6.842 Randomness and Computation Feb 13, 2020

Lecture 4

Lecturer: Ronitt Rubinfeld *Scribe: Anlong Chua*

Given some decision problem, can we count the number of ways to satisfy it? Such questions are often referred to as "counting problems." Often, counting problems are hard. In this lecture, we will see how to efficiently approximate the answer to counting problems.

1 Counting Solutions

Let φ be a boolean formula, for example, $\varphi = x_1\overline{x_2} \vee x_3$. Three questions we can ask are:

1. Is φ satisfiable?
2. How many satisfying assignments are there?
3. Is it possible to generate a satisfying assignment uniformly at random?

Last lecture, we saw that if φ is a DNF formula, then there are efficient algorithms for answering the first and third questions. Let us give a heuristic argument for why the second problem is computationally hard. Given any CNF φ in n variables, its negation $\overline{\varphi}$ is a DNF. Since

$$\#\text{ of satisfying assignments for } \varphi = 2^n - \#\text{ of satisfying assignments for } \overline{\varphi},$$

the problem of deciding whether φ is satisfiable is reduced to deciding whether $\overline{\varphi}$ has 2^n satisfying assignments. Since deciding whether CNFs are satisfiable is a hard problem, counting solutions to a DNF must also be hard.

More generally, given a decision problem, we can turn it into a *counting problem*, which asks for the number of ways to satisfy the decision problem. Of course, the counting problem is at least as hard as the decision problem. We have just seen that it is sometimes possible for an easy decision problem to have a hard counting problem. On the other hand, there are combinatorial problems with "easy" counting problems - for example, counting the number of spanning trees in a graph. This problem can be solved in polynomial time using Kirchhoff's Matrix Tree Theorem. This discussion is summarized in Table 1.

	Decision problem	Counting problem	Approximate counting
CNF	NP-complete	#P-complete	Hard
DNF	Polytime	#P-complete	Polytime
Perfect matching	Polytime	#P-complete	Open problem [1]
Spanning trees	Polytime	Polytime	Polytime

Table 1: Hardness of some decision and counting problems.

[1] Known polytime for dense and bipartite graphs

2 Approximate Counting

We have seen that counting solutions can be hard. Can we make the situation less bleak by weakening our requirements? Let's make this concrete.

Definition 1 (Approximate Counting for Boolean Formulas)**.** *Let φ be a boolean formula. Let z be the number of satisfying assignments to φ, and let $\varepsilon > 0$ be a parameter. A* randomized approximation scheme *is an algorithm that outputs a number y such that*

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

with probability at least $3/4$*.*

If the runtime of this scheme is polynomial in $|\varphi|$ and $\frac{1}{\varepsilon}$, we call it a Fully Polynomial Randomized Approximation Scheme *(or FPRAS for short).*

One can easily generalize this definition for arbitrary counting problems.

Remark What if we want a success probability greater than 3/4? In Problem Set 1, we will see how to bootstrap the FPRAS to construct an algorithm with success probability at least $1-\delta$, for any $\delta > 0$. This comes with an $O\left(\log \frac{1}{\delta}\right)$ multiplicative cost.

Let #SAT denote the counting problem for SAT. Can one find a FPRAS for #SAT? We claim that a FPRAS for #SAT gives a polynomial time randomized algorithm for SAT. Indeed, if φ is a satisfiable CNF formula, the FPRAS outputs a positive number $y \geq \frac{1}{1+\varepsilon}$ with probability at least 3/4; and if φ is not satisfiable, the FPRAS outputs $y = 0$ with probability at least 3/4. Thus, assuming that BPP $\neq$ NP, there is no FPRAS for #SAT.

On the other hand, no such restriction applies for #DNF, since the decision problem is easy. In fact, we will construct a FPRAS for #DNF in the coming sections. This discussion is summarized in the third column of Table 1.

3 Approximate Counting for #DNF

A key idea in the construction of a FPRAS for #DNF is *downward self-reducibility.*

3.1 Downward Self-Reducibility

Roughly speaking, saying that a problem is downward self-reducibile (dsr) is saying that one can compute its solution by solving smaller subproblems, and piecing these together via polynomial time computation.

Let's apply this to #DNF. If $\varphi = \varphi(x_1, \ldots, x_n)$ is a boolean formula in n variables, let $\varphi_1 = (1, \ldots, x_n)$ be the formula that results when hard-coding $x_1 = 1$, and similarly let $\varphi_0 = (0, \ldots, x_n)$. We iteratively define $\varphi_{b_1 b_2 \ldots b_k} = \varphi(b_1, \ldots, b_k, x_{k+1}, \ldots, x_n)$ for $b_i \in \{0, 1\}$.

For any boolean formula φ, let $\#\varphi$ be the number of satisfying assignments to φ. Then

$$\#\varphi = \#\varphi_0 + \#\varphi_1 = \#\varphi_{00} + \#\varphi_{01} + \#\varphi_{10} + \#\varphi_{11} = \ldots.$$

Example 2. *Let $\varphi = \overline{x_2} \vee x_2 x_1 \vee \overline{x_2} x_1$. The number of satisfying assignments can be computed using the following tree.*

$$\begin{tikzcd}
 & & & \#\varphi = 3 \arrow[dll] \arrow[drr] & & & \\
 & \#\varphi_0 = 1 \arrow[dl] \arrow[dr] & & & & \#\varphi_1 = 2 \arrow[dl] \arrow[dr] & \\
\varphi_{00} = \text{True} & & \varphi_{01} = \text{False} & & \varphi_{10} = \text{True} & & \varphi_{11} = \text{True}
\end{tikzcd}$$

Let $F_{b_0 \dots b_k} = \#\varphi_{b_1 b_2 \dots b_k}$. For example, this means that F is the number of satisfying assignments of φ, which is what we wish to compute. Let

$$S_1 = \frac{F_1}{F}$$

be the fraction of satisfying assignments with $x_1 = 1$. The key idea is that we can approximate S_1, and approximate F_1 via the dsr strategy. This allows us to approximate F. Thus, instead of having to compute the number of solutions for the entire tree (with 2^n nodes), we only have to explore one path down the tree.

3.2 Downward Self-Reducibility for #DNF

Now, assume that φ is a DNF formula. The key idea is to use the polynomial time uniform satisfying assignment generator from last lecture. Generate k satisfying assignments uniformly at random, and set

$$\widehat{S_1} = \frac{\text{\# generated assignments with } x_1 = 1}{k}.$$

If k is large enough, Chernoff bounds guarantee that $\widehat{S_1}$ is a good approximation of S_1. We will make this precise in Section 3.4.

Why did we arbitrarily choose to count solutions with $x_1 = 1$? What if there is no satisfying assignment with $x_1 = 1$? We will address these concerns later. For now, assume that we choose some sequence of bits $b_1 \dots b_n$ which is actually a satisfying assignment for φ. Let

$$S_{b_1 \dots b_i} = \frac{F_{b_1 \dots b_i}}{F_{b_1 \dots b_{i-1}}}.$$

Then

$$F = \frac{F_{b_1}}{S_{b_1}} = \frac{F_{b_1 b_2}}{S_{b_1} S_{b_1 b_2}} = \frac{F_{b_1 \dots b_n}}{\prod_i S_{b_1 \dots b_i}} = \frac{1}{\prod_i S_{b_1 \dots b_i}}.$$

Thus, if we can approximate each $S_{b_1 \dots b_i}$, we can approximate F. This formula reveals two issues with our approach. First, we require that $b_1 \dots b_n$ be a satisfying assignment. Indeed, if this were not the case, our algorithm would just return $F = 0$. Second, Chernoff bounds only give us additive estimates for $S_{b_1 \dots b_i}$. However, what we really want is to obtain multiplicative estimates for F, so we will have to somehow turn these additive estimates into multiplicative ones.

To solve both problems, we do not choose each bit b_i arbitrarily. Instead, we pick the $b_i = 1$ if more generated satisfying assignments have $x_i = 1$ (than $x_i = 0$), and $b_i = 0$ otherwise.

We first claim that following this strategy will always lead us to a satisfying assignment. This is clear, because our generator only outputs satisfying assignments. Since a nonempty DNF always has satisfying assignments, we can inductively show that this strategy only ever follows branches with satisfying assignments.

Next, if k is large enough (say $k = O(n)$), Chernoff bounds tell us that we will pick a child with at least $1/3$ of the satisfying assignments with high probability (see Section 3.4). To be precise, we want the probability of error to be $\delta \approx \frac{1}{8n}$. If this happens, setting k large enough ($k \in \mathrm{Poly}(n, \varepsilon^{-1})$) to give an additive error of say $\frac{\varepsilon}{6n}$ (with probability $1 - \frac{1}{8n}$), we have

$$r + \frac{\varepsilon}{6n} = r\left(1 + \frac{\varepsilon}{6nr}\right) \le r\left(1 + \frac{\varepsilon}{2n}\right),$$

and a similar manipulation shows that the Chernoff lower bound for $\widehat{S_i}$ becomes $r\left(1 + \frac{\varepsilon}{2n}\right)^{-1}$, where $r = \frac{F_{b_1 \ldots b_i}}{F_{b_1 \ldots b_{i-1}}} \ge 1/3$.

How often do these estimates all hold? Recall that we chose k large enough so the probability we go down a subtree with less than $1/3$ of the satisfying assignments is $\delta \approx \frac{1}{8n}$. The probability that our estimate for r is not within our range is also bounded by $\frac{1}{8n}$. Taking a union bound over these events, the probability that we make such a mistake at least once is at most $2n(1/8n) = 1/4$. Therefore, the probability that we always go down a subtree with many (at least $1/3$ of the current total) solutions, and always get a good estimate of r, is at least $3/4$. In this case,

$$\text{output } = \frac{1}{\prod_i \widehat{S_{b_1 \ldots b_i}}} \le \frac{(1 + \varepsilon/2n)^n}{\prod_i S_{b_1 \ldots b_i}} = \left(1 + \frac{\varepsilon}{2n}\right)^n F \le (1 + \varepsilon)F$$

with a similar lower bound; this is exactly what is required of a FPRAS.

To summarize, our FPRAS for #DNF is:

Algorithm 1: FPRAS for #DNF

```
    Result: Approximate number of solutions to a DNF formula
    Estimate $S_0$ and $S_1$ using uniform generation
    $b_1 \leftarrow \mathrm{argmax}_{\{0,1\}}(S_0, S_1)$
    Recurse to compute $F_{b_1}$
    Return $F = F_{b_1}/S_{b_1}$
```

Its runtime is

$$n \cdot \#\text{ samples required to get } \frac{\varepsilon}{4n} \text{ additive error} \cdot \text{runtime of generator.}$$

Via Chernoff bounds, the second term is polynomial in $(\varepsilon/4n)^{-1}$, and we saw last lecture that the last term is polynomial in n. Hence the runtime of our FPRAS is polynomial in n and $\frac{1}{\varepsilon}$.

Our algorithm works whenever all our estimates of the "larger side" hold, which happens with probability at least $3/4$, by our earlier discussion.

Remark Why did we have to go through all this trouble? Couldn't we randomly generate *arbitrary* assignments of the boolean variables, and compute what fraction of these satisfy φ? Well, the problem with this method is that we may have to generate exponentially many assignments to get a statistically significant estimate of F (if, say, there are very few satisfying assignments).

3.3 Downward Self-Reducibility in General

It is not hard to see that our algorithm did not rely on any special features of the DNF problem. We only required a polynomial time solution generator to exist. Thus, for any dsr problem, we have shown how to turn a polynomial time generation algorithm into a polynomial time approximate solution counting algorithm.

3.4 Explicit Computation of Chernoff Bounds

Theorem 3. *Let $X = \sum_{i=1}^n X_i$, where $X_i = 1$ with probability p_i and $X_i = 0$ with probability $1 - p_i$, and all X_i independent. Let $\mu = \mathbb{E}[X]$. Then*

$$\mathbb{P}(|X - \mu| \geq \delta\mu) \leq 2\exp\bigl(-\mu\delta^2/3\bigr).$$

(See, for example, Prof. Goemans' notes[2].)

Recall our set-up: we are using a uniform random generator to generate solutions to #DNF. We are interested in answering the following questions:

1. We go down the subtree with more generated solutions. What is the probability we go down a subtree that has fewer than 1/3 of the total solutions?
2. Let r be the fraction of solutions with, say, $x_i = 1$. If $\widehat{r}$ is our observed fraction, how close is $\widehat{r}$ to r?

Let us answer question 1 first. Suppose we generate k solutions, and without loss of generality, the fraction of solutions with $x_i = 1$ is $r < 1/3$. Let $X_i \sim \mathrm{Ber}(r)$ be the event that we draw a solution with $x_i = 1$. Then we go down the branch with $x_i = 1$ if $\sum X_i \geq k/2$. Since $\mu = \mathbb{E}[X] = rk < k/3$, take $\delta = \frac{k}{6\mu} = \frac{1}{6r}$. Then

$$\mathbb{P}(|X - \mu| \geq \delta\mu) \leq 2\exp\bigl(-\mu\delta^2/3\bigr) \leq \exp(-ck)$$

where c is some constant. So if we take $k = O(n)$, the probability we go down the "wrong" branch is much less than $\frac{1}{8n}$, as was required.

Now we answer question 2. Re-using our notation from earlier, $X = k\widehat{r}$ and $\mu = kr$. We want to be within $\frac{\varepsilon}{6n}$ of r, so we take $\delta = \frac{k\varepsilon}{6n\mu} = \frac{\varepsilon}{6nr}$. Then (note $r \leq 1$)

$$\mathbb{P}\left(|k\widehat{r} - kr| \geq \frac{\varepsilon k}{6n}\right) \leq 2\exp\left(-\mu\delta^2/3\right) \leq \exp\left(-c\frac{\varepsilon^2}{n^2}k\right),$$

so it suffices to take $k = O((n/\varepsilon)^3) \in \mathrm{Poly}(n, \varepsilon^{-1})$. Thus our estimate of r is good with very high probability (much larger than $1 - \frac{1}{8n}$).

[2]`http://math.mit.edu/~goemans/18310S15/chernoff-notes.pdf`

4 Approximate Uniform Generation

To begin the section, let's formalize our definition of uniform solution generation.

Definition 4. *A* uniform generator *for solutions of a problem π is an algorithm that, on input x, outputs a uniform random solution for $\pi(x)$. More precisely, define*

$$S_x := \{z \mid z \text{ is a solution to } \pi(x)\}.$$

A uniform generator outputs $y \in S_x$ with probability $\frac{1}{|S_x|}$, and should run in time $\text{Poly}(|x|)$.

Extrapolating from our past examples, uniform generation might be hard, but it may be possible to approximate a uniform distribution on the solutions. The correct definition turns out to be:

Definition 5. *An* almost uniform generator *for solutions of a problem π is an algorithm that, on input x and parameter ε, satisfies*

$$\frac{1}{|S_x|}\frac{1}{1+\varepsilon} \leq \mathbb{P}[\text{output } y] \leq \frac{1}{|S_x|}(1+\varepsilon)$$

for all $y \in S_x$, and should run in time $\text{poly}(|x|, \varepsilon^{-1})$.

In our last section, we saw that given a uniform generator for a dsr problem, we could construct an approximate counting algorithm. It is natural to wonder if, given an approximate counting algorithm, we can construct a uniform generator.

4.1 Approximate Uniform Generation for #DNF

Let's try to construct an almost uniform generator for #DNF. For this purpose, first assume that we have a perfect counting algorithm for #DNF.

Using the perfect counter, we first compute $r_0 = F_0$ and $r_1 = F_1$. We recurse on the formula φ_0 with probability $\frac{r_0}{r_0+r_1}$, and recurse on the formula φ_1 otherwise.

It is easy to see that we always reach a satisfying assignment. The probability that we output a single assignment is

$$\mathbb{P}[\text{output assignment } b = (b_1 \dots b_n)] = \frac{F_{b_1}}{F} \cdot \frac{F_{b_1 b_2}}{F_{b_1}} \cdots \frac{1}{F_{b_1 \dots b_{n-1}}} = \frac{1}{F}$$

which is indeed the uniform distribution over satisfying assignments.

Now, as we have seen, perfectly counting solutions to #DNF is hard. So, what happens if we only have an approximate counter? If we run our approximate counter with error parameter $\varepsilon' < \varepsilon/2n$, then

$$\begin{aligned}\mathbb{P}[\text{output assignment } b = (b_1 \dots b_n)] &= \frac{\widehat{F_{b_1}}}{\widehat{F}} \cdot \frac{\widehat{F_{b_1 b_2}}}{\widehat{F_{b_1}}} \cdots \frac{1}{\widehat{F_{b_1 \dots b_{n-1}}}} \\ &\leq \frac{1}{F}\left(\frac{1+\varepsilon'}{1-\varepsilon'}\right)^n \leq \frac{1}{F}\frac{1}{1-\varepsilon} \approx \frac{1}{F}(1+\varepsilon)\end{aligned}$$

with a matching lower bound. Hence, we get an almost uniform generator for #DSR.

It is easy to generalize this construction to any dsr problem. In fact, Jerrum, Valiant and Vazirani proved:

Theorem 6 (Jerrum, Valiant, Vazirani)**.** *For any problem in NP that is dsr, a polytime almost uniform generator exists if and only if a polyime approximate counter exists.*