6.5420 Randomness and Computation	March 12, 2026
Lecture 10	
Lecturer: Ronitt Rubinfeld	*Scribe: Jerry Zhang*

This lecture covers:

1. Derandomizing algorithms via conditional expectations
2. An algorithm for generating uniformly random satisfying assignments for DNF formulas

1 Derandomization via conditional expectation

In a randomized algorithm, we can view making m tosses of random coins as choosing a path down a tree of depth m. The leaves of the tree correspond to outputs of the randomized algorithm: some are "good" (i.e. yield correct answers, good approximations, etc.) while others are "bad".

In a good randomized algorithm, most of these leaves are good, but we do not know the paths to reach these good leaves. So, we can try to pick these paths bit-by-bit by choosing the path that gives us a higher *conditional expectation*.

More formally, fix some randomized algorithm $\mathcal{A}$ with error probability δ that takes in some input x, and let m be the number of random bits $\mathcal{A}$ uses on x. For $1 \le i \le m$ and $r_1, \dots, r_i \in \{0, 1\}$, let $p(r_1, \dots, r_i)$ be the fraction of continuations $r_{i+1}, \dots, r_m$ of the random bits such that $\mathcal{A}$ ends on a good output, where $p(\emptyset)$ denotes that no choices have been made. Notice that, since r_{i+1} is picked randomly,

$$p(r_1, \dots, r_i) = \frac{1}{2} p(r_1, \dots, r_i, 0) + \frac{1}{2} p(r_1, \dots, r_i, 1).$$

This implies that there exists some setting of r_{i+1} such that

$$p(r_1, \dots, r_{i+1}) \ge p(r_1, \dots, r_i)$$

If we pick all r_i in this way, notice that then $p(r_1, \dots, r_{i+1}) \ge p(r_1, \dots, r_i)$ for all i, and

$$p(r_1, \dots, r_m) \ge p(r_1, \dots r_{m-1}) \ge \dots \ge p(r_1) \ge 1 - \delta > 0 \implies p(r_1, \dots, r_m) = 1.$$

We end up at a good leaf. But how do we choose the best bit at each step? Sometimes this is not possible, but for certain problems it is.

1.1 Derandomizing Max-Cut

Recall the randomized approximation algorithm for Max-Cut:

- Flip $|V|$ coins with outcomes $(r_1, \dots, r_n)$ corresponding to either S or T.
- For each i, put v_i on side r_i.

Let

$$e(r_1, \dots, r_i) = \mathbb{E}_{r_{i+1}, \dots, r_m} \left[|\text{cut}(S, T)| \text{ given } r_1, \dots, r_i \text{ are choices for } v_1, \dots, v_i \right],$$

where $e(\emptyset)$ denotes no choices have been made. Here $e(\emptyset) = \frac{|E|}{2}$ by the analysis from previous lectures.

Furthermore, let

$$S_{i+1} = \{v_j \mid j \leq i+1,\ r_j = 0\},\ \text{the nodes in } S \text{ after the first } i+1 \text{ flips}$$
$$T_{i+1} = \{v_j \mid j \leq i+1,\ r_j = 1\},\ \text{the nodes in } T \text{ after the first } i+1 \text{ flips}$$
$$U_{i+1} = \{v_j \mid j \geq i+2\},\ \text{the undecided nodes}$$

Then we have that

$$e(r_1, \ldots, r_i, r_{i+1}) = |\text{cut}(S_{i+1}, T_{i+1})| + \frac{1}{2}|I(U_{i+1})|$$

where $I(U_{i+1}) = \partial U_{i+1} \cup E(U_{i+1})$, the edges with at least one endpoint in U_{i+1}. Consider the set of edges between v_{i+1} and S or T, i.e.

$$V_{i+1}^{(S)} = \{(v_{i+1}, s) \mid s \in S\} \qquad V_{i+1}^{(T)} = \{(v_{i+1}, t) \mid t \in T\}$$

Notice then that placing v_{i+1} in S or T does not change the expected value of the cut after adding the nodes in U_{i+1} and only the value of the cut between S and T, so

$$e(r_1, \ldots, r_i, 0) = |\text{cut}(S_i, T_i)| + |V_{i+1}^{(T)}| + \frac{1}{2}|I(U_{i+1})|$$
$$e(r_1, \ldots, r_i, 1) = |\text{cut}(S_i, T_i)| + |V_{i+1}^{(S)}| + \frac{1}{2}|I(U_{i+1})|$$

Thus we can compare the sizes of $V_{i+1}^{(S)}$ and $V_{i+1}^{(T)}$ deterministically, yielding the following greedy algorithm:

Algorithm 1 Greedy Max-Cut

```
Input: Graph G
S ← ∅, T ← ∅
for i = 0, ..., n − 1 do
    Place v_{i+1} in T if |V_{i+1}^{(S)}| > |V_{i+1}^{(T)}|, else place in S
end for
Output S, T
```

2 Random DNF assignments

We first define DNF formulas.

Definition 1. *A* ***disjunctive normal form (DNF) formula*** *is a boolean formula composed of disjunctions of conjunctions, i.e. an OR of ANDs.*

For example,

$$\varphi(x_1 \ldots x_n) = x_1\bar{x}_2x_3 \vee x_2\bar{x}_3x_4x_{10} \vee x_8\bar{x}_{10}x_{11} \vee \ldots$$

is a DNF formula. It is easy to find a satisfying assignment of a DNF formula φ; pick an arbitrary clause and assign values to satisfy that clause. However, we wish to sample satisfying assignments for DNF formulas uniformly and at random.

If φ only has one clause, then we may just satisfy that clause and assign the other variables randomly. For example, if $F = x_1\bar{x}_2x_3$, we can sample satisfying assignments by enforcing $x_1 = T, x_2 = F, x_3 = T$, and picking each $x_4, \ldots, x_n$ randomly. We can extend this idea naively to formulas with more than one clause:

- Let m be the number of clauses of φ. Pick $i \in \{1, \ldots, m\}$.

- Set the variables in clause i to be true, and set the others randomly.

However, this fails. Consider the example $F = x_1 x_2 \vee x_3$. We encounter two problems:

1. Assignments satisfying clauses of different size give different probabilities:

$$\Pr[\text{output TTF}] = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$$
$$\Pr[\text{output TFT}] = \frac{1}{2} \cdot \frac{1}{4} = \frac{1}{8}$$

2. Some assignments satisfy more than one clause:

$$\Pr[\text{output TTT}] = \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{4} = \frac{3}{8}$$

We can fix the first issue by choosing the clause with probability proportional to the number of satisfying assignments it has. In particular, if we let $A_i = \{\bar{x} = (x_1, \ldots, x_n) \mid \bar{x} \text{ satisfies } C_i\}$, then we should sample C_i with probability proportional to $|A_i|$.

For the second, we can use "rejection sampling": if an assignment satisfies k clauses, then we may flip a coin with bias $\frac{1}{k}$ to correct for it. These strategies yield the following algorithm:

Algorithm 2 Uniform DNF Sampling

Input: DNF formula $\varphi = C_1 \vee \cdots \vee C_m$
For each i, $A_i \leftarrow \{\bar{x} = (x_1, \ldots, x_n) \mid \bar{x} \text{ satisfies } C_i\}$
repeat
    Pick i with probability $\frac{|A_i|}{\sum_i |A_i|}$
    Pick an assignment $\bar{b}$ from A_i uniformly at random
    $t_{\bar{b}} \leftarrow \#\{j \mid \bar{b} \text{ satisfies } C_j\}$
    Output $\bar{b}$ with probability $\frac{1}{t_{\bar{b}}}$
until success

Claim 2. *The algorithm above uniformly samples from all satisfying assignments of φ.*

Proof. For any $\bar{b}$ that satisfies φ,

$$\begin{aligned}
\Pr[\text{output } \bar{b} \text{ in round } i] &= \frac{1}{t_{\bar{b}}} \sum_{\substack{k \in [m] \\ \text{s.t. } \bar{b} \in A_k}} \Pr[\text{pick } k \text{ in round } i] \cdot \frac{1}{|A_k|} \\
&= \frac{1}{t_{\bar{b}}} \sum_{\substack{k \text{ s.t.} \\ \bar{b} \in A_k}} \frac{|A_k|}{\sum_{j=1}^{m} |A_j|} \cdot \frac{1}{|A_k|} \\
&= \frac{1}{t_{\bar{b}}} \frac{t_{\bar{b}}}{\sum_j |A_j|} = \frac{1}{\sum_j |A_j|}
\end{aligned}$$

□

Let us also analyze the runtime of the algorithm. The probability each iteration succeeds is $\frac{1}{\max t_{\bar{b}}} \geq \frac{1}{m}$, so

$$\mathbb{E}[\# \text{ of loops before success}] \leq m.$$

Since the runtime for each iteration is $\text{poly}(m+n)$, our algorithm runs in $\text{poly}(m+n)$ in expectation.

2.1 Approximate counting

Why do we care about sampling uniformly from DNF solutions? Because of a specific class of problems called counting problems.

Definition 3. *$\#P$ is the class of languages that count the number of accept paths of a non-deterministic polynomial time Turing machine.*

We have an analogous version of completeness:

Definition 4. *A language L is $\#P$ complete if:*

- $L \in \#P$
- *For every other language $M \in \#P$, there exists a polynomial time Turing reduction from M to L.*

Counting versions of hard problems usually are hard in $\#P$:

Fact 5. *#SAT (the number of assignments satisfying boolean formula φ) is $\#P$ complete.*

Perhaps surprisingly,

Fact 6. *#DNF (the number of assignments satisfying DNF formula φ) is $\#P$ complete.*

This is because the negation $\bar{\varphi}$ of a CNF formula φ is a DNF formula. So, how can we approach these problems? We use approximate counting.

Definition 7. *Let $f : L \to \mathbb{R}_{\geq 0}$ be a function from some language L representing a counting problem. A **fully polynomial randomized approximation scheme (FPRAS)** is a randomized algorithm $\mathcal{A}$ which takes as some input $x \in L$ and $\varepsilon > 0$ such that*

- *$\mathcal{A}$ returns a value y such that*

$$\Pr\left[\frac{f(x)}{1+\varepsilon} \leq y \leq (1+\varepsilon)f(x)\right] \geq \frac{3}{4}$$

- *the running time of $\mathcal{A}$ is polynomial in $|x|$ and ε^{-1}.*

For #SAT, we would like an algorithm such that given φ with $z \equiv \#$ satisfying assignments of φ and ε, it outputs y such that

$$\frac{z}{1+\varepsilon} \leq y \leq z(1+\varepsilon)$$

with probability at least $\frac{3}{4}$ with runtime $\text{poly}(|\varphi|, \varepsilon^{-1})$.