(n^2)$ time. Now we argue correctness guarantees. If the given matrices indeed satisfy $AB = C$, then no matter which $r \in \mathbb{F}_2^n$ algorithm chooses in step 1, $ABr = Cr$ and hence it always will output YES. The error can happen only when $AB \neq C$ and the algorithm ends up choosing an unfortunate r such that $ABr = Cr$. The following claim upper bounds the probability of this .

CLAIM 1.1.2. *For any $A, B, C \in \mathbb{F}_2^n$ such that $AB \neq C$,*

$$\Pr[\mathcal{F}(A, B, C) \textit{ outputs } \text{YES}\,] \leq \frac{1}{2}$$

Proof. Let $A, B, C \in \mathbb{F}_2^n$ such that $AB \neq C$. We need to analyze the probability that $ABr = Cr$ for $r \in \mathbb{F}_2^n$ chosen uniformly at random. If $AB \neq C$, then $D = AB - C$ is a non-zero matrix. Thus

$$\Pr_{r \in \mathbb{F}_2^n}[ABr \neq Cr] = \Pr_{r \in \mathbb{F}_2^n}[Dr \neq 0]$$

Imagine that D was all 1s matrix. Now the above probability is exactly the number of vectors $r \in \mathbb{F}_2^n$ with an odd number of 1s in it. By an obvious bijection, this is also the number of subsets of $[n]$ with odd size. The latter is exactly 2^{n-1} and hence the probability of such a vector r being chosen from $\mathbb{F}_2^n$ is $\frac{1}{2}$.

Now we formalize and generalize this. Let p be the vector Dr. Since $D \neq 0$, there must be an entry $D_{ij} \neq 0$. Define, $A = \{j : D_{ij} \neq 0\}$. We know that $A \neq \phi$. $i, j \in [n]$. Thus :

$$p_i = \sum_{k=1}^{n} D_{ik} r_k = \sum_{k \in A} r_k$$

$$\text{Note that: } \Pr_{r \in \mathbb{F}_2^n}[Dr = 0] \leq \Pr_{r \in \mathbb{F}_2^n}[p_i = 0]$$

Notice that the latter probability depends only on r_k where $k \in A$. The fraction of assignments of the bits $\{r_k : k \in A\}$ which makes $p_i = 0$ is exactly $\frac{1}{2}$ since the number of even sized subsets of A and number of odd sized subsets of A are exactly the same. □

REMARK 1.1.3. Informally, if we run the above algorithm $\mathcal{F}(A, B, C)$, and the algorithm outputs NO, then we can trust the answer and conclude that indeed $AB \neq C$. But a YES answer from the algorithm cannot be trusted - it could be because of the unfortunate choice of $r \in \mathbb{F}_2^n$ that came as the outcome of the experiment.

Why are we interested in the above algorithm even though it gives a success probability bound of only $\frac{1}{2}$? The reason is that, it is a one-sided error algorithm and hence still much better than a coin toss outcome because the algorithm does not make an error when $AB = C$. In fact, such algorithms can be repeated in a natural way - run k times, and if any of them says $AB \neq C$ output NO. This reduces the error probability in exponentially in k - since each of the trials should give an error (with probability $\frac{1}{2}$) and hence the error probability bound is at most $\frac{1}{2^k}$.

1.2 Polynomial Identity Testing Problem

The previous example, while it demonstrates the point, might be a bit unsatifactory since the problem under consideration anyway has an efficient algorithm. To address this, we will now see another example problem where there is an efficient (polynomial time in the input size) randomized algorithm for solving the problem but a deterministic algorithm for solving the problem is not known.

The problem is easy-to-state algorithmic question on polynomials. Fix $\mathbb{F}$ to be the field where the coefficients are chosen from. *Given a polynomial $p \in \mathbb{F}[x_1, x_2, \ldots x_n]$, test if it is identically zero.* That is, do all the terms cancel out and become the zero polynomial.

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. How can it be given implicitly compared to the list of monomials? One answer is that, we could give it in a bracketed form. That is, the polynomial $x_1x_2 + x_1x_4 + x_3x_2 + x_3x_4$ can be given as $(x_1 + x_3)(x_1 + x_4)$. Indeed, this is implicit, since an expression of length n, can have number of actual number of monomials to be 2^n - consider the example $(x_1 + x_2)(x_3 + x_4)\dots(x_{n-1} + x_n)$ where n is the number of variables.

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.

Algorithm 1.2 A deterministic algorithm for univariate polynomial identity testing

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

If p were really the zero polynomial 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.

Multivariate PIT: 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 $p(x_1, x_2) = x_1x_2$. 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 $S = \{0, \dots, 10\}$. 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.

Generalizing this strategy that we will follow is as follows: If the total degree of the polynomial is $\leq d$, and if $S \subseteq \mathbb{F}$, such that $|S| \geq 2d$, instead of picking elements arbitrarily, we pick elements uniformly at random from S. Indeed, there may be many choices for the values which may lead to zero. But how many?

LEMMA 1.2.1 (**Schwartz-Zippel Lemma**). *Let $p(x_1, x_2, \cdots, x_n)$ be a non-zero polynomial over a field $\mathbb{F}$. Let $S \subseteq \mathbb{F}$*

$$\Pr_{\bar{a} \in S^n}[p(\bar{a}) = 0] \leq \frac{d}{|S|}$$

Proof. (By induction on n) For $n = 1$: For a univariate polynomial p of degree d, there are $\leq d$

roots. Now in the worst case the set S that we picked has all d roots. Thus for a random choice of substitution for the variable from S, the probability that it is a zero of the polynomial p is at most $\frac{d}{|S|}$.

For $n > 1$, write the polynomial p as a univariate polynomial in x_1 with coefficients as polynomials in the variables $p(x_2, \ldots, x_n)$.

$$\sum_{j=0}^{d} x_1^j p_j(x_2, x_3, \ldots, x_n)$$

For example: $x_1x_2^2 + x_1^2x_2x_3 + x_3^2 = (x_2x_3)x_1^2 + (x_2^2)x_1 + x_1^0(x_3^2)$.

We need to analyze the probability that we will choose a zero of the polynomial (even though the polynomial is not identically zero). For a choice of the variables as $(a_1, a_2, \cdots, a_n) \in S^n$, we ask the question : how can $p(a_1, a_2, \cdots, a_n)$ be zero? It could be because of two reasons:

1. $\forall j \ : \ 1 \leq j \leq n, \ \ p_j(a_2, a_3, \ldots, a_n) = 0.$
2. Some coefficients $p_j(a_2, a_3, \ldots, a_n) = 0$ are non-zero, but the resulting univariate polynomial in x_1 evaluates to zero upon substituting $x_1 = a_1$.

Now we are ready to calculate $Pr[p(a_1, a_2, \ldots, a_n) = 0]$. For a random choice of $(a_1, \ldots, a_n)$. Let A denote the event that the polynomial $p(a_1, \ldots, a_n) = 0$. Let B denote the event that $\forall j \ : \ 1 \leq j \leq n, \ \ p_j(a_2, a_3, \cdots, a_n) = 0$. Note that, $Pr[A] = Pr[A \wedge B] + Pr[A \wedge \overline{B}]$.

We calculate both the terms separately: $Pr[A \wedge B] = Pr[B].Pr[A|B] = Pr[B]$ where the last equality is because $B \Rightarrow A$. Let ℓ be the highest power of x_1 in $p(x)$. That is $p_\ell \neq 0$. Since the event B insists that for all j, $p_j(a_2, a_3, \ldots, a_n) = 0$, we have that $Pr[B] \leq Pr[p_\ell(a_2, a_3, \ldots, a_n) \neq 0]$. By induction hypothesis, since this polynomial has only $n-1$ variables and has degree at most $\frac{d-\ell}{S}$. Thus, $Pr[B] \leq \frac{d-\ell}{S}$.
To calculate the other term,

$$Pr[A \cap \overline{B}] \quad = \quad Pr[\overline{B}].Pr[A|\overline{B}] \leq Pr[A|\overline{B}] \leq \frac{\ell}{|S|}$$

where the last inequality holds because the degree of the non-zero univariate polynomial after substituting for $a_2, \ldots, a_n$ is at most ℓ and hence the base case applies. □

This suggests the following efficient algorithm for solving PIT. Given d and a blackbox evaluating the polynomial p of degree at most d.

Algorithm 1.3 : Schwartz-Zippel Algorithm for Multivariate PIT

1: Choose $S \subseteq \mathbb{F}$ of size $\geq 4d$.
2: Choose $(a_1, a_2, \ldots, a_n) \in_R S^n$.
3: Evaluate $p(a_1, a_2, \ldots a_n)$ by querying the blackbox.
4: If it evaluates to 0 accept else reject.

The algorithm runs in time $\mathsf{poly}(n)$. The following Lemma states the error probability and follows from the Schwartz-Zippel Lemma that we saw before.

LEMMA 1.2.2. *There is a randomized polynomial time algorithm A, which, given a black box access to a polynomial p of degree d (d is also given in unary), answers whether the polynomial is identically zero or not, correctly with probability at least $\frac{3}{4}$.*

Notice that in fact the lemma is weak in the sense that it ignores the fact that when the polynomial is identically zero then the success probability of the algorithm is actually 1 !. In other words, is it is a one-sided error randomized algorithm.

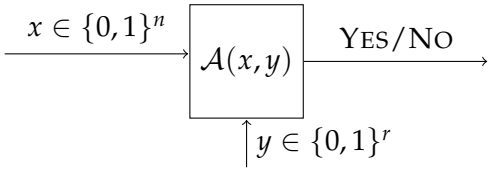
1.3 Derandomization Problem

In the previous section (lecture), we talked about randomized algorithms for problems for which we do not know deterministic algorithms with similar complexity resource bounds. Indeed, we are not happy about randomized algorithms as such since these algorithms require perfect unbiased coin toss experiments to be performed and we do not have them in practice. Indeed, the fact that they can output erroneous answers, even though with low probability makes them useless in critical practical applications.

How do convert them to deterministic algorithms without causing much overhead?. One possible way is to look at each algorithm and use inherent properties of the problem to analyze the randomized algorithm better to come up with ways to remove randomness from that algorithm. Here, we start with the original randomized algorithm for a particular problem, and improve it to derandomize it and the techniques are usually very algorithm specific. We will do some examples of this kind later in the course.

1.3.1 Abstract Model of Derandomziation

From now on, we will be concentrating only on abstract models of these randomized algorithms. We fix some notations first. A randomized algorithm $\mathcal{A}$ on input x runs in time $t(n)$ (where $n = |x|$) and let $y \in \{0,1\}^{r(n)}$ be the concatenation of the unbiased coin toss experiement that the algorithm does during its execution. Notice that $r(n) \leq t(n)$ (we drop the n when it is not required explicitly). If the algorithm runs in polynomial time $t(n) \leq n^c$ for a constant c independent of n.



The guarantee we have is there is an $\epsilon \in (0, \frac{1}{2}]$.

$$\forall x \in \{0,1\}^n, \Pr_{y \in \{0,1\}^r}[A(x,y) \text{ is correct.}] \geq \frac{1}{2} + \epsilon$$

1.3.2 Derandomization by Brute Force Approach

The trivial approach to obtain an equivalent deterministic algorithm is run over all possible outcomes of the experiment and check the answer from the algorithm for each of them. Whichever answer comes as majority - report that as the final answer.

Algorithm 1.4 ($\mathcal{A}'$) : input $x \in \{0,1\}^n$, where success prob. $\frac{1}{2} + \epsilon$ for $\mathcal{A}$

1: $count \leftarrow 0$.
2: **for** `each` $y \in \{0,1\}^r$ **do**
3: `Check if` $\mathcal{A}(x,y)$ `accepts, if so increment` *count*
4: **end for**
5: If $[count > 2^{r-1}]$ then output YES else output NO.

If the running time of the randomized algorithm $\mathcal{A}$ is $t(n)$, then the running time of the new algorithm (which is deterministic) is $t(n)2^{r(n)}$. To argue correctness, if the actual answer for input $x \in \{0,1\}^n$ is YES, then the fraction of $y \in \{0,1\}^r$ which makes $\mathcal{A}(x,y)$ accept is strictly more than $\frac{1}{2}$ and hence the algorithm will output YES. If the actual answer for input $x \in \{0,1\}^n$ is NO, then then the fraction of $y \in \{0,1\}^r$ which makes $\mathcal{A}(x,y)$ accept is strictly less than $\frac{1}{2}$ and hence the algorithm will output NO.

REMARK 1.3.1. Note that the algorithm $\mathcal{A}$ will run in $\mathsf{poly}(n)$ time if the original randomized algorithm was running in $t \leq \mathsf{poly}(n)$ time and was using $r \leq O(\log n)$ random bits.

1.4 Pseudorandomness : An Informal Overview

Ideally, we would like to replace the randomized algorithm with a deterministic one as done in the previous section. However, we know how to do this trivially only when the randomized algorithm uses $O(\log n)$ random bits.

We outline two "out of the box" thoughts related to our target of derandomization of randomized algorithms.

Fooling the Algorithm with Pseudorandom bits : PRGs The first one is about using $y \in \{0,1\}^r$ as not independent random bits. But use *dependent* random bits instead. Indeed, the analysis for the error bound for the algorithm $\mathcal{A}$ now may fail since it may assume total independence between the bits of y in its mathematical argument. However, sometimes, it is possible that same analysis (or even a better analysis) may work even when the bits of the y are dependent in a limited way [2] But this may be specific to the algorithm and sometimes to the problem itself. We would ideally want a more abstract strategy which would work for randomized algorithms in general, modelled by what we described in the previous section. But even if we made it work with some dependent randombits, how do we produce this distribtion of y' with the desired limited dependence among them? Construction of the methods which can produced limited dependence thus becomes important.

Taking a more abstract view point, informally, we would like to have a box (formally an algorithm G) which takes in pure random bit string of length $y' \in \{0,1\}^{r'}$ and produces a string $y \in \{0,1\}^r$ such that the distribution of y "looks" pseudo-random for the resource limited algorithm $\mathcal{A}$. The idea is that we will run the algorithm $\mathcal{A}$ with the random string provided the $G(y')$

[2] At one extreme, if we had an algorithm and an analysis which works wen all the bits of the y are the same (which is an example of extreme dependence) we dont require the random bit at all - we can directly simulate the algorithm deterministically the trivial way.

- the output of G on input $y' \in \{0,1\}^{r'}$ chosen uniformly at random.

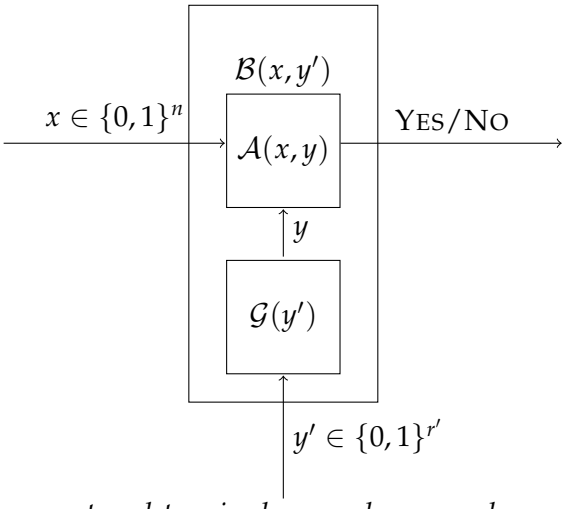
Indeed, since we are not providing pure random bits to $\mathcal{A}$. Hence we should expect its correctness guarantees to not hold good anymore. That is, it will deteriorate a bit. Can we guarantee that it does not deteriorate too much? This can be done in two ways (1) rework or reanalyse the algorithm $\mathcal{A}$ and argue that it is still having success probability greater than $\frac{1}{2}$ (which is enough for trivial derandomization) (2) use only resource bounds of $\mathcal{A}$ to argue that the change of y to $G(y')$ will not affect the success probability much. For this, the generator has to satisfy certain properties.

A function $G : \{0.1\}^{r'} \to \{0,1\}^{r}$ is said to be a Peudo-Random Generator (PRG) for complexity measure[3] s and error parameter $\delta \in [0,1]$ if, for any algorithm $\mathcal{A}$ which runs in time $t \leq s$ (or having complexity measure bounded by s): For any x,

$$\left| \Pr_{y \in \{0,1\}^r}[\mathcal{A}(x,y) \text{ Accepts}] - \Pr_{y' \in \{0,1\}^{r'}}[\mathcal{A}(x, G(y')) \text{ Accepts}] \right| \leq \delta$$

Connecting to the informal description, δ is the quantity by which the success probability deteriorates because of the use of the pseudorandom generator output, instead of pure random bits. Hence if we ensure that $\delta < \epsilon$, even after the use of the pseudorandom generator output, we still will have a randomized algorithm $\mathcal{B}$ with the following guarantee $\forall x \in \{0,1\}^n$:

$$\Pr_{y \in \{0,1\}^r}[\mathcal{B}(x,y) \text{ is correct.}] \geq \frac{1}{2} + \epsilon - \delta > \frac{1}{2}$$



We will end the description by asking the question - *what parameters determine how good our pseudrandom generator is?*. As per the above discussion it is:

- The relative values of r and r'. This leads to the definition of the *stretch* of the psuedorandom generator. We would ideally want an exponential stretch function so that with $r' \in O(\log n)$ we can produce y for $\mathcal{A}$ which is of length $r = O(n^c)$ for constant c.
- The value of s. This determines how powerful an algorithm can the pseudorandom generator manage to fool. The larger the s the better. Ideally we want s to be covering all polynomial time running time bounds.
- The value of δ. This determines the quantity by which the success probability of the algorithm deteriorates after plugging in the output of the pseudorandom generator instead of the y from the pure random bits. Ideally, we want $\epsilon - \delta > 0$. The smaller the δ, the better.

[3]We will make it more precise when it comes to the section where we handles these objects. We are leaving at the above description at a less precise level.

- Running time of the generator itself. Notice that we need $\mathcal{G}$ to be explicit polynomial time algorithm, which runs in time $\mathsf{poly}(n)$. That is, if $r' \in O(\log n)$ (which is what ideally we would want, so that the trivial derandomization runs in $\mathsf{poly}(n)$ time), then technically, the generator can run for exponential time in terms of its input size[4].

The main part of the game is in describing the generator algorithms (or functions from $\{0,1\}^{r'} \to \{0,1\}^{r}$). However, it is not even clear whether such functions exists for the range of parameters that we care about. Indeed, this is the kind of flavour that we will have.

- We can prove that the functions that we are looking for exist, with a non-constructive argument. This is done by - what is termed as the *Probablistic method*.
- Explicit descriptions of the functions, which are are required for the algorithms with required runtime bounds for $\mathcal{G}$ are not known. In fact, if we have such descriptions, then a complete derandomization of all randomized algorithms is possible, which will be a big achievement.

Refining Randomness : Randomness Extractors - Here is a completely different idea about supplying dependent randomness. We do not have source of pure random bits to supply for the randomized algorithm $\mathcal{A}$. But we may have impure radom bit sources. An imaginative question is *can we invest a few pure random bits in order to purify/extract and hence improve the impurity in the given random source?*. This, at first sounds crazy and leads to the following questions.

- How do we define *impure random bit sources*. They define distributions which are not uniform. There is the notion of entropy which can tell us how uniform the source is.
- How do we define *how good the output is*. Again, one could have used entropy here too naturally. However, noticing the fact that we would like to finally apply it our algorithms like $\mathcal{A}$, a different measure of "purity" is used which is the notion of statistical distance[5] to uniform distribution.

The above discussion leads to the definiton of a randomness extractor, which is a function $\mathcal{E} : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m$ such that when X is a distribution on $\{0,1\}^n$ with entropy at least k, then the distribution of the output $\mathcal{E}(X, U_d)$ is ϵ-close to U_m where U_d and U_m are uniform distributions on the set $\{0,1\}^d$ and $\{0,1\}^m$ respectively.

Again, how do we determine how good is our extractor function? We want extractors which works on highly biased distributions (the smallest k possible) using fewest number of pure random bits (the smallest d possible) and produces output distributions which are closest to uniform distributions (ϵ must be smallest) - and still run in time polynomial in n.

Similar to pseudorandom generator functions, it is unclear apriori whether such functions even exist for the range or parameters we care about. A similar situation arises, where by using probablistic method, we can prove that such objects (functions) exists, but at the same time, we do not know how to construct them deterministically (equivalently describe the algorithm for $\mathcal{E}$).

[4] This marks the difference between the pseudorandom generators studied in cryptography and derandomization.

[5] Informally, this is the sum of the difference (in absolute value) between the probability values assigned to points in the sample space.

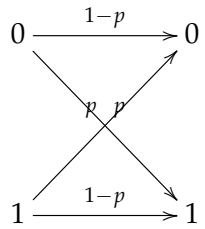
REMARK 1.4.1 ((Informal) - **Psuedo-random Objects**). The common flavour that we observed about the previous sections is that there are mathematical objects which we would like to describe (by providing algorithms to compute those functions) and we do know that the function that we seek exists. This situation is a common phenomenon in many objects. In fact, in most situations, it is not just the existance of the objects that is argued, but also that if the object that is of interest is chosen at random (with appropriately set up experiments), the object of interest shows up at the outcome with high probability. Thus, there is a randomized algorithm to explicitly construct the object, and now we have to derandomize them !. However, notice that in such situations, we can think of deranodomizations which just depends on those algorithms which chooses the object at random.

Other Contexts and Psuedorandom Objects We now describe a totally unrelated context in which the required mathematical functions display such a psuedorandom behaviour and an explicit construction is being sought for. The context is that of coding theory.

Coding theory had its inception in the late 1940's with the theory of reliable communication over a channel in the presence of noise - an area that started with the pioneering work of Claude Shannon and Richard Hamming. The former addressed and answered the fundamental questions about the possibility of the use of codes for reliable communication and the later develped some basic combinatorial constructions of error correcting codes that laid the foundations for the work later.

Theoretical computer scientists have a major role to play in the algorithmic aspects of coding theory research, and coding theory has proved to be instrumental in several interesting results in theoretical computer science as well. There has been several surprising applications of codes and the associated mathematical objects, in areas like algorithms, complexity theory and cryptography. Some part of the course will aim to discuss of those applications. However, we do not intend to be exhaustive.

The channel is not harmless in the real world. It introduces errors in the transmission. Depending on the application the error may be in the physical storage media (communication over time) or in the physical channel (communication over space). Some of the 0s gets flipped to 1s and vice versa, and some bits may get dropped too. For the purposes of this course we will study only model (Shannon studied several interesting variants), namely what are called Binary Symmetric Channels. In this model, each bit gets flipped with a probability p. That is, a 1 gets flipped to a 0 with probability and 0 gets flipped to 1 with probability p.



What is the natural strategy to cope up with errors in transmission? Create redundancy. For example, if Alice wants to send a bit 0 to Bob, she will do it five times, and send 11111 and ask Bob to take the majority of the bits as the bit that was sent. In this simple looking example we

have all the essence. The string that was sent will be called the *codeword* and the original bit to be sent is called the *message*. There are only two codewords 00000 and 11111 in the above example. If we define the notion of distance as the hamming distance, then the majority decoding mechanism described above can also be seen as choosing the codeword that is closest to the received word. This natural strategy of decoding is called *nearest neighbor decoding* or *maximum likelyhood decoding*.

Now let us observe facts about guarantees. Clearly if the channel is such that it will not corrupt more than 2 bits in a sequence of 5 bits, then Bob will be able to decode the message bit correctly. But the channel may actually flip more number of bits but with relatively lower probability. Thus if we increase the number of copies we make of the original message, with high probability (over the errors) introduced by the channel we are going to be able to decode the bit correctly.

To fix some notations, we denote $E : \{0,1\}^k \to \{0,1\}^n$ as the encoding function where k is the message length (in general) and n is the length of the codeword (which we will call the *block length*. Let $m \in \{0,1\}^k$ be a message, and $E(m) \in \{0,1\}^n$ is the transmitted word. The channel corrupts the message and let $y \in \{0,1\}^n$ is the received word. The error introduced by the channel could also be thought of as a string $\eta \in \{0,1\}^n$ where the η_i determines whether $y_i = (E(m))_i$ or not.

We want the following guarantee for any $m \in \{0,1\}^k$ as translating the above intuition:

$$Pr_\eta(D(E(m) + \eta) = m) \geq 1 - o(1)$$

where the $o(1)$ term is exponentially small depending on n and hence on k (since c is a constant).

Although the above statement is written in terms of a probability over choice of the channel error vector, a natural combinatorial guarantee that we would want is an encoding and decoding scheme such that if the error string η has weight at most $t < \frac{d}{2}$ the decoder retrieves the message correctly. That is, the encoder-decoder pair is guaranteed to get the message across the channel, if the number of corrputions by the chaneel is limited a number t. Indeed, the relative redundant information we sent should be minimised (which is the ration of k and n called the rate of the code).

Shannons theorem essentially states that under sutiable choice of the parameters there is a pair of encoding-decoding functions that can achieve this high confidence decoding of the original message. We will state the theorem formally only later. But again, the spirit of the theorem is that there does exist good encoding and decoding schemes with respect to the parameters we usually care about (which we make precise later). The area of algorithmic coding theory essentially attempts to address the question of constructing coding schemes for which there is an efficient decoding.

We conclude the lecture by stating that the three mathematical objects that we stated in this lecture do have some interconnections among themselves and also the pseudorandom objects that we are going to state in the next lecture too.

Method of Conditional Expectation

Lecturer : Jayalal Sarma

WEEK

2

2.1 MAXCUT Problem

For an undirected graph $G(V, E)$, a cut is a partition of vertices into two sets $S, T \subseteq V$. The size of a cut is the number of edges that go across these partitions. That is, the size of the set :

$$cut(S, T) = \{e = (u, v) \mid (u, v) \in E, u \in S, v \in T\}$$

Maximum cut is a cut whose size is at least the size of any other cut. That is $|cut(S, T)|$ is the largest possible. Given a graph, the problem of finding a maximum cut in a graph is known as the MAXCUT problem.

The problem is hard: The MAXCUT problem is known to be NP-hard. This implies, in particular, that if we have an efficient algorithm for the MAXCUT problem, then some of the very hard problems will yield to having efficient algorithms solving them. This is believed to be unlikely.

Approximation algorithms: Hence it makes sense to talk about algorithms which may not output the exact maximum cut, but instead another cut. Indeed, this is useless unless there are guaranteed how large is the cut output by the algorithm. An example guarantee that we may want to target is, for the algorithm, no matter what the input graph G is, the cut output by the algorithm will be, say, at least $\left(\frac{1}{10}\right)$-th of the size of the maximum cut. This is called a 0.1-approximation algorithm[6]. Even with this relaxed target for the algorithm, is not immediately clear how to design such an algorithm. It turns out that the best known algorithm for MAXCUT does much better than this and achieves an approximation guarantee of 0.875, and for many reasons this is believed to be the best possible ratio that any polynomial time algorithm can achieve for MAXCUT problem. However, in this lecture, we will concentrate on much smaller rations.

Randomized Approximation algorithms: We resort to randomized algorithms - which in this context will be called randomized approximation algorithms. Notice that unlike the previous examples, MAXCUT is not a decision problem. Hence, we need to be careful about designing and analysing randomized algorithms for it. For example, there is nothing like the algorithm being correct. Instead, we have only the notion of the approximation ratio - that is how close the output of the algorithm is, to the optimal value.

[6]Exercise: if you have not seen it already, think about what would be a similar statement that you would like to target for a minimization problem, like vertex cover problem

2.1.1 A Simple Randomized Algorithm

We start with a simple randomized algorithm for MAXCUT problem.

Algorithm 2.5 : Randomized Approx. Algorithm for MAXCUT for graph $G(V, E)$, $|V| = n$

```
S = T = φ
for each i ∈ [n] do
    Choose bit b_i ∈ {0, 1} uniformly at random.
    if b_i = 1 then
        S = S ∪ {i}
    else
        T = T ∪ {i}.
    end if
end for
Output cut(S, T).
```

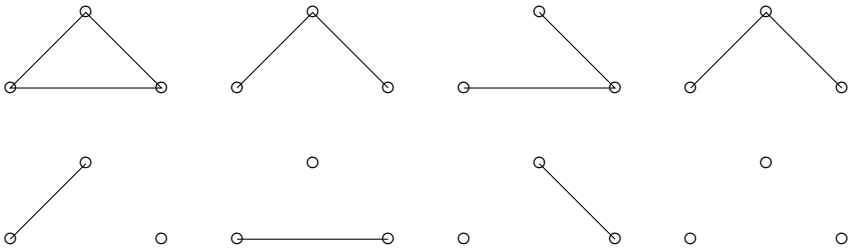
Notice that the sets S and T will form a partition of V at the end of the algorithm. Clearly, the algorithm will run in linear time. Indeed, the above description is also equivalent to - choose a random subset S of vertices from V and outputing $cut(S, \overline{S})$. Indeed, we need to give a guarantee about the size of the cut output by the algorithm. Roughly, the claim is that the average size of the cut (where average is taken over the 2^n different outcomes of the random choices). We recap some basics of random variables and expections now before stating the correctness claim.

2.2 Recap of Probability Basics, Random Variables, Expectation

Fix a set Ω, which is called the *sample space*. A probability distribution is defined over Ω, is a function $\Pr : \Omega \to [0, 1]$ satisfying the additional condition that, $\sum_{w \in \Omega} \Pr(w) = 1$. An event is a subset $\mathcal{E} \subseteq \Omega$. The probability of an even $\mathcal{E}$ is nothing buy the sum of the probability values by assigned by the distribution functon to the elements in the subset $\mathcal{E}$. That is, $\Pr(\mathcal{E}) = \sum_{w \in E} \Pr(w)$.

We consider an example which are going to be relevant for us. This is the notion of random graphs. Consider n vertices, and there are $\binom{n}{2}$ possible edges. Imagine that we will choose (independently) each edge to be present in our graph with probability p and to be absent in our graph with probability. The outcome of the experiment is an n-vertex simple graph and hence the sample space is the set of all n vertex graphs.

As an example, we can consider, $n = 3$. There are 3 possible edges and hence 8 possible graphs. The probability assigned the triangle graph (which is the complete graph on 3 vertices) is p^3 since all the three edges have to be chosen for this particular outcome to happen. In a similar way, the following pictures denote the sample space in this case with the corresponding probability values.



The probabilities in that order are p^3, $p^2(1-p)$, $p^2(1-p)$, $p^2(1-p)$, $p(1-p)^2$, $p(1-p)^2$, $p(1-p)^2$, $(1-p)^3$.

How do we analyse the probability that we get a connected graph as the outcome of the experiment. This is where events are used. Recall that formally, an event $\mathcal{E}$ is a subset of Ω.

$$Pr(\mathcal{E}) = \sum_{w \in \mathcal{E}} Pr(w)$$

In the above example, if the event $\mathcal{E}$ represent the set of connected graphs.

$$Pr(\mathcal{E}) = p^3 + 2p^2(1-p)$$

In the above example, if the event $\mathcal{E}'$ represent the set of bipartite graphs.

$$Pr(\mathcal{E}') = 1 - p^3$$

PROPOSITION 2.2.1 (**Subadditivity of Probability - a.k.a - Union theorem**). *Let $\mathcal{E}_1$,$\mathcal{E}_2 \ldots \mathcal{E}_n$ be events, then :*

$$Pr\left[\bigcup_i \mathcal{E}_i\right] \leq \sum_{i=1}^{n} Pr[\mathcal{E}_i]$$

DEFINITION 2.2.2 (**Conditional Probability**). *For two events $\mathcal{E}$ and $\mathcal{E}'$, we define,*

$$Pr(\mathcal{E}|\mathcal{E}') = \frac{Pr(\mathcal{E} \cap \mathcal{E}')}{Pr(\mathcal{E}')}$$

The conditional probability captures the questions of the kind, what is the probability that we get a connected graph if we are given that the outcome is a bipartite graph?

DEFINITION 2.2.3 (**Independent events**). *Two events $\mathcal{E}$ and $\mathcal{E}'$ are said to be independent, if*

$$Pr(\mathcal{E}|\mathcal{E}') = Pr(\mathcal{E})$$

Equivalently,

$$Pr(\mathcal{E} \cap \mathcal{E}') = Pr(\mathcal{E})Pr(\mathcal{E}')$$

For example, if we consider the events event $\mathcal{E}$ represent the set of connected graphs and event $\mathcal{E}'$ represent the set of bipartite graphs, then:

$$Pr(\mathcal{E} \cap \mathcal{E}') = 3p^2(1-p)$$

$$Pr(\mathcal{E})Pr(\mathcal{E}') = \left[(1-p)^3 + 3p(1-p)^2 + 3p^2(1-p)\right](1-p^3)$$

Since they are not equal, we conclude that the two events are not indendent. That is, the event that the graph is bipartite has an "influence" on the event that the graph is connected. To make this clearer, we suggest the following exercise:

Exercise 2.3. Let $G \in G(n,p)$. For all $S \subseteq V$, let A_S be the event that S forms an independent set in G. Show that if S and T are two distinct subsets of k vertices then A_S and A_T are independent if and only if $|S \cap T| \leq 1$.

Now, we will generalize the above notion of independence to more than two events. An event $\mathcal{E}$ is independent of a set of events $\{\mathcal{E}_j \mid j \in J\}$ if, for all subset $J' \subseteq J$, $Pr[\mathcal{E}| \cap_{j\in J} B_j] = Pr(A)$.

Exercise 2.4. Prove that an event $\mathcal{E}$ is independent of a set of events $\{\mathcal{E}_j \mid j \in J\}$ if and only if for all $J_1, J_2 \subseteq J$ such that $J_1 \cap J_2 = \phi$

$$\Pr[\mathcal{E} \cap \left(\cap_{j\in J_1} B_j\right) \cap \left(\cap_{j\in J_2} \overline{B_j}\right)] = \Pr(\mathcal{E}) \Pr[\left(\cap_{j\in J_1} B_j\right) \cap \left(\cap_{j\in J_2} \overline{B_j}\right)]$$

Let $\{\mathcal{E}_i \mid i \in I\}$ be a (finite) set of events. They are *pairwise independent* if for all $i \neq j$ the events $\mathcal{E}_i$ and $\mathcal{E}_j$ are independent. Events are *mutually independent* if each of them is independent from the set of the others. It is important to note that events may be pairwise independent but not mutually independent. Following exercise demonstrates that.

Exercise 2.5 (See Problem Set 1(Problem 1))**.** A random k-colouring fo a graph G is an element of the probability space (Ω, Pr) where Ω is the set of all k-colourings (i.e. partition of V into k sets $(V_1, V_2, \ldots, V_k)$, all this colourings being equally likely (so happening with probability $\frac{1}{k^n}$. For every edge e of G, let A_e be the event that the two endvertices of e receive the same colour. Show that:

(a) for any two edges e and f of G, the events A_e and A_f are independent.

(b) if e, f and g are three edges of a triangle of G, the events A_e, A_f and A_g are dependents.

Random Variables: We need the idea of random variables which we recap now. A random variable is another function $X : \Omega \to \mathbb{R}$. The expected value of the random variable is the "weighted average" value that it takes over the real numbers - weighted by the correspoding probability values. That is,

$$E[X] = \sum_{\alpha \in \mathbb{R}} \alpha \Pr[X = \alpha]$$

Indeed, $[X = \alpha]$ represents an event $\{w \in \Omega \mid X(w) = \alpha\} \subseteq \Omega$. Hence, the expectation can also be written equivalently as follows:

$$E[X] = \sum_{\alpha \in \mathbb{R}} \alpha \left(\sum_{\substack{w\in\Omega \\ X(w)=\alpha}} \Pr(w) \right) = \sum_{w\in\Omega} X(w) \Pr(w)$$

We need the following properties of expectation:

Tool 1 : Boolean Random Variables - Suppose X is a random variable that takes only Boolean values. In this case, $E[X] = \Pr[X = 1]$ which follows from the definitions.

Tool 2 : Linearity of Expectation : Suppose X_1 and X_2 are random variables defined based on the same probability distribution, consider the new random variable defined as $X = c_1X_1 + c_2X_2$. This is also a random variable as it is a function from $\Omega \to \mathbb{R}$ defined as $X(w) = c_1X_1(w) + c_2X_2(w)$ for every $w \in \Omega$. It turns out there is a neat relationship between the expectation of the random variables X, X_1 and X_2. This is one of the most imporant relation that is extensively used in analysis of randomized algorithms.

$$\begin{aligned} E[X] &= \sum_{w\in\Omega} X(w)\Pr(w) = \sum_{w\in\Omega} (c_1X_1(w) + c_2X_2(2))\Pr(w) \\ &= c_1\left(\sum_{w\in\Omega} X_1(w)\Pr(w)\right) + c_2\left(\sum_{w\in\Omega} X_2(w)\Pr(w)\right) = c_1E[X_1] + c_2E[X_2] \end{aligned}$$

We suggest practicing the application of linearity of expectation using the following exercise.

Exercise 2.6 (See Problem Set 1(Problem 2))**.** A graph $G = (V, E)$ is created at random by selecting each edge with probability p. What is the expected number of spanning trees in the randomly sampled graph? (*Hint : Use Cayley's Theorem that the number of distinct spanning trees on n vertices is n^{n-2}. Order them, and define an indicator random variable.*)

Tool 3 : Averaging Principle - Suppose X is a random variable and $E[X] = \mu$, then the following statements follow:

$$\exists w \in \Omega : X(w) \geq \mu \qquad \exists w \in \Omega : X(w) \leq \mu$$

Both of them can be proved by contradiction. For sample, we spell out the first one, suppose that the first statement is false. That is, $\forall w \in \Omega, X(w) < \mu$, then:

$$E[X] = \sum_{w\in\Omega} X(w)\Pr(w) < \sum_{w\in\Omega} (\mu \times \Pr(w)) = \mu\left(\sum_{w\in\Omega} \Pr(w)\right) = \mu$$

This implies, $E[X] < \mu$ which is a contrdiction. A similar proof holds for the other claim as well.

Tool 4 : Tail inequalities - Suppose we have a random variable X such that $E[X] = \mu$. What kind of probability guarantees can we write for X? For example, can we bound (in terms of the expectation) the probability that $X > \alpha$ for some $\alpha \in \mathbb{R}$? This is what tail bounds do. They help us write probability upper bounds based on expectations and other related parameters.

As a first example, consider a random variable that takes only non-negative values. Then we can write :

$$\textbf{Markov's Inequality :}\ \Pr[X \geq a] \leq \frac{E[X]}{a}$$

The proof is also quite simple.

$$E[X] = \sum_{\alpha \in \mathbb{R}} \alpha \Pr[X = \alpha] \geq \sum_{\alpha \geq a} \alpha \Pr[X = \alpha] \geq \sum_{\alpha \geq a} a \Pr[X = \alpha] \geq a \Pr[X \geq a]$$

For example, this helps us make statements of the form :

$$\Pr[X > 4\mu] \leq \frac{\mu}{4\mu} = \frac{1}{4}$$

That is the probability that the random variable takes a value which is more than 4 times the expected value is at most 0.25. Unfortunately, this does not help us write down a probability bound for X taking value less than say $\frac{\mu}{4}$. Indeed, Markov's inequality is also pretty weak - as demonstrated by the following example - consider tossing n coins and X be the number of heads. Clearly $E[X] = \frac{n}{2}$. By Markov's inequality, $\Pr[X \geq n] \leq \frac{1}{2}$ but we know that it is much smaller than that, namely $\frac{1}{2^n}$.

What do we want if we want to bound the probability that the random variable takes a much lower value than the expectation? This is where we require more the next tail bound (without any assumptio of positivity on the random variable).

Chebychev's Inequality : $\Pr[|X - \mu| \geq a] \leq \frac{Var[X]}{a^2}$ where, $Var[X] = E[X^2] - E[X]^2$

We will not discuss the proof of Chebychev's inequality since we do not require it immediately. This concludes the review.

2.7 Analysis of the Algorithm for MAXCUT

Recall Algorithm **??**. We want to guarantee that th expected size of the cut is at most half of the optimal cut size. In fact, we prove something stronger.

CLAIM 2.7.1. *Let X be the size of the cut output by the algorithm* **??**. *Then, $E[X] \geq \frac{m}{2}$ where m is the number of edges in the graph G.*

Proof. For each edge $e \in E$ define a random variable X_e as the following indicator variable:

$$X_e = \begin{cases} 1 & \text{if } e \in cut(S, T) \\ 0 & \text{otherwise} \end{cases}$$

By definition, $X = \sum_{e \in E} X_e$. Hence, by linearity of expectation:

$$E[X] = \sum_{e \in E} E[X_e] = mE[X_e]$$

We just need to notice that $E[X_e] = \Pr[X_e = 1] = \Pr[e \in cut(S, T)]$ because of tool 1. Notice that an edge e is in the cut if the two end points get into different sets among S and T. That is, out of

the four possible outcomes of the random coin tosses correspnoding to the endpoint vertices of e, two of them leads to e being in $cut(S,T)$. Hence this is exactly $\frac{1}{2}$. This gives: $E[X] \geq \frac{m}{2}$. Hence the proof. □

Notice that the claim is stronger. Indeed, since the optimum cut can only cut at most m edges, the above also implies $E[X] \geq \frac{m}{2} \geq \frac{\text{OptCut}}{2}$.

2.8 Method of Conditional Expectations

We now describe the main technical idea to be learned this week. Mainly an algorithm specific technique of derandomization of randomized algorithms. This presentation is from Salil Vadhan's book on Pseudorandomness.

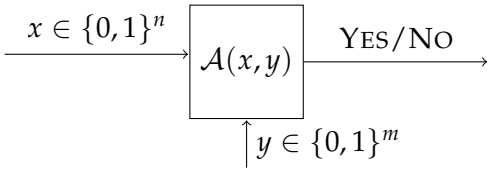
There are two kinds of randomized algorithms that we have seen so far - essentially to solve two kinds of problems. One is for the decision problems where the probability over different paths of the computation tree, the algorithm being correct is at least $\frac{2}{3}$. The other is for optimization problems where the expected size of the output has guarantees. The method of derandomization that we are going to discuss can in principle be applied for both, if the randomized algorithm in question uses the random bits in a peculiar way that certain measures can be computed efficiently about the output for particular settings of the randomness.

Main Idea: (as we described in the lecture) - we discuss the first kind of algorithms first and then adapt it to the second type. In the decision problem case, we know that $\frac{2}{3}$-rd paths in the computation tree are going to make the algorithm answer correctly. A vague idea would be - walk down the path of the tree, *making a choice deterministically and efficiently at each node (without trying both choices which leads to exponential time) maintaining the invariant that within the subtree that have restricted ourselves to, a $\frac{2}{3}$ fraction of paths within the subtree still make the algorithm go correct*. If we make choices like this and set the random bits based on that choices, it is intuitive that we will reach a leaf that makes the algroithm answer correctly and then we can just run the algorithm on that leaf (that choice of random bits) and output the answer. The process is deterministic and effcient and hence gives a derandomization of the original algorithm.

Of course, this is easier said than done. An important question remains - *at an intermediate node, in the above walk-down, how do we deterministically and efficiently decided whether to take the edge labelled* 0 *or* 1 *to move to the child*? Formalizing this will require us to fix the type of problem as decision vs search/optimization problem.

2.8.1 Framework for Algorithms for Decision Problems

Recall the following notational set up where $\mathcal{A}$ is the randomized algorithm solving a decision problem.



$$\forall x \in \{0,1\}^n, \quad \Pr_{y \in \{0,1\}^m}[A(x,y) \text{ is correct.}] \geq \frac{2}{3}$$

Since we have to keep track of the fraction of paths for a particular partial setting of the random bits (while analysing the walk-down of the tree at an intermediate stage) - we define the following notation:

For every $i \in [n]$, bits $r_1, r_2, \ldots r_i \in \{0,1\}$, define:

$$p(r_1, r_2, \ldots, r_i) = \Pr_{y \in \{0,1\}^m} \{A(x,y) \text{ is correct } \mid (y_1 = r_1) \wedge (y_2 = r_2) \wedge \ldots \wedge (y_i = r_i)\}$$

Indeed, if we are at a particular node represented by a partial assignments $r_1, r_2, \ldots r_i$, the value of $p(r_1, r_2, \ldots, r_i)$ is the average of the value at the two children of that node since the bit is chosen uniformly at random. In terms of expectation, this is equivalent to :

$$p(r_1, r_2, \ldots r_i) = E_{y_{i+1} \in \{0,1\}}(r_1, r_2, \ldots r_i, y_{i+1})$$

To understand this definition clearly, let us ask, a first question, what is the value of $p(r_1, r_2, \ldots r_m)$ for a setting $r_1, r_2, \ldots r_m \in \{0,1\}$. (Class answered 0 or 1 depending on whether the setting represents a path which makes $\mathcal{A}$ correct or not). How about $p(\phi)$, which represents the value of the function when no bit is set. Clearly, by definition, this represents the top of the computation tree, and hence the fraction of correct paths under the node is exactly the success probability of the algorithm. That is, $p(\phi) \geq \frac{2}{3}$.

Indeed, if we call $p(r_1, r_2, \ldots r_i) = \mu$, we have that: $E_{y_{i+1} \in \{0,1\}} p(r_1, r_2, \ldots r_i, y_{i+1}) = \mu$. Hence we know that there must exist a setting of y_{i+1} such that the value of the random variable - which in this case is $p(r_1, r_2, \ldots r_i, y_{i+1})$ - is at least the expected value. That is,

$$\exists r_{i+1} \in \{0,1\} : p(r_1, r_2, \ldots r_{i+1}) \geq p(r_1, r_2, \ldots r_i)$$

Applying this repeatedly, we have that $\exists r_1, r_2, \ldots r_m \in \{0,1\}$:

$$p(r_1, r_2, \ldots r_m) \geq p(r_1, r_2, \ldots r_{m-1}) \geq \ldots \geq p(r_1, r_2) \geq p(r_1) \geq p(\phi) \geq \frac{2}{3}$$

Notice that, by our observation, the left-end term is Boolean, and hence it must be that there exists $r_1, r_2, \ldots r_m \in \{0,1\}$ such that $p(r_1, r_2, \ldots r_m) = 1$. But then, this is not a big deal in the end, we knew about existance of such r_i's anyway. So in the end it does not look very useful.

But the above framework has an interesting feature. It also shows how to construct r_i's bit-by-bit, if we have an efficient algorithm to compute $p(r_1, r_2, \ldots r_i)$ for any i. Suppose we have computed $r_1, \ldots r_i$ already, we can compute r_{i+1} as follows: compute $p(r_1, r_2, \ldots, r_i, 0)$ and $p(r_1, r_2, \ldots, r_i, 1)$ using the above algorithm and set r_{i+1} to be whichever bit in $\{0,1\}$ which achieves the maximum.

However, we still have the problem of computing $p(r_1, r_2, \ldots r_i)$ for any $i \in [m]$ efficiently. This is where the algorithm-specifics come in. The algorithm $\mathcal{A}$ should be using the random bits in a peculiar way such that the value of $p(r_1, r_2, \ldots, r_i)$ can be computed efficiently for that algorithm $\mathcal{A}$. Indeed, the trivial method of computing $p(r_1, r_2, \ldots r_i)$ ends up taking exponential time in the worst case. Hence one has to use the algorithm-specific attributes to design this algorithm. We will demonstrate this in the next context.

2.8.2 Framework for Algorithms for Optimization Problems

We now the adapt the above framework for search and optimization problems. The guarantee for algorithms solving such problems is as follows - the expected size of the output is at least as "good" as this, where "good" means at most or at least in minimization and maximization problems respectively. For demonstrative purposes, we restrict ourselves to the randomized algorithm for MAXCUT that we presented earlier. Notice that the algorithm uses exactly n random bits where $|V| = n$. Note that S and T are the two subsets output by the algorithm. For any $i \in [n]$, define:

$$V(r_1, r_2, \ldots r_i) = \mathbb{E}_{y \in \{0,1\}^n} \left[|cut(S,T)| : (y_1 = r_1) \wedge (y_2 = r_2) \wedge \ldots \wedge (y_i = r_i) \right]$$

Similar to the previous setting, note that $V(\phi) \geq \frac{|E|}{2}$ since the expected size of the cut when no random bit is conditioned is similar to the original analysis of the algorithm. We apply the averaging principle and argue in a similar way that there must exist a choice of the random bits $r_1, r_2, \ldots, r_n \in \{0,1\}$ such that $cut(S,T)$ output by the algorithm has at least $\frac{|E|}{2}$ many edges.

Indeed, we start with $V(\phi) \geq \frac{|E|}{2}$. By averaging principle, there must exist $r_1 \in \{0,1\}$ such that $V(r_1) \geq V(\phi) \geq \frac{|E|}{2}$. Continuing in a similar way there must exist $r_1, r_2, \ldots, r_n \in \{0,1\}$ such that :

$$V(r_1, r_2, \ldots r_{n-2}, r_{n-1}, r_n) \geq V(r_1, r_2, \ldots r_{n-2}, r_{n-1}) \geq V(r_1, r_2, \ldots r_{n-2}) \ldots \geq V(r_1) \geq V(\phi) \geq \frac{|E|}{2}$$

Again, this is not new information. We can always derive by a globally applying averaging principle that there must exist such a choice of random bits. But the advantage here is that if there is an efficient algorithm for computing $V(r_1, r_2, \ldots r_i)$ for any i, then we can find out the explicit choice of values of the bits as well. As in the previous case, if $r_1, r_2, \ldots r_i$ is already fixed, then we compute r_{i+1} as : compute $V(r_1, r_2, \ldots r_i, 0)$ and $V(r_1, r_2, \ldots r_i, 1)$ and set r_{i+1} to be that value in $\{0,1\}$ which results in the maximum among the two.

Computing $V(r_1, r_2, \ldots r_i)$ for the algorithm for MAXCUT : To apply the above framework, all we need is an efficient algorithm to compute the value of $V(r_1, r_2, \ldots r_i)$ for any choice of i and $r_1, r_2, \ldots r_i \in \{0,1\}$. Note that this is a speciality of the algorithm - more importantly the way the bits $y_1, y_2, \ldots y_n$ are used by the algorithm.

At any intermediate point of computation, there are vertices for which the decision (of whether they should be in the set S or not) is already made by then and there are vertices which are decided later. To keep track of this, we define:

$$\begin{aligned} S_i &= \{j \in [n] \mid j \leq i, b_j = 1\} \\ T_i &= \{j \in [n] \mid j \leq i, b_j = 0\} \\ U_i &= \{j \in [n] \mid j > i\} \end{aligned}$$

The algorithm will grow S_i to S and T_i to T, by randomly choosing the remaining vertices (U_i) to be in S or T. We need to compute the expected size of the cut conditioned on the fact that the sets S_i and T_i are already fixed by the algorithm. An immediate observation is that the edges that go across S_i and T_i will necessarily a part of the cut, since their endpoints are already at S and T

respectively by definition. The edges which are fully within S_i or fully within T_i are not going to be a part of the cut finally. But there may be more number of edges which forms a part of the final cut. Considering this, we can write:

$$V(r_1, r_2, \ldots r_i) = |cut(S_i, T_i)| + \frac{|(cut(S_i, U_i)| + |cut(U_i, T_i)| + |cut(U_i, U_i)|}{2}$$

We need to explain the second term in the RHS. Consider edges $e = (u, v) \in E$ that has one endpoint $u \in S_i$ and the other endpoint in $v \in U_i$. Note that $u \in S$ finally, and hence (u, v) edge will be counted in the cut, if v falls into T. Since this is decided by choosing a random bit, the probability that the edge appears in the cut finally is $\frac{1}{2}$. Hence, the expected number of edges in $cut(S_i, U_i)$ which appear in the final cut is $\frac{|cut(S_i,U_i)|}{2}$. Similar argument explains the term $\frac{|cut(T_i,U_i)|}{2}$. To see the $\frac{|cut(U_i,U_i)|}{2}$ term, consider edges which are having both end points in U_i. They are both going to be put in S or T uniformly at random - hence out of the four possible outcomes (for these two vertices), two of them puts them in the final cut and two of them puts them outside the final cut output by the algorithm. Hence for any edge in $cut(U_i, U_i)$, with probability $\frac{1}{2}$ it will form a part of the cut. That is, expected number of edges that gets contributed to the final cut is $\frac{|cut(U_i,U_i)|}{2}$. Hence the expression for $V(r_1, r_2, \ldots r_i)$ is correct.

Exercise 2.9 (See Problem Set 1(Problem 3))**.** In the derandomization of MAXCUT algorithm that we described, we derived an expression for $V(r_1, r_2, \ldots r_i)$ for any i and $r_1, r_2, \ldots r_i \in \{0, 1\}$. We used this to determine, the value of r_{i+1} by computing $V(r_1, r_2, \ldots r_i, 0)$ and $V(r_1, r_2, \ldots r_i, 0)$ and then choosing the largest. Prove that the choice of r_{i+1} will be 1 if vertex $i + 1$ has more neighbors in T_i than in S_i and vice versa. Hence, write down the derandomized 0.5-approximation deterministic polynomial time algorithm for MAXCUT as a simple greedy algorithm in terms of the above rule.

Exercise 2.10 (See Problem Set 1(Problem 4))**.** Let $x_1, x_2, \ldots x_n \in \{0, 1\}$ be Boolean variables and let f be a Boolean formula in CNF form. That is, $f = C_1 \wedge C_2 \wedge \ldots \wedge C_m$ where each C_i (called a *clause*) is a disjunction of literals in the set $\{x_1, \overline{x_1}, x_2, \overline{x_2}, \ldots, x_n, \overline{x_n}\}$. We want to find an assignment of the Boolean variables that satisfies as many clauses in the formula as possible.

(a) Write down a randomized algorithm that outputs an assignment with the guarantee that the expected number of clauses satisfied is at least $\frac{m}{2}$.

(b) Derandomize this algorithm using the method of conditional probabilities discussed in class to get a deterministic algorithm that satisfies at least $\frac{m}{2}$ number of clauses.

(c) Suppose k is the minimum number of literals in any clause, how will you modify the parameters in part(a) and (b)

Part II

Exercise & Problem Sets

Chapter 3

Exercises

3.1 Exercises

Exercise 3.2. Let $G \in G(n,p)$. For all $S \subseteq V$, let A_S be the event that S forms an independent set in G. Show that if S and T are two distinct subsets of k vertices then A_S and A_T are independent if and only if $|S \cap T| \leq 1$.

Exercise 3.3. Prove that an event $\mathcal{E}$ is independent of a set of events $\{\mathcal{E}_j \mid j \in J\}$ if and only if for all $J_1, J_2 \subseteq J$ such that $J_1 \cap J_2 = \phi$

$$\Pr[\mathcal{E} \cap \left(\cap_{j \in J_1} B_j\right) \cap \left(\cap_{j \in J_2} \overline{B_j}\right)] = \Pr(\mathcal{E}) \Pr[\left(\cap_{j \in J_1} B_j\right) \cap \left(\cap_{j \in J_2} \overline{B_j}\right)]$$

3.4 Curiosity Drive

Here we list down all the ”out of curious” questions that we discussed (sometimes even not discussed) in the class (and hence in this document).

Chapter 4

Problem Sets

4.1 Problem Set #1

(1) (See Exercise 2.5) A random k-colouring fo a graph G is an element of the probability space (Ω, Pr) where Ω is the set of all k-colourings (i.e. partition of V into k sets $(V_1, V_2, \ldots, V_k)$, all this colourings being equally likely (so happening with probability $\frac{1}{k^n}$. For every edge e of G, let A_e be the event that the two endvertices of e receive the same colour. Show that:

(a) for any two edges e and f of G, the events A_e and A_f are independent.

(b) if e, f and g are three edges of a triangle of G, the events A_e, A_f and A_g are dependents.

(2) (See Exercise 2.6) A graph $G = (V, E)$ is created at random by selecting each edge with probability p. What is the expected number of spanning trees in the randomly sampled graph? (*Hint : Use Cayley's Theorem that the number of distinct spanning trees on n vertices is n^{n-2}. Order them, and define an indicator random variable.*)

(3) (See Exercise 2.9) In the derandomization of MAXCUT algorithm that we described, we derived an expression for $V(r_1, r_2, \ldots r_i)$ for any i and $r_1, r_2, \ldots r_i \in \{0, 1\}$. We used this to determine, the value of r_{i+1} by computing $V(r_1, r_2, \ldots r_i, 0)$ and $V(r_1, r_2, \ldots r_i, 0)$ and then choosing the largest. Prove that the choice of r_{i+1} will be 1 if vertex $i + 1$ has more neighbors in T_i than in S_i and vice versa. Hence, write down the derandomized 0.5-approximation deterministic polynomial time algorithm for MAXCUT as a simple greedy algorithm in terms of the above rule.

(4) (See Exercise 2.10) Let $x_1, x_2, \ldots x_n \in \{0, 1\}$ be Boolean variables and let f be a Boolean formula in CNF form. That is, $f = C_1 \wedge C_2 \wedge \ldots \wedge C_m$ where each C_i (called a *clause*) is a disjunction of literals in the set $\{x_1, \overline{x_1}, x_2, \overline{x_2}, \ldots, x_n, \overline{x_n}\}$. We want to find an assignment of the Boolean variables that satisfies as many clauses in the formula as possible.

(a) Write down a randomized algorithm that outputs an assignment with the guarantee that the expected number of clauses satisfied is at least $\frac{m}{2}$.

(b) Derandomize this algorithm using the method of conditional probabilities discussed in class to get a deterministic algorithm that satisfies at least $\frac{m}{2}$ number of clauses.

(c) Suppose k is the minimum number of literals in any clause, how will you modify the parameters in part(a) and (b)