6.895 Randomness and Computation March 31, 2008

Lecture 14

Lecturer: Ronitt Rubinfeld *Scribe: Tahira Naseem*

1 Last Time

Last time we defined the following complexity classes.

Definition 1 RP*(randomized polynomial time) is the class of languages L for which there exists a probabilistic polynomial-time algorithm $\mathcal{A}$ such that*

$$\begin{aligned} x \in L &\Rightarrow Pr[\mathcal{A}(x) \text{ accepts}] \geq \frac{1}{2} \\ x \notin L &\Rightarrow Pr[\mathcal{A}(x) \text{ accepts}] = 0 \end{aligned}$$

Definition 2 BPP*(bounded-error probabilistic polynomial time) is the class of languages L for which there exists a probabilistic polynomial-time algorithm $\mathcal{A}$ such that*

$$\begin{aligned} x \in L &\Rightarrow Pr[\mathcal{A}(x) \text{ accepts}] \geq \frac{2}{3} \\ x \notin L &\Rightarrow Pr[\mathcal{A}(x) \text{ accepts}] \leq \frac{1}{3} \end{aligned}$$

2 This Time: Derandomization

In this lecture we discuss how to derandomize algorithms. We will see a brute force algorithm (enumeration) for derandomization. We will also see that some random algorithms do not need true randomness. Specifically, we will see an example where only pairwise random bits are needed. Next, we will see how we can generate pairwise random values and how this conservation on the amount of randomness can reduce the time needed for derandomization. In the end we discuss a two-point sampling algorithm that also reduces the number of random bits needed but it worsens the running time.

3 Derandomization via enumeration

Idea:
In general, a random algorithm picks values from a domain randomly and then based on those random samples, approximates a solution. The idea of this algorithm is that rather than picking random samples from space, explore the whole space.

Algorithm

```
Given a BPPalgorithm $\mathcal{A}$, create a deterministic algorithm $\mathcal{B}$ as follows:
    try all random strings on $\mathcal{A}$
    count how many accept
    return the answer corresponding to the fraction of random strings on which $\mathcal{A}$ accepts
      (the exact interpretation of this count depends
      on the algorithm but it will give a deterministic answer)
```

Analysis

Runtime of $\mathcal{B}$:

$r(n)$ = number of random bits used by $\mathcal{A}$
$T_{\mathcal{A}}(n)$ = upper bound on runtime of $\mathcal{A}$
$T_{\mathcal{B}}(n) \leq T_{\mathcal{A}}(n)2^{r(n)}$

Corollary 3 BPP $\leq$ EXPTIME $= \bigcup_c$ DTIME(2^{n^c})

This means randomness does not help in terms of computation ability.

In some randomized algorithms, we do not need truly random bits. Suppose an algorithm $\mathcal{A}$ needs m random bits then the search space for derandomizing the algorithm would be of size 2^m. Now if we know that the algorithm needs these bits to be only weakly random (we will soon discuss what weekly random may mean) and we know some mechanism of generating these weakly random bits using l truly random bits, where $l \ll m$, then this will greatly reduce the search space while derandomizing. Figure 1 shows a graphical form of this concept.

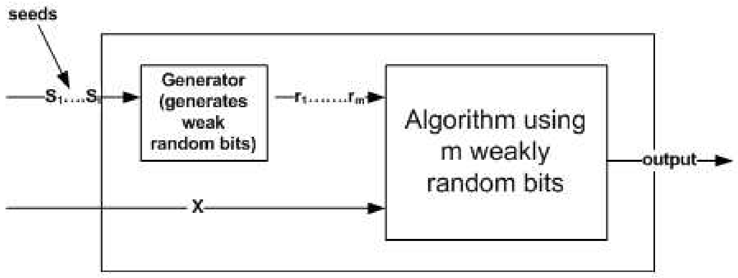
seeds
$S_1 \ldots S_l$
Generator (generates weak random bits)
$r_1 \ldots r_m$
Algorithm using m weakly random bits
output
X

Figure 1: conserving the amount of randomness

Now we will see an example of an algorithm that does not need perfect randomness.

4 MAXCUT: A randomized algorithm that does not need true randomization

Given: A graph $G(V, E)$ where $|V| = n$

Output: Partition V into $H + T$ such that, the size of CUT $= \{(u, v) | u \in H, v \in T\}$ is maximized.

Algorithm:

```
flip n coins $r_1, r_2 ..... r_n$
for each $r_i$
    if $r_i$ is Head
        put vertex $i$ in $H$
    else
        put vertex $i$ in $T$
```

Analysis:

$$
\begin{aligned}
\mathbb{E}[\text{CUT size}] &= \mathbb{E}\left[\sum_{(u,v)} 1_{(u,v)}\right] \\
&\text{(where } 1_{(u,v)} = 1 \text{ if edge } (u,v) \text{ is cut by the CUT and 0 otherwise)} \\
&= \sum_{(u,v)} \Pr[(u,v)\text{crosses the CUT}] \\
&= \sum_{(u,v)} \Pr[(u \in H \text{ and } v \in T) \text{ or } (v \in H \text{ and } u \in T)] \\
&= \sum_{(u,v)} (\Pr(u \in H \cap v \in T) + \Pr(v \in H \cap u \in T)) \\
&= \sum_{(u,v)} (\Pr(u \in H) \cdot \Pr(v \in T) + \Pr(v \in H) \cdot \Pr(u \in T)) \\
&(u \text{ and } v \text{ are assigned independently to their corresponding sets with probability } \tfrac{1}{2}) \\
&= \sum_{(u,v)} \frac{1}{2} = \frac{|E|}{2}
\end{aligned}
$$

The only independence assumption needed to make this analysis work is that in every pair of nodes, both nodes are assigned independently uniformly to any side of the cut. Thus, this algorithm needs only pairwise independence.

5 Pairwise Independent Random Variables

Pick n values $X_1, X_2, \ldots, X_n$ where each $X_i \in T$ such that $|T| = t$

Definition 4 *$X_1, \ldots, X_n$ are* independent *if for all $b_1, \ldots, b_n \in T$, $Pr[X_1 \ldots X_n = b_1 \ldots b_n] = \frac{1}{t^n}$.*

Definition 5 *$X_1, \ldots, X_n$ are* pairwise independent *if for all $i \neq j$, and for all $b_1, b_2 \in T$, $Pr[X_i, X_j = b_1, b_2] = \frac{1}{t^2}$.*

We will use the short form p.i. for "pairwise independent" from this point on wards.

Definition 6 *$X_1, \ldots, X_n$ are k-wise* independent *if for all distinct $i_1, \ldots, i_k$, and all $b_1, \ldots, b_k \in T$ $Pr[X_{i_1}, \ldots, X_{i_k} = b_1, \ldots, b_k] = \frac{1}{t^k}$.*

Informally, a set of n values is k-wise independent if any size-k subset has a uniform distribution over all k-size sets. To achieve the uniform distribution over subsets of variables we do not need the uniform distribution over whole vectors of variables. Consider the following example of three bit vectors. In the second column, if we consider any two bit position in all strings, we will get a random distribution over two bits.

independent	pairwise independent
000	000
001	011
010	101
011	110
100	
101	
110	
111	

If the sample space is small we need lesser number of bits to generate a sample. To pick a number randomly from among 4 numbers we need two random bits, to pick an number randomly from among 8 numbers we need three bits.

Next, we will see how many truly random bits are needed to get n p.i. bits.

5.1 Generating pairwise independent bits

Algorithm

```
Choose $k$ truly random bits $b_1 .... b_k$
$\forall S \subseteq [k]$ such that $S \neq \phi$
    set $C_s = \bigoplus_{i \in S} b_i$
output all $C_s$
```

Claim: for $S \neq T$, C_S and C_T are p.i.

In the above algorithm, k truly random bits give $2^k - 1$ p.i. random bits. This means we only need $\log n$ bits to get n pairwise independent bits. Hence, in the MAXCUT algorithm, we can simulate n coin flips using only $\log n$ random bits and generating n p.i. random bits. To derandomize the algorithm, we need to search a space of size $2^{\log n} = n$.

5.2 Generating pairwise independent numbers

Now we will see an algorithm to generate p.i. numbers in the range $0, \ldots, q-1$, where q is a prime number.

Algorithm

```
Pick $a, b \in \mathbb{Z}_q$ randomly
$\forall_i \in [0....q-1]$
    $r_i \leftarrow ai + b \mod q$
output $r_i$'s
```

This algorithm needs two randomly chosen numbers from among q numbers. Which requires $2 \log q$ random bits. Before further analyzing this algorithm we will define the concept of pairwise independent family of functions.

Definition 7 $\mathcal{H} = \{h_i : [N] \to [M]\}$ *is* pairwise independent family of functions *if* $\forall x \neq y \in [N]$, $\forall a, b \in [M]$, $Pr_{h \in \mathcal{H}}[h(x) = a \text{ and } h(y) = b] = \frac{1}{M^2}$.

Now let us see why the above algorithm gives p.i. values. We can represent the mapping of any pair of numbers, using matrices:

$$\begin{bmatrix} x & 1 \\ y & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} w \\ z \end{bmatrix}.$$

If $x \neq y$ then $\det \begin{bmatrix} x & 1 \\ y & 1 \end{bmatrix} \neq 0$ and for any values of $x \neq y, w, z \in \mathbb{Z}_q$ there is only one pair (a, b) that can map (x, y) to (w, z). (We have seen this trick before for proving that every set has a large subset that is sum free.) Since there are q^2 pairs (a, b) and we choose randomly from them $\Pr_{a,b}[h_{a,b}(x) = w \bigwedge h_{a,b}(y) = z] = \frac{1}{q^2}$, which implies that the set $\mathcal{H} = \{h_{a,b} | \mathbb{Z}_q \to \mathbb{Z}_q\}$, where $h_{a,b} = ai + b \mod q$, is a p.i. family of function.

To fully derandomize an algorithm that uses q p.i. numbers, we only have to go through $2^{2 \log q} = q^2$ possibilities. Note that a p.i. family of function can be useful in this scenario only if a random choice $h_{a,b} \in \mathcal{H}$ is computable in time poly$(\log N, \log M)$ where N and M are the sizes of the domain and range.

6 Two Point Sampling

Given an algorithm $\mathcal{A}$ such that:

$$\begin{aligned} x \in L \quad &\Rightarrow \quad \Pr_R[\mathcal{A}(x,R)=0] < \frac{1}{2}, \\ x \notin L \quad &\Rightarrow \quad \Pr_R[\mathcal{A}(x,R)=0] = 1, \end{aligned}$$

an R such that $\mathcal{A}(x,R)=1$ is a *witness* if $x \in L$.

To reduce the error of this algorithm:

```
    repeat k times,
    output 1 if any $\mathcal{A}(x,R)=1$,
    output 0, otherwise.
```

Let us call this new algorithm $\mathcal{A}'$. It is such that
if $x \in L$, $\Pr[\mathcal{A}'(x,R')=0] \leq \frac{1}{2^k}$,
if $x \notin L$, $\Pr[\mathcal{A}'(x,R')=0] = 1$.

The number of random bits needed by $\mathcal{A}'$ is $O(|R| \cdot k)$. This is because we need to generate $|R|$ bits k times. If we generate only two values of size $|R|$, and generate p.i. values from these two for use in other iterations, we will get a "two point sampling algorithm".

Two point sampling algorithm

```
Pick $a, b$ randomly from $\mathbb{Z}_{2^{|R|}}$
Construct $r_1, \ldots, r_k$, where
    $r_i = a \cdot c_i + b$, where each $c_i$ is a different fixed element of $\mathbb{Z}_{2^{|R|}}$
Compute $\mathcal{A}(x,r_1), \mathcal{A}(x,r_2), \ldots, \mathcal{A}(x,r_k)$.
If there is an $i$ such that $\mathcal{A}(x,r_i)=1$
    output 1
else
    output 0
```

Analysis

If $x \notin L$, the above algorithm never misclassifies.
If $x \in L$,
then it misclassifies if it never sees a witness i.e. $\mathcal{A}(x,r_i)=0$, for all i.
Let $Y = \sum_{i=0}^{k} \mathcal{A}(x,r_i)$, $\mathbb{E}[Y] \geq \frac{k}{2}$.
Pr[never sees a witness] $= \Pr[Y=0] \leq \frac{c}{k} = O(\frac{1}{k})$.

The last step can be obtained by using the Chebyshev inequality, and applying it to pairwise independent r.v.'s. We have $\Pr[|\mathbf{X}-\mu| \geq \epsilon] \leq \frac{1}{k\epsilon^2}$, where X_i is the indicator if r_i is a witness, $\mathbf{X} = \frac{1}{k}\Sigma X_i$, and $\mu = E[\mathbf{X}]$. For $X \in L$, we have $\mu = \mathbb{E}[\frac{Y}{k}] = \mathbb{E}[\mathbf{X}] \geq \frac{1}{2}$. Hence, $\Pr[Y=0] \leq \Pr[|\frac{Y}{k}-\mu| \geq \mu] \leq \frac{1}{k\mu^2} = O(\frac{1}{k})$. This holds because $|\frac{Y}{k}-\mu| \geq \mu$ includes the case where $Y=0$.

Thus, we can get $\frac{1}{k}$ error bound using $O(\log k)$ random bits. If we want our original $\frac{1}{2^k}$ error bound, we will need $\log 2^k = k$ random bits and 2^k iteration. So we will still be saving on random bits, but the running time will be worse.