princeton univ. F'13 cos 521: Advanced Algorithm Design

Lecture 20: Counting and Sampling Problems

Lecturer: *Sanjeev Arora* Scribe:

Today's topic of counting and sampling problems is motivated by computational problems involving multivariate statistics and estimation, which arise in many fields. For instance, we may wish to integrate a function $f(x_1, x_2, \ldots, x_n)$ over some domain:

$$\int_{x_1=0}^{1} \int_{x_2=0}^{1} \cdots \int_{x_n=0}^{1} f(x_1, x_2, \ldots, x_n) dx_1 dx_2 \ldots dx_n.$$

This kind of integration can be NP-hard even when f is polynomial-time computable.

Turns out these are related to combinatorial *counting problems* introduced by Valiant in 1979. The goal in such problems is to compute the size of a set S where we can test *membership* in S in polynomial time.

Example 1 ♯SAT is the problem where, given a boolean formula φ, we have to compute the *number* of satisfying assignments to φ. Clearly it is NP-hard since if we can solve it, we can in particular solve the *decision* problem: *decide* if the number of satisfying assignments at least 1.

♯CYCLE is the problem where, given a graph $G = (V, E)$, we have to compute the number of *cycles* in G. Here the decision problem ("is G acyclic?") is easily solvable using breadth first search. Nevertheless, the counting problem turns out to be NP-hard.

♯SPANNINGTREE is the problem where, given a graph $G = (V, E)$, we have to compute the number of *spanning trees* in G. This is known to be solvable using a simple determinant computation (Kirchoff's matrix-tree theorem) since the 19th century.

Valiant's class ♯P captures most interesting counting problems. Many of these are NP-hard, but not all. You can learn more about them in *Computational Complexity* next term. □

It is easy to see that the above integration problem can be reduced to a counting problem with some loss of precision. First, recall that integration basically involves summation: we appropriately discretize the space and then take the sum of the integrand values (assuming in each cell of space the integrand doesn't vary much). Thus the integration reduces to some sum of the form

$$\sum_{x_1 \in [N], x_2 \in [N], \ldots, x_n \in [N]} g(x_1, x_2, \ldots, x_n),$$

where $[N]$ denotes the set of integers in $0, 1, \ldots, N$. Now assuming $g(\cdot) \geq 0$ this is simply the size of the following set (note if g is computable in polynomial time then we can test membership in this set in polynomial time given (x, c)):

$$\{(x, c) : x \in [N]^n;\ c \leq g(x)\}.$$

We will also be interested in *sampling* a random element of a set S. In fact, this will turn out to be intimately related to the problem of counting.

1 Counting vs Sampling

We say that an algorithm is an *approximation scheme* for a counting problem if for every $\epsilon > 0$ it can output an estimate of the size of the set that is correct within a multiplicative factor $(1+\epsilon)$. We say it is a *randomized fully polynomial approximation scheme* (FPRAS) if it is randomized and it runs in $\text{poly}(n, 1/\epsilon, \log 1/\delta)$ time and has probability at least $(1-\delta)$ of outputting such an answer. We will assume $\delta < 1/poly(n)$ so we can ignore the probability of outputting an incorrect answer.

An *fully polynomial-time approximate sampler* for S is one that runs in $\text{poly}(n, 1/\epsilon, \log 1/\delta)$ and outputs a sample $u \in S$ such that $\sum_{u \in S} \left| \Pr[u \text{ is output}] - \frac{1}{|S|} \right| \leq \epsilon$.

Theorem 1 (Jerrum, Valiant, Vazirani 1986)
For "nicely behaved "counting problems (the technical term is "downward self-reducible") sampling in the above sense is equivalent to counting (i.e., a algorithm for one task can be converted into one for the other).

Proof: For concreteness, let's prove this for the problem of counting the number of satisfying assignments to a boolean formula. Let $\sharp\varphi$ denote the number of satisfying assignments to φ.

Sampling $\Rightarrow$ Approximate counting: Suppose we have an algorithm that is an approximate sampler for the set of satisfying assignments for any formula. Given any formula $\varphi(x_1, x_2, \ldots, x_n)$ take $\text{poly}(n)$ samples from it and let p_0 be the fraction that have a 0 in the first bit x_i, and p_1 be the fraction that have a 1. Assume $p_0 \geq 1/2$. Now produce the formula $\varphi|_{x_1=0}$ obtained from φ by fixing x_1 to 0, then use the same algorithm recursively on this smaller formula to estimate N_0, the value of $\sharp\varphi|_{x_1=0}$. Then output N_0/p_0 as your estimate of $\sharp\varphi$.

This works because if $p_0 \geq 1/2$ then Chernoff bounds imply that p_0 is correct (up to $1 + 1/poly(n)$ factor) estimate of the fraction of assignments in which the first bit is 0. Assuming inductively the estimate N_0 is also approximately correct, the output N_0/p_0 is a good estimate.

Approximate counting $\Rightarrow$ Sampling: This involves reversing the above reasoning. Given an approximate counting algorithm we are trying to generate a random satisfying assignment. First use the counting algorithm to approximate $\sharp\varphi|_{x_1=0}$ and $\sharp\varphi$ and take the ratio to get a good estimate of p_0, the fraction of assignments that have 0 in the first bit. (If p_0 is too small, then we have a good estimate of $p_1 = 1 - p_0$.) Now toss a coin with $\Pr[\text{heads}] = p_0$. If it comes up heads, output 0 as the first bit of the assignment and then recursively use the same algorithm on $\varphi|_{x_1=0}$ to generate the remaining $n-1$ bits. If it comes up tails, output 1 as the first bit of the assignment and then recursively use the same algorithm on $\varphi|_{x_1=1}$ to generate the remaining $n-1$ bits.

Note that the quality ϵ of the approximation suffers a bit in going between counting and sampling. There are a few other subtleties we are pushing under the rug. $\square$

1.1 Monte Carlo method

The classical method to do counting via sampling is the *Monte Carlo* method. A simple example is the ancient method to estimate the area of a circle of unit radius. Draw the circle in a square of side 2. Now throw darts at the square and measure the fraction that fall in the circle. Multiply that fraction by 4 to get the area of the circle.

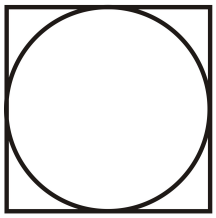

Figure 1: Monte Carlo (dart throwing) method to estimate the area of a circle.

Now replace "circle" with any set S and "square" with any set Ω that contains S and can be sampled in polynomial time. Then just take many samples from Ω and just observe the fraction that are in S. This is an estimate for $|S|$. The problem with this method is that usually the obvious Ω is much bigger than S, and we need $|\Omega| \,/\, |S|$ samples to get any that lie in S. (For instance the obvious Ω for computing $\sharp\varphi$ is the set of all possible assignments, which may be exponentially bigger.)

2 Dyer's algorithm for counting solutions to KNAPSACK

The Knapsack problem models the problem faced by a kid who is given a knapsack and told to buy any number of toys that fit in the knapsack. The problem is that not all toys give him the same happiness, so he has to trade off the happiness received from each toy with its size; toys with high happiness/size ratio are prefered. Turns out this problem is NP-hard if the numbers are given in binary. We are interested in a counting version of the problem that uses just the sizes.

Definition 1 *Given n weights $w_1, w_2, \ldots, w_n$ and a target weight W, a* feasible solution *to the knapsack problem is a subset T such that $\sum_{i \in T} w_i \leq W$.*

We wish to approximately count the number of feasible solutions. This had been the subject of some very technical papers, until M. Dyer gave a very elementary solution.

First, we note that the counting problem can be solved *exactly* in $O(nW)$ time, though of course this is not polynomial since W is given to us in binary, i.e. using $\log W$ bits. The idea is dynamic programming. Let $\mathrm{Count}(i, U)$ denote the number of feasible solutions involving only the first i numbers, and whose total weight is at most U. The dynamic programming follows by observing that there are two types of solutions: those that involve the ith element, and those that don't. Thus

$$\text{Count}(i,U) = \begin{cases} \text{Count}(i-1,U-w_i) + \text{Count}(i-1,U) \\ 1 \qquad \text{if } i=1 \text{ and } w_1 \le U \\ 0 \qquad \text{if } i=1 \text{ and } w_1 > U \end{cases}$$

Denoting by S the set of feasible solutions, $|S| = \text{Count}(n,W)$. But as observed, computing this exactly is computationally expensive and not polynomial-time. Dyer's next idea is to find a set Ω containing S but at most n times bigger. This set Ω can be exactly counted as well as sampled from. So then by the Monte Carlo method we can estimate the size of S in polynomial time by drawing samples from Ω.

Ω is simply the set of solutions to a Knapsack instance in which the weights have been *rounded* to lie in $[0,n^2]$. Specifically, let $w'_i = \lfloor \frac{w_i n^2}{W} \rfloor$ and $W' = n^2$. Then Ω is the set of feasible solutions to this modified knapsack problem.

CLAIM 1: $S \subseteq \Omega$. (Consequently, $|S| \le |\Omega|$.)
This follows since if $T \in S$ is a feasible solution for the original problem, then $\sum_i w'_i \le \sum_i w_i n^2/W \le n^2$, and so T is a feasible solution for the rounded problem.

CLAIM 2: $|\Omega| \le n\,|S|$.
To prove this we give a mapping g from Ω to S that is at most n-to-1.

$$g(T') = \begin{cases} = T' \qquad \text{if } T' \in S \\ = T' \setminus \{j\} \quad \text{(else)} \qquad \text{where } j = \text{index of element in } T' \text{ with highest value of } w'_j \end{cases}$$

In the second case note that this element j satisfies $w_j > W/n$ which implies $w'_j \ge n$.

Clearly, g is at most n-to-1 since a set T in S can have at most n pre-images under g. Now let's verify that $T = g(T')$ lies in S.

$$\begin{aligned} \sum_{i \in T} w_i &\le \sum_{i \in T} \frac{W}{n^2}(w'_i + 1) \\ &\le \frac{W}{n^2} \times (W' - w'_j + n - 1) \\ &\le W \qquad (\text{since } W' = n^2 \text{ and } w'_j \ge n\ ) \end{aligned}$$

which implies $T \in S$. □

Sampling algorithm for Ω To sample from Ω, use our earlier equivalence of approximate counting and sampling. That algorithm needs an approximate count not only for $|\Omega|$ but also for the subset of Ω that contain the first element. This is another knapsack problem and can thus be solved by Dyer's dynamic programming. And same is true for instances obtained in the recursion.