Lexicographic generation

Contents

1 Introduction to generation and random generation

Two things we might want to do given a combinatorial class $\mathcal{A}$ would be to generate all elements of $\mathcal{A}_n$ or to generate a random element of $\mathcal{A}_n$.

What does it mean to generate something at random? For a combinatorial class $\mathcal{A}$, we want to describe a process (e.g. algorithm) that takes as input n, and outputs an object α from $\mathcal{A}$ such that any element has probability $\frac{1}{A_n}$ of being generated. We call this Uniform generation.

1.1 Features we might want in an exhaustive generation algorithm

- We might want to be able to generate the elements of $\mathcal{A}_n$ sequentially, so that we can step from one to the next. That is we might want a successor function which takes one element of $\mathcal{A}_n$ and gives the next one in the order. This is useful if we want to iterate through all elements of $\mathcal{A}_n$ and do something with each one.
- Given an object in $\mathcal{A}_n$ we might want to be able to determine the position of this object in the order, that is we might want a ranking function. We also might want to be able to determine the element in position i, that is we might want an unranking function

We would want to be able to do these things efficiently.

Definition. Let S be a finite set. A rank function is a bijection

$$\text{rank} : S \to \{0, 1, \ldots, |S| - 1\}$$

The corresponding unrank function is the inverse bijection

$$\text{unrank} : \{0, 1, \ldots, |S| - 1\} \to S$$

So $\text{rank}(s) = i$ if and only if $\text{unrank}(i) = s$.

If we have a good unranking function we can generate a random element by first generating a random element of $\{0, \ldots, \|S| - 1\}$ and then unranking. If we have a good ranking and unranking function we can generate a successor function by $\text{successor}(c) = \text{unrank}(\text{rank}(c) + 1)$. In both cases other approaches might be more efficient.

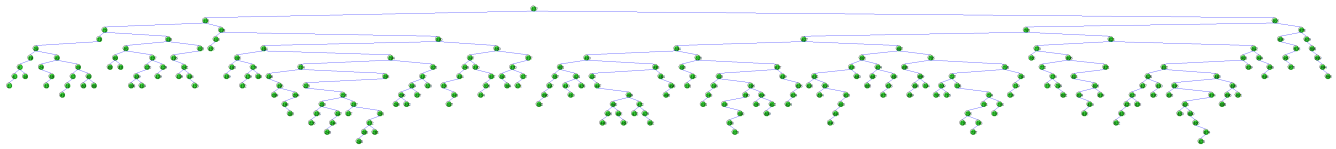
1.2 What about random generation?

Why would we want to do random generation?

- To get a sense of what the objects in a given class "look" like on average: estimate parameters like average depth of a tree; average number of cycles in a permutation etc.
- To test a hypothesis about the class using an experimental method;

- To feed as input into an algorithm to test the algorithm.

A random labelled binary tree on 200 nodes:



Hallmarks of a good random generation scheme

Correct Each element of size n is generated with equal probability.

Efficient How long does it take to generate an element of size n as a function of n?

Space Efficiency How much space does it take to generate an element

Ease of implementation How long would it take to actually code the algorithm?

Re-usability Are there optimizations possible if I am going to run the algorithm multiple times?

Generalizability Is this part of a wider framework, or simply an adhoc solution applicable to a single problem?

Strategies

1. Consider simple models like binary strings, and make a direct argument;
2. Reduce the random generation to a very simple model via a bijection;
3. Surjective method: Find a simpler class that is in a k to one correspondence– uniform generation is still preserved;
4. Ranking/unranking: Order all A_n elements. Generate uniformly a random number k between 1 and A_n, and determine the k-th element.
5. Rejection method: Generate a larger set and then filter the output;
6. Recursive method: Decompose the object by the counting probabilities
7. Markov method: use Markov chains;
8. ...

2 Lexicographic ranking and unranking

(see Donald Kreher and Douglas Stinson, *Combinatorial Algorithms*, chapter 2 for a reference on this material)

Definition. Let $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_m)$ be distinct lists of integers. $A < B$ in lexicographic order, if $a_1 = b_1, \ldots, a_{k-1} = b_{k-1}$ and either $a_k < b_k$ or $n = k - 1$ and $m \geq k$.

For example $(1, 2, 2) < (2, 1)$, $(1, 2, 2) < (1, 3, 1)$, $(1, 2, 2) < (1, 2, 2, 1)$. This is the usual dictionary ordering on words using alphabetical order on the letters (hence the name).

If we can represent our combinatorial objects as lists, then we can use lexicographic order to order them, and from that get successor, rank, and unrank functions.

2.1 Binary strings

Binary strings are simple enough to directly do random generation. Let us represent a binary string of length n by an array of length n with entries 0 or 1. Assume we have a random number generator.

Maple

```
> R:=rand(0..1); # picks a random integer in the range 0..1, i.e. picks 0 or 1
> Bin:= proc(n)
>    for i from 1 to n do :
>        A[i]:= R();
>    od;
>   return (seq(A[i], i=1..10));
>  end proc;
```

Each string has probability $(0.5)^n$ of being generated, so it is uniform. There is one call made for each n, so we say that it is linear in n. It is quite easy to implement, as we see.

Binary strings can also be lexicographically ordered since they are lists of 0s and 1s. It will be handiest if we do reverse lexicographic order (so start at the right), because then lexicographic order corresponds exactly to the order given by interpreting the binary string as a binary number. Conceptually ranking and unranking is just converting from a binary string to a binary number and back.

So we have

Algorithm: RankBin

```
input: n (length of word), w (word)
r = 0
for i from 1 to n
  if w(i) = 1
    r = r+2^i
output: r
```

Algorithm: UnrankBin

```
input: n (length of word), r (rank)
for i from 1 to n
  if r is congruent to 1 mod 2
    b(i) = 1
    r = floor(r/2)
output: b
```

2.2 Bijection to subsets

Now, let us apply a bijection. Let $\mathcal{S}$ be a set of size n. Imagine we want to generate a subset of $\mathcal{S}$ such that every subset is equally probable. How do we do it? Let us use a bijection to binary strings:

1. Order all elements: $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$;
2. Generate a random binary string $\beta = (\beta_1, \beta_2, \ldots, \beta_n)$
3. Create the subset $\{s_i : \beta_i = 1\}$.

2.3 k-subsets

In the language of subsets it is natural to ask about ranking/unranking/randomly generating subsets of size k of $\{1, \ldots, n\}$.

We can also give these a lexicographic order, for example, list the elements of the subset from smallest to largest, and then use lexicographic order on these lists.

For example the subsets of size 3 of $\{1, 2, 3, 4, 5\}$ in this order are

$$\{1,2,3\},\{1,2,4\},\{1,2,5\},\{1,3,4\},\{1,3,5\},\{1,4,5\}$$
$$\{2,3,4\},\{2,3,5\},\{2,4,5\},\{3,4,5\}$$

Using this we can get a successor function

Algorithm: SuccessorkSubset

```
input: L (current k subset as a list in increasing order), k, n
Lnew = L
i = k
while (i >= 1) and (L(i) = n-k+i)
  i=i-1
if i=0
  return no successor (L was the last in the order)
else
  for j from i to k
    Lnew(j) = L(i) + 1 + j - i
  return Lnew
```

To rank and unrank we need to count how many elements preceed a given one. Namely we need

Proposition. *Let* $\{n_1, \ldots, n_k\} \subseteq \{1, 2, \ldots, n\}$ *with* $n_1 < n_2 < \cdots < n_k$. *Write* $L = (n_1, \ldots, n_k)$. *Then*

$$\mathit{rank}(L) = \sum_{i=1}^{k} \sum_{j=n_{i-1}+1}^{n_i - 1} \binom{n-j}{k-i}$$

Proof. There are $\binom{n-(n_i+\ell)}{k-i-1}$ k-subsets (as lists in increasing order) beginning with $n_1, n_2, \ldots n_i, \ell$. For $\ell < n_{i+1}$ all such subsets preceed L in lexicographic order. Summing these completes the proof. □

From this we have

Algorithm: RankkSubset

```
input: L (current k subset as a list in increasing order), k, n
r = 0
L(0) = 0
for i from 1 to k
  if L(i-1)+1 <= L(i)-1
    for j from L(i-1)+1 to L(i)-1
      r = r + binom(n-j,k-i)
output: r
```

Algorithm: UnrankkSubset

```
input: r (rank), k, n
j = 1
for i from 1 to k
  while binom(n-j,k-i) <= r
    r = r - binom(n-j,k-i)
    j = j+1
  T(i) = j
  j = j+1
output: T
```

Successor is $O(k)$ in the worst case, and the other two algorithms are each $O(n)$ in the worst case. We can do better and rank in $O(k)$. What we need to do is use the reverse lexicographic order again. Call this the corank, and apply it to lists in decreading order

Proposition. *Let* $\{n_1, \ldots, n_k\} \subseteq \{1, 2, \ldots, n\}$ *with* $n_1 < n_2 < \cdots < n_k$. *Write* $L = (n_k, \ldots, n_1)$. *Then*

$$\mathit{corank}(L) = \sum_{i=1}^{k} \binom{n_{k-i+1} - 1}{k-i+1}$$

Proof. There are $\binom{n_{k-i+1}-1}{k-i+1}$ lists which begin with $n_k, n_{k-1}, \ldots, n_{k-i+2}$ and have all remaining elements less than n_{k-i+1}. All of these lists comes before L in the order, and so summing we get the result. □

Proposition. *Let* $\{n_1, \ldots, n_k\} \subseteq \{1, 2, \ldots, n\}$ *with* $n_1 < n_2 < \cdots < n_k$.
Let $L = (n_1, n_2, \ldots, n_k)$ *and* $\tilde{L} = (n+1-n_1, n+1-n_2, \ldots, n+1-n_k)$. *Then*

$$\textit{rank}(L) + \textit{corank}(\tilde{L}) = \binom{n}{k} - 1$$

Proof. Suppose $(n_1, n_2, \ldots, n_k) < (m_1, m_2, \ldots, m_k)$ in lexicographic order, then $(n+1-n_1, \ldots, n+1-n_k) > (n+1-m_1, \ldots, n+1-m_k)$ in lexicographic order. So given L, for every other k-subset M either $L > M$ or $\tilde{L} > \tilde{M}$. Therefore $\text{rank}(L) + \text{corank}(\tilde{L})$ equals the number of k-subsets other than L itself. □

From this we can build

Algorithm: Rank2kSubset

```
input: L (current k subset as a list in increasing order), k, n
r = 0
L(0) = 0
for i from 1 to k
  r = r + binom(n-L(i), k+1-i)
output: binom(n,k) - 1 - r
```

This algorithm is $O(k)$.

Exercise. Can you make a similar `Unrank2kSubset`? What is its runtime?