6.842 Randomness and Computation February 12, 2014

Lecture 3

Lecturer: Ronitt Rubinfeld *Scribe: Sheela Devadas*

1 Randomized Complexity Classes

Definition 1 *A* language L *is a subset of* $\{0,1\}^*$.

The strings in a language represent 'yes' instances of a language. Examples include graphs with a Hamiltonian cycle , or descriptions of sets with a proper 2-coloring. A language poses a problem in the sense that determining if a string is in the language is a decision problem.

Definition 2 P *is a class of languages L with polynomial time deterministic algorithms A such that* $x \in L \to A(x)$ *accepts and* $x \notin L \to A(x)$ *rejects.*

The algorithm A decides whether x is in the language L or not - it solves the problem posed by the language.

Definition 3 RP *is a class of languages L with polynomial time probabilistic algorithms A such that* $x \in L \to A(x)$ *accepts with probability at least* $1/2$ *and* $x \notin L \to A(x)$ *rejects with probability* 1.

The algorithm A in this case has *one-sided error*, since there are no false positives.

Definition 4 BPP *is a class of languages L with polynomial time probabilistic algorithms A such that* $x \in L \to A(x)$ *accepts with probability at least* $2/3$ *and if* $x \notin L$ *the probability that* $A(x)$ *accepts is* $\leq 1/3$.

In this case we have *two-sided error*, since we have both false positives and false negatives as possibilities. $2/3$ and $1/3$ are arbitrary; they just need to be bounded away from each other and can be replaced with $c_1, c_2 \in [0,1]$ with $c_1 > c_2$. These constants are arbitrary because we can just repeat the algorithm A a sufficient number of times and output 'accept' if A ever outputs 'accept'. If we want the probability of error to be β, we need to repeat A $\log \frac{1}{\beta}$ times. By Chernoff bounds, if we repeat $\Theta(\log \frac{1}{\beta})$ times, the probability that the majority answer is correct is $\geq 1 - \beta$. (Effectively Chernoff bounds show that the probability of error goes down exponentially with the number of times you repeat.)

It's clear that P $\subseteq$ RP $\subseteq$ BPP. One open question is if P is equal to BPP, which is one of the themes of the course. Primality testing was a candidate for showing that P and BPP were unequal; however a deterministic algorithm was found ([1]).

2 Derandomization Via Enumeration

The question is whether algorithms really require tossing coins. There are some distributed settings in which it's impossible to solve problems without tossing coins, but here we're just considering the case of algorithms.

Coins can be simulated by trying all possible coin tosses - this is *enumeration* - but this results in a loss of efficiency in many cases.

Given a probabilistic algorithm A and input x, we run A on every possible random string of length $r(n)$ - $r(n)$ is the number of random bits used by A on inputs of size n. We note that $r(n) \leq t_A(n)$ where $t_A(n)$ is a runtime bound on A (since it takes at least one step to access a random bit). We then output the majority answer. Our runtime is $O(2^{r(n)} t_A(n))$, since we run the algorithm A $2^{r(n)}$ times.

If $x \in L$, then at least $2/3$ of the random strings will cause A to accept, so in that case our algorithm gives the correct answer. If x is not in the language L, then at least $2/3$ of the random strings will cause A to reject, so our algorithm will also reject, which is the correct answer.

Therefore we see that coin tosses do not allow us to solve problems that cannot be solved deterministically. We also note that BPP $\subseteq$ EXP, where EXP is DTIME($\bigcup_C 2^{n^C}$).

We note that if we have an algorithm only using logarithmically many random bits, that our derandomized algorithm actually runs in polynomial time.

3 Pairwise Independence and Randomness

3.1 Randomized Algorithm for Max Cut

We consider a randomized algorithm for max cut. Given a graph $G = (V, E)$, the goal is to output a partition of the vertices into S, T to maximize the number of edges crossing the cut - the number of edges (u, v) such that $u \in S, v \in T$, which is the size of the S, T-cut. When the graph is bipartite, it is possible for the size of the cut to be equal to the number of edges in the graph. This is an NP-hard problem in general.

The randomized algorithm works as follows. We flip n coins, $r_1, \ldots, r_n$, and we put vertex i in S if $r_i = 0$ and in T if $r_i = 1$.

Theorem 5 *The expected size of the random cut produced by this algorithm is* $|E|/2$.

Proof Let $1_{u,v} = 1$ if (u, v) crosses the cut - meaning that $r_u \neq r_v$ - and 0 otherwise (if $r_u = r_v$). The size of our cut is $\sum_{(u,v)\in E} 1_{u,v}$. Then the expected size of our cut is just $\mathbb{E}[\sum_{(u,v)\in E} 1_{u,v}] = \sum_{(u,v)\in E} \mathbb{E}[1_{u,v}]$ by linearity of expectation; we then use the fact that the expected value of an indicator variable is the probability that it is 1, so the expected size of our cut is $\sum_{(u,v)\in E} Pr[1_{u,v} = 1] = \sum_{(u,v)\in E} Pr[r_u \neq r_v] = \sum_{(u,v)\in E} Pr[r_u = 1, r_v = 0 \text{ or } r_u = 0, r_v = 1]$. Since the two events $r_u = 1, r_v = 0$ and $r_u = 0, r_v = 1$ are disjoint, the probability that one or the other happens is the sum of the probabilities, which is $1/4 + 1/4 = 1/2$. The probability of each event we calcuate based on the fact that r_u, r_v are independent. Then the expected size of our cut is $\sum_{(u,v)\in E} 1/2 = |E|/2$. ■

3.2 Using Pairwise Independence

Recall we relied on the coin tosses for r_u, r_v being independent. They are pairwise independent; however, we did not need a guarantee of independence beyond pairwise independence.

If we could generate pairwise independent random variables using less randomness than it takes to generate completely independent random variables, then our derandomization (using the method described in the previous section) could be more efficient.

We pick n values $x_1, \ldots, x_N$ with $x_i \in T$ with $|T| = t$. For any $b \in T$, the probability that $x_i = b$ is $1/t$ for all i - we are assuming uniform distribution.

Definition 6 *We call the* x_i "independent" *if for any string* $b_1, \ldots, b_n \in T^n$, *the probability that* $x_1, \ldots, x_n = b_1, \ldots, b_n$ *is* $1/t^n$. *We call the* x_i "pairwise independent" *if for any* $i \neq j$ *and* $b_i, b_j \in T^2$, *the probability that* $x_i, x_j = b_i, b_j$ *is* $1/t^2$. *We call the* x_i "k-wise independent" *if for any* $i_1, \ldots, i_k$ *and* $b_{i_1}, \ldots, b_{i_k} \in T^2$, *the probability that* $x_{i_1}, \ldots, x_{i_k} = b_{i_1}, \ldots, b_{i_k}$ *is* $1/t^k$.

We note that in the previous proof, we only used pairwise independence rather than general independence. In a full enumeration for the max cut algorithm, we would have to try all possible coin tosses (all possible cuts) and then pick the best one. While this gets you the very best cut, it's not efficient.

Instead we consider a partial enumeration: we pick a subset of coin tosses that is not as large as the full set but still satisfies the pairwise independence property. We then enumerate on this subset.

Consider the following sets of three coin tosses:

r_1	r_2	r_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

It's clear that our random bits here are completely independent since this is all the possible sets of three coin tosses. However, if we consider the following sets of three coin tosses instead, in which $r_1 \oplus r_2 = r_3$:

r_1	r_2	r_3
0	0	0
0	1	1
1	0	1
1	1	0

we do in fact have pairwise independence but obviously not general independence since given two bits we can always determine the third.

What we want is to take a few truly random or totally independent bits - $b_1, \dots, b_m$ and put them into a randomness generator to generate pairwise independent bits $r_1, \dots, r_n$. The question is then how much bigger than m can we make n? (Previously we just had the case $m = 2, n = 3$).

We are going to enumerate over all of the b_i, which we call the 'seeds', and use them to pick a random row from the 'randomness generator', and generate $r_1, \dots, r_n$ for the algorithm. If we derandomize in this way, our running time is then $O(2^m T_A(n))$, which could be in polynomial time if m is logarithmic.

We assume we have a 'randomness generator' taking $O(\log n)$ truly random bits and producing n random bits.

First, we make a new randomized max cut algorithm, MC'. It uses fewer random bits and runs as follows: we pick $\log n + 1$ truly random bits $b_1, \dots, b_{\log n+1}$ and use the generator to construct n pairwise independent random bits $r_1, \dots, r_n$. We then use the r_i in the max cut algorithm described earlier and evaluate the size of the cut.

We now describe a deterministic, derandomized max cut algorithm. For all choices of $b_1, \dots, b_{\log n+1}$, we run MC' and evaluate the cut size, and output the best size found.

This is not necessarily the best cut possible, but we wish to show that we get a fairly good approximation. The runtime is $2^{\log n+1} \times t_{MC'}(n)$, where $t_{MC'}(n)$ includes the time it takes to run the generator and the time to run the actual max cut algorithm. Our previous analysis shows that at least one of our random cuts must be fairly good ($\geq |E|/2$), so the final output of our algorithm must be fairly good too.

3.3 Generating Pairwise Independent Bits

We consider a construction of the randomness generator. We choose k truly random bits $b_1, \dots, b_k$, and for every subset $S \subseteq [k]$ such that $S \neq \emptyset$, we let $c_s = \bigoplus_{i \in S} b_i$, and we output all the c_s. Proving that these are all pairwise independent is a homework exercise.

We had k bits that are truly random, and we output $2^k - 1$ pairwise independent random bits.

We consider another construction of the randomness generator. We wish to generate random integers in $[0, \dots, q-1]$, where q is prime.

A trivial method that works when $q = 2^l$ (here q is not prime) is to just repeat the previous construction for each position in $1 \dots l$: that is, if we wish to produce n pairwise independent random integers, we construct l sets of n pairwise independent bits; the first bit of each of the integers is from the first set of n bits generated, the second bit of each integer from the second set of bits, and so on. This construction uses $O(\log n \log q) = O(l \log n)$ truly random bits.

A better construction, for q prime, uses $O(\log q)$ bits when $n \approx q$. Specifically it uses $2 \log q$ random bits to yield q pairwise independent elements of the field $\mathbb{Z}_q$. We pick $a, b \in \mathbb{Z}_q$. We let $r_i = ai + b \mod q$ for all i in $0, \ldots, q-1$ and we output $r_1, \ldots, r_q$. It's useful to think of this as a function $h_{a,b}$ from $[0, \ldots, q-1]$ to $\mathbb{Z}_q$. We now have a family of functions $H = \{h_1, h_2 \ldots\}$ for $h_i : [N] \to [M]$ that is *pairwise independent* in the sense that for all x in the domain, $H(x)$ is uniformly distributed in $[M]$ ($H(x) \in_u [M]$) and for any $x_1 \neq x_2 \in [N]$, $H(x_1)$ and $H(x_2)$ are independent.

Our family is $H = \{h_{a,b} : \mathbb{Z}_q \to \mathbb{Z}_q\}$. For any $x \neq w$, the probability that $h_{a,b}(x) = c$ and $h_{a,b}(w) = d$ is $1/q^2$, since only one choice of a, b is a solution to $ax + b = c$ and $aw + b = d$. Effectively we are trying to solve the matrix equation $\begin{pmatrix} x & 1 \\ w & 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} c \\ d \end{pmatrix}$, which has a unique solution for $\begin{pmatrix} a \\ b \end{pmatrix}$ because $\begin{pmatrix} x & 1 \\ w & 1 \end{pmatrix}$ is invertible; its determinant is $x - w \neq 0$.

To generalize to k-wise independence, we can use polynomial equations instead of linear ones - for example, for 3-wise independence, we could use functions of the form $h_{a,b,c}(x) = ax^2 + bx + c \mod q$. We can use a similar argument with a matrix equation to prove that these functions are 3-wise independent, though it is more difficult to prove that the necessary matrix (for example, $\begin{pmatrix} x^2 & x & 1 \\ w^2 & w & 1 \\ z^2 & z & 1 \end{pmatrix}$) is invertible in this case.

References

[1] Manindra Agrawal, Neeraj Kayal, Nitin Saxena, PRIMES is in P. In *Annals of Mathematics*, 160(2), pages 781-793, 2004.