CO 330, LECTURE 31 SUMMARY

FALL 2017

Summary

If you didn't do the course evaluation in class today make sure you do it on your own time (`evaluate.uwaterloo.ca`).

We again continued our work on Boltzmann samplers.

Last time we left off by showing the code where pointing helped performance, but without going over how pointing works in this situation. Previously we have pointed before building the specification (usually to make building the specification easier). Now we already have a specification and want to know what happens to it when we point. The answers are as follows:

$$\begin{aligned}
\mathcal{Z}^\bullet &\text{ is a new atom} \\
\mathcal{E}^\bullet = \emptyset &\quad \text{so } E^\bullet(x) = 0 \\
(\mathcal{A} \cup \mathcal{B})^\bullet = \mathcal{A}^\bullet \cup \mathcal{B}^\bullet &\quad \text{when } \mathcal{A} \cap \mathcal{B} = \emptyset \\
(\mathcal{A} \times \mathcal{B})^\bullet &= \mathcal{A}^\bullet \times \mathcal{B} \cup \mathcal{A} \times \mathcal{B}^\bullet \\
\textsc{Seq}(\mathcal{A}) &= \textsc{Seq}(\mathcal{A}) \times \mathcal{A}^\bullet \times \textsc{Seq}(\mathcal{A})
\end{aligned}$$

these can be proved by looking at what derivatives do to the generating functions or by thinking about what pointing does to the objects.

With all this in mind we looked in more detail at the pointed code (on the website with lecture 30.)

The next practicality we looked at is how to implement a geometric random generator if you have a uniform random generator:

```
def geometric_rand
  input lambda
  p(k) = (1-lambda)*lambda^k
  u = rand() (uniform random in (0,1))
  s = 0
  k = 0
  while s<u
    s = s + p(k)
    k = k + 1
  return k
```

Finally, we talked a bit about speed. There's a lot more that can be said here, but that would be a different more algorithm-oriented course. See also question B2 on assignment 8. Let's forget about the desired size for the moment and just ask how long it takes to draw from the Boltzmann sampler.

For each of $\mathcal{E}$, $\mathcal{Z}$, $\cup$, $\times$, the only potentially non-constant-time things (other than the recursive calls themselves) are `rand()` and evaluating the generating function. We will assume these are both constant-time. The assumption that we can evaluate the generating function in constant time is called the *oracle assumption* (that is we have a generating function oracle). It is plausible in practice because we can precompute the generating function to some fixed precision (by series expansion if we don't have a closed form) and then only compute more if we need it, though there are issues of sensitivity to round-off errors. As to the recursive calls themselves, the worst we need to do is generate all the atoms, so all together this gives:

Proposition 1. *Let $\mathcal{A}$ be a combinatorial class specified (either iteratively or recursively) in terms of $\mathcal{E}$, $\mathcal{Z}$, $\cup$ and $\times$. Under the oracle assumptions, generating $a \in \mathcal{A}$ by the Boltzmann generator takes $O(|a|)$ time.*

References

We're continuing in the same references as last time: 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`.