CS294 Markov Chain Monte Carlo: Foundations & Applications **Fall 2009**

Lecture 15: October 27

Lecturer: Alistair Sinclair *Scribes:*

15.1 Introduction

This lecture further develops the connection between random sampling and approximate counting. Informally, the two main results we will prove in this lecture are as follows:

1. Random sampling is "equivalent" to approximate counting.
2. A counting problem either has a very good approximation algorithm or it has none at all (within any polynomial factor).

Recall a counting problem is a function $f : \Sigma^* \mapsto \mathbb{N}$ that maps input (strings) to the number of solutions. For example, in 3-SAT, the input is a CNF formula and the output is the number of satisfying assignments. Almost all interesting counting problems belong to the class #P, which means that there is a polynomial time nondeterministic Turing machine with the property that, for all x, $f(x)$ is the number of accepting computations of the machine on input x.

A *fully polynomial randomized approximation scheme* (f.p.r.a.s.) for f is a randomized algorithm which, on input $(x, \epsilon) \in \Sigma^* \times (0, 1]$, outputs a random variable Z such that

$$\Pr[(1+\epsilon)^{-1} f(x) \leq Z \leq (1+\epsilon) f(x)] \geq 3/4$$

and runs in time $\text{poly}(|x|, \epsilon^{-1})$.

If Z is in the range $[(1+\epsilon)^{-1} f(x), (1+\epsilon) f(x)]$, we say that Z estimates $f(x)$ *within ratio* $(1+\epsilon)$. We choose to work with this definition rather than $(1-\epsilon) f(x) \leq Z \leq (1+\epsilon) f(x)$ (which is almost equivalent for small ϵ)because it makes the algebra a little cleaner, and also still makes sense when ϵ is larger than 1 (as it will be in Section 15.3).

Note that the estimate could be arbitrarily bad with probability 1/4. However, this probability can be reduced to any desired $\delta > 0$ by performing $O(\log \delta^{-1})$ trials and taking the median.

A f.p.r.a.s. is considered a very good algorithm, since most interesting counting problems we care about are #P-complete, so a polynomial time exact algorithm is impossible (unless P = NP). A f.p.r.a.s. is essentially the best approximation one can hope for for such problems. Note also that we cannot even hope for a f.p.r.a.s. for counting versions of NP-complete problems, because a f.p.r.a.s. always allows us to tell the difference between 0 and non-zero w.h.p., which would allow us to solve the decision problem (w.h.p.) in polynomial time. Hence interest focuses on those counting problems that are #P-complete but whose decision version is in P. There are many such examples, including: counting matchings (monomer-dimer systems), counting perfect matchings (dimer systems), computing the partition function of the Ising model, computing the volume of a convex body, counting independent sets (the hard core model), counting bases of a matroid, counting colorings of a graph with $q \geq \Delta + 1$ colors, etc.

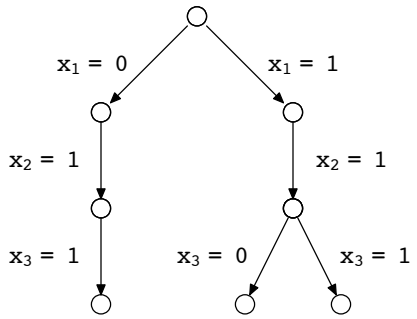

Figure 15.1: An example of a self-reducibility tree for the SAT instance $(\neg x_1 \vee x_2) \wedge (x_1 \vee x_2) \wedge (x_1 \vee x_3)$.

15.2 Equivalence between counting and sampling

Last time, we showed how to count the number of colorings of a graph G by reducing the problem to sampling colorings of G, thus obtaining a f.p.r.a.s. by invoking sampling. Although a similar approach can be used for a number of other problems, this technique is rather problem-specific. Today, we will give a *generic* reduction from counting to sampling and vice-versa, which is captured in the following theorem:

Theorem 15.1 *For all* self-reducible *NP problems, there exists a f.p.r.a.s. for counting iff there exists a polynomial time algorithm for (almost) uniform sampling.*

Rather than giving a formal definition of self-reducibility, we illustrate it with an example:

Example (SAT) Consider a boolean formula, for example,

$$(\neg x_1 \vee x_2) \wedge (x_1 \vee x_2) \wedge (x_1 \vee x_3).$$

We can represent the solutions to this SAT instance by a *self-reducibility tree*, as show in Figure 15.1. Each root-to-leaf path in this tree is a satisfying assignment. Along the path, the edges leaving the i-th internal node correspond to the assignments of variable i that eventually lead to complete satisfying assignments. Note that one or two outgoing edges are both possible, but not zero, because we would not have reached that node in the first place. Another way to see the tree is to consider it as the set of all accepting paths (witnesses) of a nondeterministic Turing machine that solves the decision version of the SAT problem.

The tree is of depth m, where m is the number of variables in the SAT formula. Each subtree at level i corresponds to a smaller SAT instance, which is the original instance but with the variables $x_1, \cdots, x_i$ already set to certain values. This recursiveness is exactly the self-reducibility property.

More generally, for an NP search problem, self-reducibility means that the set of witnesses/solutions can be partitioned into a polynomial number of sets, each of which is in 1-1 correspondence with the set of solutions of a smaller instance. The number of partitions is the maximum branching factor of the tree, which in our example is two, but more generally could be any polynomial in size of the instance. Note that the tree has polynomial depth because the size of the instance decreases at each level of the tree. We may assume w.l.o.g. that all leaves are at the same depth. The number of leaves is exactly the number of solutions, i.e., the function value $f(x)$.

Proof: We will now prove Theorem 15.1, dealing with each direction separately.

⇐: Our goal is to show how to use an algorithm for random sampling in order to construct a f.p.r.a.s. For simplicity, we will assume that we can sample exactly uniformly although the theorem requires only sampling almost uniformly. As will be evident from the proof, this extra error can easily be absorbed into the error of the f.p.r.a.s. We will also assume for simplicity that the tree has branching factor 2.

Assume that we can draw perfect samples in polynomial time. We will now give a polynomial time algorithm to approximately count. The idea is to work with the self-reducibility tree. Recall that the number of solutions is the number of leaves.

In the first stage of the algorithm, we sample some leaves uniformly at random. Using these samples, we can get an estimate $\hat{r}$ of the following quantity:

$$r = \frac{\text{number of leaves in left subtree}}{\text{total number of leaves}}.$$

Without loss of generality, assume that the left subtree contains at least half of the leaves. Now, recursively estimate number of leaves in the left subtree; call this estimate $\hat{N}_1$. The key observation is that, by virtue of self-reducibility, this is just a smaller instance of the same problem, so counting the number of leaves in this subtree can be handled by invoking the same approximate counting procedure on the smaller instance. Eventually, the recursion bottoms out at a leaf, at which point the algorithm returns 1. Our final estimate of the total number of leaves is $\hat{N} = \frac{\hat{N}_1}{\hat{r}}$.

If m is the depth of the tree, a standard second moment calculation (as sketched in the previous lecture) tells us that the number of samples needed to ensure the final estimate is within ratio $(1+\epsilon)$ with probability $3/4$ is $O(m\epsilon^{-2})$ per level, or $O(m^2\epsilon^{-2})$ total. Since by assumption each sample is produced in polynomial time, we have a f.p.r.a.s.

⇒: Assume that we have a f.p.r.a.s. for counting. We will give an algorithm for almost uniform sampling. This direction is even easier than the other direction. The idea is to simply incrementally construct a path going down the tree.

At each node, we use the f.p.r.a.s. to estimate the number of leaves in the left and right subtrees, $\hat{N}_l$ and $\hat{N}_r$, respectively. Now we branch left with probability $\frac{\hat{N}_l}{\hat{N}_l+\hat{N}_r}$, right otherwise, and repeat.

To ensure a variation distance at most ϵ from uniform in the distribution on the leaves, we need the error in each invocation of the f.p.r.a.s. to be within ratio $(1+\epsilon/2m)$, where m is the number of levels in the tree; for this gives an accumulated bias at most $(1+\epsilon/2m)^{2m} \leq \epsilon$ at any leaf.

We thus invoke the f.p.r.a.s. $O(m)$ times, each time requesting an approximation within ratio $(1+\epsilon/2m)$. Since the running time of the f.p.r.a.s. scales only polynomially with the inverse error, the cost of each invocation is polynomial in m and $\log(m/\epsilon)$. Hence the entire sampling algorithm is polynomial in m and $1/\epsilon$. ■

Rejection sampling Note that, in the second implication proved above, the resulting random sampling algorithm had a running time that depended polynomially on ϵ^{-1}. We can do better than this using the idea of *rejection sampling*. Suppose we have an algorithm that runs in time polynomial in m and counts solutions within ratio $(1+1/m)$. This is certainly the case if we have a f.p.r.a.s., since we can just set its error parameter to $\epsilon = 1/m$.

Now apply the above algorithm, branching down the tree with probabilities proportional to these counting estimates. Suppose we reach leaf ℓ. The probability, p_ℓ, of reaching ℓ will deviate from its ideal value $\frac{1}{N}$ (where N is the total number of leaves) by a factor of at most $(1+1/m)^{2m} \leq e^2$. I.e., the error is bounded

by a constant factor. (We will use this fact below.) Another crucial point is that we can compute p_ℓ, simply by multiplying the branching probabilities along the path we took.

Once we have reached ℓ, we do the following. We output ℓ with probability $\frac{\alpha}{p_\ell}$, and we *fail* otherwise. Here α is a constant chosen so that $\alpha \leq p_\ell$ for all ℓ (so that the above value really is a probability). Note that this will ensure that each leaf is output with *exactly uniform* probability α. For α, we can take the value $\alpha = (1+1/m)^{-(2m+1)}\hat{N}^{-1}$; for then we have

$$p_\ell \geq \frac{1}{N}(1+1/m)^{-2m} \geq \frac{1}{\hat{N}}(1+1/m)^{-(2m+1)} = \alpha.$$

And the failure probability is given by

$$1 - N\alpha \leq 1 - \frac{N}{\hat{N}}(1+1/m)^{-(2m+1)} \leq 1 - (1+1/m)^{-(2m+2)} \leq 1 - c$$

for a constant $c > 0$. Since the failure probability in one trial is at most a constant, if we repeat until we output a leaf we get a perfectly uniform sample in $O(1)$ expected trials, each of which takes time polynomial in m. Moreover, if we repeat for a fixed number $O(\log \epsilon^{-1})$ of trials, and just output the final leaf if all of them fail, then we get a leaf within variation distance ϵ from uniform in time polynomial in m and $\log \epsilon^{-1}$.

Note that this approach breaks down if our counting estimates are within a ratio larger than $(1 + c/m)$ for constant c, since the accumulated error at the leaves is too large to be efficiently corrected by rejection sampling (the failure probability would be too large).

Note: We can also apply rejection sampling in the case where the f.p.r.a.s. may produce estimates that are not within ratio $(1 + \epsilon)$ with small probability δ. In that case the above analysis still holds except on an event of probability at most $m\delta$, so the bias in our sample is at most $m\delta$. Note that this can be made as small as we like by reducing δ by repeated trials of the f.p.r.a.s.; as we saw in the last lecture, the cost of this is $O(\log \delta^{-1})$.

15.3 All or nothing

In the following we show the surprising fact that a counting problem either has a very good approximation (in the strong sense of an f.p.r.a.s.), or cannot be approximated in any reasonable sense in polynomial time. Specifically, we prove the following theorem from [JS89]:

Theorem 15.2 *For a self-reducible problem, if there exists a polynomial time randomized algorithm for counting within a factor of* $(1 + \mathit{poly}(|x|))^{\pm 1}$*, then there exists a f.p.r.a.s.*

Note that this theorem says that, if we can approximately count colorings (say) in polynomial time within a factor of 1000, or even within a factor of n^{1000}, then we can get an f.p.r.a.s. for colorings!

Corollary 15.3 *For a self-reducible counting problem, one of the following two holds:*

1. *there exists a f.p.r.a.s.;*
2. *there does not exist a polynomial time approximation algorithm within any polynomial factor.*

This dichotomy between approximable and non-approximable is very different from the situation with optimization problems, for which many different degrees of approximability exist (e.g., approximation schemes $(1 \pm \epsilon$ for any $\epsilon)$; constant factor; logarithmic factor, polynomial factor etc.)

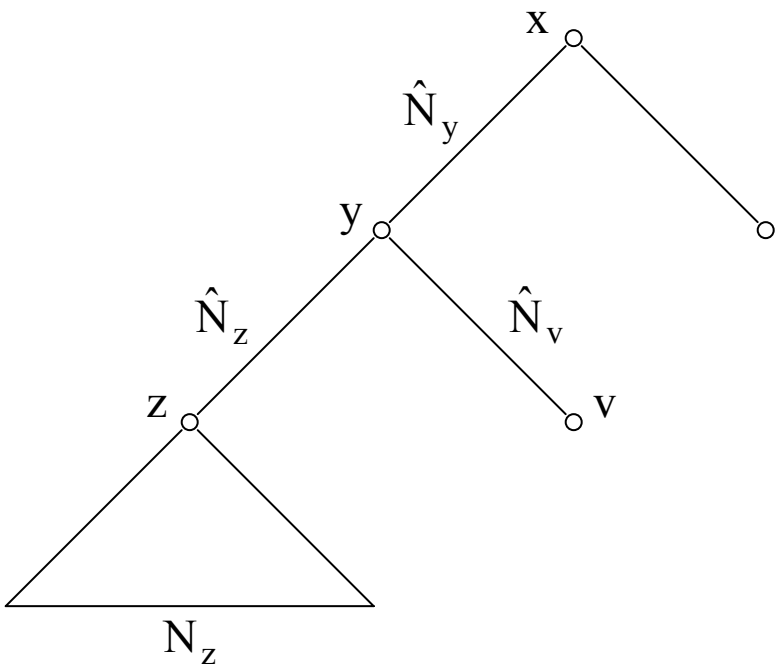

Figure 15.2: Part of a self-reducibility tree. $\hat{N}_z$ denotes the estimated number of leaves in the subtree rooted at z. The edges connected to a node y are weighted according to the sizes of subtrees as shown.

We now give a constructive proof of Theorem 15.2.

Proof: We again work with the self-reducibility tree. For any node z of the tree, let N_z denote the number of leaves in the subtree rooted at z. We assume we have an algorithm (a black box) that provides counting estimates within a factor of $\alpha = \text{poly}(n)$; i.e., for each z the algorithm outputs an estimate $\hat{N}_z$ that satisfies

$$\alpha^{-1} N_z \leq \hat{N}_z \leq \alpha N_z. \tag{15.1}$$

As will be clear from the proof below (**exercise**), we can allow Z to be a random variable, and we need only assume that (15.1) holds with high probability. However, for simplicity we just work with (15.1) deterministically.

Consider Figure 15.2. Assume the tree height is m. Weight each edge in the tree as $w_{zy} = \hat{N}_z$, where y is the parent of z; thus the weight of an edge is an estimate (provided by the counter) of the number of leaves in the subtree rooted at its lower end. Now consider the Markov chain defined by weighted random walk on the tree; i.e., at any given vertex, we choose an incident edge proportional to its weight and move along that edge. Note that both upward moves (to the parent) and downward moves (to a child) are allowed. We may assume w.l.o.g. that the weight on every leaf edge is exactly 1 (because these correspond to trivial instances of the counting problem, for which we can assume the f.p.r.a.s. has no error).

The intuition for this MC is the following. The downward moves correspond exactly to the vanilla process we defined earlier, in which we use counting estimates to guide our choice of a root-leaf path. The problem with that process, however, is that it requires the counting estimates to be within ratio about $(1 \pm 1/m)$; otherwise we can build up a large bias by the time we reach the leaves. (For example, if the counting estimates are off by a constant factor, we can get a bias at the leaves that is exponential in m.) This was not a problem before because we assumed we had an f.p.r.a.s. Now, however, we are allowed to assume only that our counting

estimates are within some (possibly very large) constant or even polynomial. This is handled by the upward moves of the walk: suppose the weight on some edge is too large (i.e., it is an overestimate of the number of leaves in the subtree below it). Then this very same edge will tend to pull the process back upward in the next step! In this sense the process is "self-correcting." Remarkably, this self-correction actually works in a rigorous sense, as we will now prove.

We will prove the following three claims about the above Markov chain:

1. The stationary distribution π is uniform over the leaves;
2. The total weight of the leaves in π is at least $\frac{\text{constant}}{\alpha m}$;
3. The mixing time is $\tau_{\text{mix}} = O(m^2\alpha^2 \log(\alpha m))$.

Note that the above three facts establish the theorem. Claim 1 gives us uniformity over solutions; claim 2 means that we only have to take expected $O(\alpha m)$ samples from the stationary distribution until we see a leaf (solution) (if the final state of the MC is not a leaf, we reject and start a new simulation); and claim 3 ensures that the time to generate a sample is polynomial. The overall expected running time of the random sampling algorithm will be $O(\alpha^3 m^3)$.

Let us start by observing that the stationary distribution of a weighted random walk is always proportional to the (weighted) vertex degrees, i.e., $\pi(x) = D(x)/D$, where $D(x) = \sum_{x\sim y} w_{xy}$ and $D = \sum_x D(x)$. Note that $D(x) = 1$ for every leaf x, which establishes claim 1.

Next, let us compute the total weight of the leaves in the stationary distribution:

$$\sum_{\text{leaves } x} \pi(x) = \frac{\#\ \textit{leaves}}{D} = \frac{N}{D}.$$

The denominator here is

$$D = \sum_x D(x) = 2 \times \text{sum of all edge weights} \le 2\alpha m N, \tag{15.2}$$

where the inequality follows from the fact that the sum of the edge weights on any given level of the tree approximates N (the total number of leaves) within a factor of α. Hence,

$$\sum_{\text{leaves } x} \pi(x) \ge \frac{N}{2\alpha m N} = \frac{1}{2\alpha m},$$

which establishes claim 2.

Finally, we bound the mixing time using flows arguments. Note that since the underlying graph of the MC is a tree, there is a unique simple path $x \rightsquigarrow y$ for any pair of vertices x, y. Hence our choice of flow here is forced. Considering a generic edge e with lower vertex y, the flow along e (in either direction) is given by

$$f(e) = \sum_{z\in T_y, v\notin T_y} \pi(z)\pi(v) = \pi(T_y)\pi(\overline{T}_y) \le \pi(T_y),$$

where T_y denotes the subtree rooted at y. Now we have

$$\pi(T_y) = \sum_{z\in T_y} \frac{D(z)}{D} \le \frac{1}{D} 2\alpha m N_y,$$

where the inequality follows as in equation (15.2) above. Hence

$$f(e) \leq \frac{1}{D} 2\alpha m N_y. \tag{15.3}$$

On the other hand, the capacity of edge e is

$$C(e) = \pi(y)\frac{\hat{N}_y}{D(y)} = \frac{D(y)}{D} \times \frac{\hat{N}_y}{D(y)} = \frac{\hat{N}_y}{D}. \tag{15.4}$$

Combining (15.3) and (15.4) we get

$$\frac{f(e)}{C(e)} \leq \frac{2\alpha m N_y}{\hat{N}_y} \leq 2\alpha^2 m,$$

where we used the fact that $\frac{N_y}{\hat{N}_y} \leq \alpha$. Recalling that $\rho = \max_e \frac{f(e)}{C(e)}$, the mixing time assuming we start at the root is

$$\tau_{\text{mix}} = O(\rho \ell \log \pi(\text{root})^{-1}),$$

where $\ell = 2m$ is the length of a longest flow-carrying path. But $\pi(\text{root}) = \frac{D(\text{root})}{D} \geq \frac{1}{2\alpha m}$, yielding a mixing time of $\tau_{\text{mix}} = O(\alpha^2 m^2 \log(\alpha m))$, which is claim 3. This concludes the proof. ■

Exercise. One might think that, rather than rejecting the sample and restarting if the final state is not a leaf, it is OK to simply wait until the first time after τ_{mix} that a leaf is hit and output that leaf. This would save a factor of $O(\alpha m)$ in the running time because of the waiting time until we get a leaf. Show that this is **not** OK because it may introduce bias into the leaf probabilities.

Exercise. The waiting time factor $O(\alpha m)$ mentioned in the previous exercise can in fact be saved, by modifying the bottom of the tree slightly (adding a suitable self-loop to each leaf). Show how to do this and argue that your method does not increase the bound on the mixing time.

References

[JS89] M.R. Jerrum and A.J. Sinclair. "Approximate Counting, Uniform Generation and Rapidly Mixing Markov Chains," *Information and Computation* **82** (1989), pp. 93–133.