CO 330, LECTURE 28 SUMMARY

FALL 2017

Summary

Today we started random generation. Random generation is useful for getting a feel for what objects in a class look like; it is useful for testing hypotheses experimentally; and it is useful for making input for testing an algorithm.

By random generation we mean the following

Definition 1. *Let $\mathcal{C}$ be a combinatorial class. An algorithm which takes as input a nonnegative integer n and returns as output an element $c \in \mathcal{C}_n$ such that every element of $\mathcal{C}_n$ has probability $\frac{1}{c_n}$ of being produced is known as a* uniform generation *algorithm.*

We will assume that we can generate random numbers uniformly in $(0, 1)$, call this function `rand()`.

Today's random generation algorithm was recursive random generation. The idea is to use the specification to randomly generate elements. Today we stuck to unlabelled classes. To start with we also restricted ourself to specifications using $\cup, +, \mathcal{Z}$ and $\mathcal{E}$ (but they could be recursive).

We just need to know how to deal with each of these things and will call recursively on the children if any. $\mathcal{Z}$ and $\mathcal{E}$ are easy.

```
def genZ
  input n (the desired size of the output object)
  if n=1
    return Z
  else
    return null
def genE
  input n
  if n=0
    return E
  else
    return null
```

Next suppose $\mathcal{A} = \mathcal{B} \cup \mathcal{C}$ with $\mathcal{B} \cap \mathcal{C} = \emptyset$. How do we pick a random element of size n? We should pick from $\mathcal{B}_n$ $\frac{b_n}{a_n}$ of the time and we should pick from $\mathcal{C}_n$ $\frac{c_n}{a_n}$ of the time in order to be uniform over all. This gives

```
def genA=BcupC)
  input n
  x = rand()
  if x < b_n/a_n
```

```
    return genB(n)
  else
    return genC(n)
```

Note that we're cheating slightly. The input is more than just n, we also need b_n and a_n (or b_n and c_n). For some classes we have closed forms for the b_n and a_n so we can just put these in the function. Otherwise we should precalculate them and put them in a table. We can do this by iterating the specification so this is reasonably fast.

Also, you should think of this as acting on the expression tree for the specification, so when we are acting on a $\cup$ vertex then the need the two subtrees (the $\mathcal{B}$ and $\mathcal{C}$) so we can call recursively on them.

Now suppose $\mathcal{A} = \mathcal{B} \times \mathcal{C}$ and we want to generate a random element of $\mathcal{A}_n$. The probability that an element of $\mathcal{A}_n$ has its $\mathcal{B}$ part of size k is

$$\frac{b_k c_{n-k}}{a_n}$$

so to generate uniformly, imagine stacking together all those chunks for $k = 0$, $k = 1$, $k = 2$, etc. which together fill up $(0, 1)$. Then we can choose a random number in $(0, 1)$ and we just need to find which k's chunk it belongs to. This is done in the following:

```
def genA=BxC
  input n
  x = rand()
  k = 0
  s = b_0*c_n/a_n
  while x > s
    k = k+1
    s = s+b_k*c_{n-k}/a_n
  return (genB(k), genC(n-k))
```

Finally what about Seq? We can always rewrite Seq away. This is because $\mathcal{A} = \textsc{Seq}(\mathcal{B})$ is equivalent to $\mathcal{A} = \mathcal{E} \cup \mathcal{B} \times \mathcal{A}$. (Check the generating functions for one justification of why.) Note that inside a larger specification you'll probably need to introduce a new class to represent each Seq and so you'll end up with more equations than you started with. Also this makes the specification recursive even if it wasn't originally recursive.

References

Today we followed `http://people.math.sfu.ca/~kyeats/teaching/math343/11-343-all.pdf`