| Com S 633: Randomness in Computation | |
|---|---|
| Lecture 14 | Scribe: Ankit Agrawal |

# 1 Derandomization

We have seen various applications of randomization, wherein we were able to design simple and efficient algorithms for some (supposedly) deterministically hard problems. Now, we shall see whether randomness can be *eliminated* or at least *reduced* from these algorithms; and if so, how?

Broadly, there are two ways of achieving derandomization:
i) Algorithmic derandomization
ii) Complexity theoretic derandomization

*Algorithmic derandomization* techniques look at a particular randomized algorithm, and using the inherent properties of the problem, 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.

*Complexity theoretic derandomization* techniques refer to a general strategy that can be used to remove randomness from a broad class of algorithms. Typically, this is done by pseudo-random generators, which produce random-looking bits.

## 1.1 Derandomizing Maxcut

*Max-Cut problem*
Given a graph $G = (V, E)$,$|V| = n$, $|E| = m$, output two sets $A$ and $B$ s.t. $A \cup B = V, A \cap B = \Phi$, maximizing the number of edges from $A$ to $B$.
We have earlier seen a $\frac{1}{2}$-approximate randomized algorithm for this problem:

Max-Cut
1 $A = \phi$ , $B = \phi$.
2 **for** $i$=1 to $n$
  2.1 toss a coin
  2.2 **if** $H$, $A = A \cup \{v_i\}$
  2.3 **if** $T$, $B = B \cup \{v_i\}$

The resulting cut is given by $(A, B)$.

$$E\,[\text{ cut size}] \geq \frac{m}{2}$$

Now, we want to remove randomness from this algorithm. We will use the idea of *conditional expectance* . Suppose $v_1, v_2, ..., v_i$ are already placed into $A$ or $B$ (deterministically). Let us analyze the expectation of cut-size if we were to allow randomization from now on.

$E\,[\text{cut size} \mid v_1, v_2, ..., v_i \text{ are placed into } A \text{ or } B]$

$$= \frac{1}{2} E\,[\text{ cut size} \mid v_1, v_2, ..., v_i \text{ are placed, And } v_{i+1} \in A]$$
$$+ \frac{1}{2} E\,[\text{ cut size} \mid v_1, v_2, ..., v_i \text{ are placed, And } v_{i+1} \in B]$$

It is easy to observe that at least with one of the two choices of keeping $v_{i+1}$ in $A$ or $B$, the expected cut-size does not decrease. Therefore at least one of the following enequalities is true:

$$E\,[\text{ cutsize} \mid v_1, v_2, ..., v_i \text{ are placed, And } v_{i+1} \in A] \geq E\,[\text{ cutsize} \mid v_1, v_2, ..., v_i \text{ are placed }]$$

$$E\,[\text{ cutsize} \mid v_1, v_2, ..., v_i \text{ are placed into, And } v_{i+1} \in B] \geq E\,[\text{ cutsize} \mid v_1, v_2, ..., v_i \text{ are placed}]$$

Therefore, our strategy strategy is to look at the expected size of the cut in both cases (putting $v_{i+1}$ in $A$, and in $B$), and put $v_{i+1}$ in the set which leads to a greater expected cut-size. For a moment assume that we can compare the expected cut sizes efficiently.

We know that $E\,[cut\,size] \geq \frac{m}{2}$ for the case when the choices are randomized from the beginning. Now, starting from the beginning, we use the above strategy to place vertices $v_1, v_2, ..., v_n$. Therefore, we have

$$\begin{aligned}
\frac{m}{2} &\leq E\,[\text{ cutsize} \mid v_1 \text{ is placed according to above strategy}] \\
&\leq E\,[\text{cutsize} \mid v_1, v_2 \text{ are placed according to above strategy}] \\
&\leq \\
&\vdots \\
&\leq E\,[\text{ cutsize} \mid v_1, v_2, v_3, ...v_n \text{ are placed}]
\end{aligned}$$

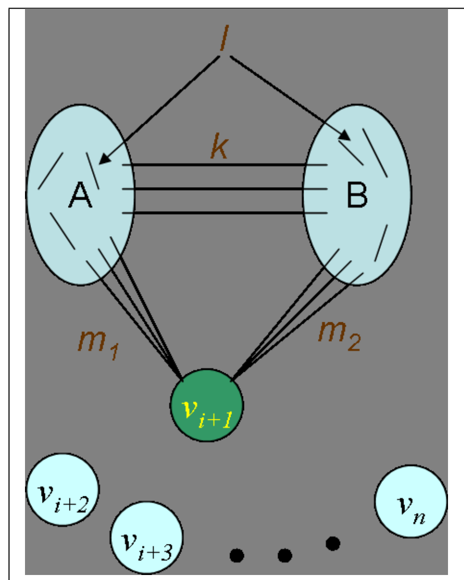

Fig. 1

Now, we will see how to find which expectation is bigger. Consider the situation where vertices $v_1, v_2, ...v_i$ are placed in $A$ or $B$, and we are making the choice for $v_{i+1}$. This situation on an undirected graph is shown in Fig. 1. Let there be $k$ edges between vertices in set $A$ and vertices in set $B$, $m_1$ vertices between $v_{i+1}$ and vertices in set $A$, and $m_2$ vertices between

$v_{i+1}$ and vertices in set $B$. Also, let the total number of vertices within sets $A$ and $B$ be $l$. Therefore,

$$\text{cutsize(tillnow)} = k$$

$$\text{number of edges involving } v_{i+2}, v_{i+3}, ..., v_n = m - k - m_1 - m_2 - l$$

Now, let us consider the expected cut-size for the two possible choices of $v_{i+1}$.

$$E\left[\text{cutsize} \mid v_1, v_2, ..., v_i \text{ are placed, And } v_{i+1} \in A\right] = k + m_2 + \frac{m - k - m_1 - m_2 - l}{2}$$

$$E\left[\text{cutsize} \mid v_1, v_2, ..., v_i \text{ are placed, And } v_{i+1} \in B\right] = k + m_1 + \frac{m - k - m_1 - m_2 - l}{2}$$

The difference of the above two values is $m_1 - m_2$. Therefore, the simple **greedy** strategy is to just compare $m_1$ and $m_2$ and put $v_i$ in that set which leads to a larger value of current cut-size, i.e.,

**if** $m_1 > m_2$, put $v_i$ in $B$ **else** put $v_i$ in $A$

**Derandomized algorithm:** Below is a deterministic algorithm based on our above analysis.

MAX-CUT
1 $A = \{v_1\}$ , $B = \phi$.
2 **for** $i$=2 to $n$
 2.1 place $v_i$ into a set that maximizes the current cut-size

Therefore, there exists a derandomized algorithm with the same performance as the randomized algorithm. This technique is known as derandomization via *Conditional Expectations.*

This technique can very well be applied to a class of randomized algorithms.

## Is Randomness Really Useful?

Now we get to the important question of whether or not randomness really adds any power to the computation. Specifically, the question we want to answer here is:

Is $BPP = P$? or Is $RP = P$?

We will analyze here the second question here (Is $RP = P$?) simply for the simplicity of analysis.

**Definition:** A language $L$ is in $RP$ if there is a polynomial-time deterministic algorithm $A$ and polynomial $r(\cdot)$ s.t.
$\forall x$,

$$x \in L \Rightarrow \Pr_{r \in \sum^{r(n)}} [A(x, r) accepts] \geq \frac{1}{2}$$

$$x \notin L \Rightarrow \Pr_{r \in \sum^{r(n)}} [A(x, r) accepts] = 0$$

Here, the string of random bits is represented by $r$.

We can trivially remove randomness from any randomized algorithm by increasing its time complexity. Suppose an algorithm $A$ uses $r(n)$ random bits and is $t(n)$-time bounded,

and gives the correct answer with probability $\geq \frac{1}{2}$. Then, consider the following deterministic algorithm: Run $A$ on every possible $r(n)$-bit sequence, and take the majority vote. Clearly, the running time of the above algorithm is $2^{r(n)}t(n)$ which is exponential if $r(n)$ is of the order of $n$. We are interested in eliminating (or reducing) randomness with at most a polynomial blow up in running time.

**General Strategy (Pseudo-Random Generator):** Say $L \in RP$, and say $A$ is an algorithm that witnesses that $L \in RP$, and $A$ uses $r(n)$ random bits. Algorithm $A$ can be viewed as making a series of $r(n)$ coin tosses and then using a deterministic algorithm using the outcome of those $r(n)$ coin tosses. Suppose, $l(n) \leq r(n)$, and we have a function:

$$f : \Sigma^{l(n)} \rightarrow \Sigma^{r(n)}$$

$$\forall x \Pr_{r \in \sum^{r(n)}} [A(x, r)\, accepts] \cong \Pr_{y \in \sum^{l(n)}} [A(x, f(y))\, accepts]$$

(Here, $\cong$ denotes *very close*, typically with a tolerance of something like $\frac{1}{n^2}$)

Then $L$ can be solved in time $O(2^{l(n)}t(n)t'(n))$ where $t(n)$ is the running time of $A$, and $t'(n)$ is the running time $f$ on inputs of length $l(n)$ .

Therefore, if $l(n)$ is of the order of $\log n$, then $O(2^{l(n)}t(n)t'(n))$ is polynomial.

Thus, the idea here is to start with a short random seed of $l(n)$ bits, stretch it to $r(n)$ bits, and use it in algorithm $A$. It is important to note here that since $f$ is a function, the cardinality of co-domain of $f$ can be at most $2^{l(n)}$, which is much less than $2^{r(n)}$. Also, there is no surity that the $2^{l(n)}$ distinct possible function values are uniformly distributed. Therefore, the $r(n)$ bit sequence produced is not really random, but for algorithm $A$, it may be *good enough*. Although the $r(n)$ bit string, produced by first randomly picking a $l(n)$ bit string and stretching it using $f$, is not a true random string, the behaviour of $A$ does not change. If such function exists, then we can reduce (or eliminate) randomness from algorithm $A$. Such functions are called *pseudo-random generators*, and the question is: "Do such functions exist for all randomized algorithms?" If such functions exist for every language in $RP$ (with $l(n) = O(\log n)$), $RP = P$.

Over the next few lecturs we will study various types of pseudo-random generators.

## 1.2 Pairwise Independent Generator/Chor-Goldreich Generator

This generator can be used to *fool* a certain class of algorithms.

**Definition:** A function $f$ is $(m, t, l)$-pairwise independent generator if:
1. $f : \Sigma^m \rightarrow \Sigma^{tl}$
2. $f$ is computable in time $poly(tl)$.
3. $\forall i \neq j (1 \leq i, j \leq l), \forall \alpha, \beta \in \Sigma^t$

The output of $tl$ bits can be viewed as $l$ blocks, each of length $t$ as shown in Fig. 2.

$$\Pr_{x \in \Sigma^m} [f_i(x) = \alpha \wedge f_j(x) = \beta] = \frac{1}{2^{2t}}$$

where $f_i(x)$ denotes $i^{th}$ block of $f(x)$.

Typically, $t \leq m, l \geq m$.

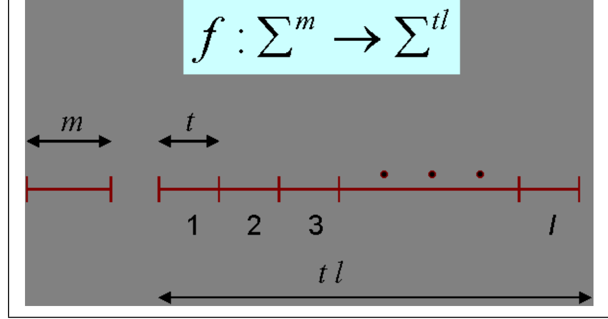


Fig. 2

To make things clearer, consider $t = 1$, i.e., $f : \Sigma^m \rightarrow \Sigma^l$

In a $l$-bit random sequence L,
$\forall i \neq j (1 \leq i, j \leq l)$

$$\Pr[L(i)L(j) = 00] = \Pr[L(i)L(j) = 01] = \Pr[L(i)L(j) = 10] = \Pr[L(i)L(j) = 11] = \frac{1}{4}$$

But if we start with a $m$-bit random sequence ($m \leq l$), and stretch it to $l$-bits using a function $f$, then above property may not hold. If the function $f$ is such that the above property is satisfied, it is called a pairwise-independent generator.

**Theorem:** $\forall l, \forall t, \forall m$, s.t. $2^m \geq l$, $m \geq t$, there exists a $(2m, t, l)$ pairwise-independent generator.

**Proof:** Let $H$ be a class of 2-universal hash functions from $\Sigma^m \rightarrow \Sigma^t$ s.t. $|H| = 2^{2m}$, i.e., each function can be represented by $2m$ bits ($h(x) = ax + b$ where $|a| = |b| = m$).
e.g.

$$H = \{h_{ab} | a, b \in GF(2^m)\}$$

$$h_{ab}(x) = \text{ first } t \text{ bits of } ax + b$$

Then, function $f$ can be defined as:

$$f(a, b) = h_{ab}(\alpha_1) \cdot h_{ab}(\alpha_2) \ldots h_{ab}(\alpha_l)$$

where $\alpha_1, \alpha_2, \ldots, \alpha_l$ are some distinct fixed elements of $GF(2^m)$. It is important to note that a different choice of the set of $l$ $\alpha$'s would result in a different function $f$.

Function $f$ is polynomial-time computable in output length $tl$, because each computation takes time of the order of $m$, and $m \geq t$. So each block can be computed in time roughly $O(t)$, and there are $l$ blocks. So the total time is $O(lt)$, which is polynomial time computable.

Now, we want to show that

$$\forall i \neq j (1 \leq i, j \leq l), \forall u, v \in \Sigma^t$$

$$\Pr_{a,b \in \sum^m} [f_i(a, b) = u \wedge f_j(a, b) = v] = \frac{1}{2^{2t}}$$

which is same as

$$\Pr_{a,b \in \sum^m} [h_{ab}(\alpha_i) = u \wedge h_{ab}(\alpha_j) = v] = \frac{1}{2^{2t}}$$

(by the property of 2-universal hash functions)