CO 330, LECTURE 30 SUMMARY

FALL 2017

Summary

We continued our work on Boltzmann samplers today.

Last time we left off without defining the Boltzmann sampler for products. Suppose $\mathcal{A} = \mathcal{B} \times \mathcal{C}$. Take $a = (b, c) \in \mathcal{A}$. Then a should be generated with probability

$$\frac{x^{|a|}}{A(x)} = \frac{x^{|b|+|c|}}{B(x)C(x)} = \frac{x^{|b|}}{B(x)}\frac{x^{|c|}}{C(x)}$$

so the two components are independent! This is a big speedup compared to pure recursive random generation and we get

```
def BoltzmannA=BxC
  input x
  return(BoltzmannB(x), BoltzmannC(x))
```

We tried out standard binary tree example (code linked on the webpage in both maple and python) but it didn't actually work very well for reasons which we'll clarify soon.

First, let's think about Seq. As we did for recursive random generation we could rewrite specifications to not use Seq, but for Boltzmann generators there is also a direct way.

Say to generate an element of $\textsc{Seq}(\mathcal{B})$ we first pick k and then generate an element of $\mathcal{B}^k$. The second part is easy, we just saw the components are independent so just make k calls to `BoltzmannB`. The question is with what probability each k should occur. We need

$$P_x((b_1, b_2, \ldots, b_k)) = \frac{x^{|b_1|+\cdots+|b_k|}}{A(x)} = \frac{x^{|b_1|}}{B(x)} \cdots \frac{x^{|b_1|}}{B(x)} \left(B(x)^k(1 - B(x))\right)$$

so we should choose k with probability $(1 - B(x))B(x)^k$. This is a geometric distribution.

Definition 1. *The random variable X is geometrically distributed if $P(X = k) = (1 - \lambda)\lambda^k$ for some $0 < \lambda \leq 1$.*

How do we know $B(x) \leq 1$? We know that $x < \rho$ where ρ is the radius of convergence of $A(x) = 1/(1 - B(x))$ but $B(x)$ is increasing as x increases so when $B(x) = 1$ we get a singularity. So either this singularity is ρ or ρ is smaller. Either way $B(x) \leq 1$.

Taking this together we get

```
def BoltzmannA=Seq(B)
  input x
  k = geometric_rand(B(x))
  return(BoltzmannB(x), ..., BoltzmannB(x)) (k times)
```

Next we started talking about practicalities. First how do we find the right x? We saw last class that the expected size is $E_x = xA'(x)/A(x)$ so if you want size n, solve $E_x = n$ to find the best x. Typically you will do this numerically.

However, our tree example showed that this doesn't always work very well – even if the expected value is right, there could still be many small ones generated (and occaisional very big ones). Different classes have quite different distributions for each value of x. We looked at the three examples from Figure 1 of `http://algo.inria.fr/flajolet/Publications/DuFlLoSc04.pdf`. The bumpy ones are great. The flat ones work if you use a rejection strategy (that is just keep generating until you get one in the size range you want). The peaked ones are not so good, and that is what we had in our tree example.

So how do we fix peaked ones? You need to modify the class. We already know a way to do this: pointing. If you sample from the pointed class and then forget the extra information of which atom is pointed to then you get an element of the original class. Furthermore this will still be uniform by size relative to the original class because all elements of size n have n pointed versions. Even better, in the pointed class larger objects are preferred (since more copies have been made), so the distribution is improved.

We ran code in our example (also included on the website) and it was much improved. We'll talk more about the theory of it and other practicalities on Friday.

Also bring a device for course evaluations on Friday.

References

An excellent paper on Boltzmann generators is `http://algo.inria.fr/flajolet/Publications/DuFlLoSc04.pdf`. I recommend it.

We are more specifically following `http://people.math.sfu.ca/~kyeats/teaching/math343/12-343.pdf` and `http://people.math.sfu.ca/~kyeats/teaching/math343/13-343.pdf`.