skewed : tree size under sweepstakes reproduction

Bjarki Eldon[1][2]

Abstract

This code estimates the total tree size of a sample of a given size up to and including the total population. The population is finite and haploid and evolves according to a model of sweepstakes reproduction. The code is part of joint work with Jonathan A. Chetwyn-Diggle and Alison Etheridge about trying to understand what happens to genealogies when sample size becomes 'large' (1).

Contents

[1]MfN Berlin, Germany
September 11, 2021

[2]Supported by Deutsche Forschungsgemeinschaft (DFG) - Projektnummer 273887127 through DFG SPP 1819: Rapid Evolutionary Adaptation grant STE 325/17-2 to Wolfgang Stephan; acknowledge funding by Icelandic Centre of Research through an Icelandic Research Fund Grant of Excellence no. 185151-051 to Einar Árnason, Katrín Halldórsdóttir, Wolfgang Stephan, AME, and BE.

1 Copyright

2 compilation and execution

⟨SEC:comp⟩ Use `ctangle` to generate a C++ file ending in `.c`, and `cweave` to generate a LaTeX file ending in `.tex` for compilin the documentation. Compiling the C++ code:

```
c++ -Wall -Wextra -pedantic -std=c++20 -m64 -Og -march=native poptreeskewed.c
-lm -lgsl -lgslcblas
```

The code passes `valgrind` checks for memory leaks, and a basic `cppcheck`.

Call the compiled code with

```
./a.out $(shuf -i 1000-100000 -n1)
```

3 intro

?⟨SEC:intro⟩?

Consider a haploid population evolving according to sweepstakes reproduction. In any given generation the current N individuals independently produce juveniles (potential offspring) according to a given law. From the total pool of at least N juveniles we sample N juveniles independently and uniformly at random without replacement to form the next set of reproducing individuals. The random number of juveniles is given the distribution

$$\mathbb{P}\left(X_1 = k\right) = \mathbb{1}_{\{1 \le k \le u(N)\}} \left(\frac{1}{k^\alpha} - \frac{1}{(1+k)^\alpha} \right) \frac{1}{1 - \frac{1}{(1+u(N))^\alpha}}, \qquad (1)$$ eq:1

where $u(N) \in \{1, 2, \ldots\}$ is an upper bound on the number of juveniles. Assuming an upper bound on the number of juveniles is biologically reasonable. We assume $1 < \alpha < 2$.

We are interested in the tree size $B^N(N)$ of the whole population measured in generations. We would like to see if we can identify a scaling $f(N)$ so that $B^N(N)/f(N)$ converges (ideally almost surely, but in L_1 would do), and we are hoping simulating this could give us an idea of the scaling (although for sure can be misleading).

4 code

?⟨SEC:code⟩?

The sampling is individual-based, meaning we consider a finite haploid population, traverse the tree backwards-in-time generation by generation, and measure the tree size in generations. The code is simple enough, unfortunately however, the sampling becomes damn slow for $N \geq 10^6$. The following sections describe each small part of the code, each is self-explanatory.

4.1 GSL random number generator

⟨SEC:gslrng⟩

We will use the GSL random number generators for drawing a random uniform from the unit interval, and in the hypergeometric sampling.

5 ⟨ GSL rng 5 ⟩ ≡

```
gsl_rng * rngtype;
static void setup_rng(unsigned long int s)
{
  const gsl_rng_type*T;
  gsl_rng_env_setup( );
  T = gsl_rng_default;
  rngtype = gsl_rng_alloc(T);
  gsl_rng_set(rngtype, s);
}
```

This code is used in chunk 13.

4.2 constants

?⟨SEC:const⟩?

For convenience we define here the key data, the α parameter `CONST_ALPHA` and the cut-off $u(N)$ `CONST_CUTOFF` of the juvenile distribution Eq (1), and the population size N `CONST_POP_SIZE`. The code is configured to estimate the tree size of the whole population, however any sample size works.

6 ⟨constants 6⟩ ≡

```
const double CONST_ALPHA = 1.05;
const size_t CONST_POP_SIZE = 1 · 10^1;
const double CONST_CUTOFF = 1. · 10^1;
```

This code is used in chunk 13.

4.3 includes

⟨SEC:includes⟩

Probably don't need all the libraries.

7 ⟨includes 7⟩ ≡

```
#include <iostream>
#include <vector>
#include <random>
#include <functional>
#include <memory>
#include <utility>
#include <algorithm>
#include <ctime>
#include <cstdlib>
#include <cmath>
#include <string>
#include <fstream>
#include <chrono>
#include <assert.h>
#include <math.h>
#include <unistd.h>
#include <omp.h>
#include <stdio.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
```

This code is used in chunk 13.

4.4 compute the cumulative density function

⟨SEC:cdf⟩
Compute the cumulative density function for sampling a number of juveniles using the mass function Eq (1).

8 ⟨cdf 8⟩ ≡

```
static void mass_function ( std::vector < double > &vcdf )
{      /* set probability of zero juveniles to zero vcdf[0] = 0.0
          */
  vcdf.push_back(0.0);
  for (double j = 1; j ≤ CONST_CUTOFF; ++j) {
        /* set vcdf[j] to vcdf[j − 1] +ℙ (X_1 = j) Eq (1)
          */
    vcdf.push_back(vcdf.back() + (pow(1.0/j, CONST_ALPHA) − pow(1.0/(1. + j),
        CONST_ALPHA))/(1. − pow(1./(CONST_CUTOFF + 1.), CONST_ALPHA)));
  }
}
```

This code is used in chunk 13.

4.5 sample one number of juveniles

C:samplejuvs⟩

Sample one realisation of number of juveniles by the inverse-CDF method using the CDF computed by *mass_function* § 4.4

9 ⟨ sample number juveniles 9 ⟩ ≡

```
static size_t sample_juveniles ( const std::vector < double > &vcdf , gsl_rng*r )
{
  size_t j = 1;
  const double u = gsl_rng_uniform(r);
  while (vcdf[j] < u) {
    ++j;
  }
  return (j);
}
```

This code is used in chunk 13.

4.6 sample a pool of juveniles

⟨SEC:pool⟩
Sample a pool of juveniles using *sample_juveniles* § 4.5

10 ⟨ pool 10 ⟩ ≡

```
static size_t sample_pool_juveniles ( std::vector < size_t > &pool_juvs,  const std::vector
        < double > &v_cdf, gsl_rng*r )
   {
     pool_juvs.clear( );
     size_t s = 0;
     for (size_t i = 0; i < CONST_POP_SIZE; ++i) {
       pool_juvs.push_back(sample_juveniles(v_cdf, r));
       s += pool_juvs.back( );
     }     /* return the total number of juveniles; since $\mathbb{P}(X_1 = 0) = 0$ we always have
           at least $N$ juveniles
           */
     return (s);
   }
```

This code is used in chunk 13.

4.7 compute new number of lines

⟨SEC:newlines⟩
Compute new number of lines by assigning current lines to families uniformly at random and without replacement; i.e. the joint distribution of number of lines per family is multivariate hypergeometric. We sample the marginals, and only need to record the number of families with at least one line. Conditional on $(X_1, \ldots, X_N) = (x_1, \ldots, x_N)$ and m current number of lines

$$\mathbb{P}\left((v_1, \ldots, v_N) = (m_1, \ldots, m_N)\right) = \frac{\binom{x_1}{m_1}\cdots\binom{x_N}{m_N}}{\binom{\sum_i x_i}{m}}. \quad (2) \boxed{\texttt{eq:2}}$$

where $m_1 + \cdots + m_N = m$. Then the new number of lines is

$$m' = \sum_i \mathbb{1}_{\{m_i > 0\}}. \quad (3) \boxed{\texttt{eq:3}}$$

We sample the marginals, meaning we sample consecutively v_1, v_2, etc. updating the number of remaining lines until all the current lines have been assigned a family, or we have sampled v_{N-1}, in which case the remaining lines are assigned to v_N.

11 ⟨ new number lines 11 ⟩ ≡

```
static size_t r_new_number_lines (size_t k, const std::vector < size_t > &v_juvs,  const
        size_t SN, gsl_rng*r )
    {     /* k is the current number of lines */
      size_t new_number_lines = 0;
      size_t n_others = SN − v_juvs[0];     /* sample $v_1$ from the marginal of Eq (2)
        */
      size_t x = gsl_ran_hypergeometric(r, v_juvs[0], n_others, k);
        /* update the new number of lines according to Eq (3)
        */
      new_number_lines += (x > 0 ? 1 : 0);
      k −= x;
      size_t i = 0;
      while ((k > 0) ∧ (i < CONST_POP_SIZE − 1)) {
```

```
    ++i;      /* we can stop as soon as all lines have been assigned to a family
        */
    n_others −= v_juvs[i];
    x = gsl_ran_hypergeometric(r, v_juvs[i], n_others, k);
    new_number_lines += (x > 0 ? 1 : 0);
    k −= x;
  }
  new_number_lines += (k > 0 ? 1 : 0);      /* return m′ in Eq (3)
      */
  return (new_number_lines);
}
```

This code is used in chunk 13.

4.8 estimate the tree size

imatetreesize⟩

Estimate the tree size

```
12 ⟨ estimatesize 12 ⟩ ≡

  static void tree_size(gsl_rng * r) {
          /* initialise the container for holding the CDF from Eq (1)
           */
      std::vector < double > v_cdf;
      v_cdf.reserve(static_cast⟨size_t⟩(CONST_CUTOFF) + 1);
          /* compute the CDF from Eq (1) § 4.4
           */
      mass_function(v_cdf);     /* initialise the container for holding realisations of
           $X_1, \ldots, X_N$ the random number of juveniles
           */
      std::vector < size_t > v_pjuvs;
      v_pjuvs.reserve(CONST_POP_SIZE);     /* initialise the local variables
           */
      size_t current_number_lines = CONST_POP_SIZE;
      size_t SN = 0;
      size_t tree_size = 0;
      while (current_number_lines > 1) {
        tree_size += current_number_lines;     /* sample pool of juveniles § 4.6
             */
        SN = sample_pool_juveniles(v_pjuvs, v_cdf, r);
            /* compute the new number of lines § 4.7
             */
        current_number_lines = r_new_number_lines(current_number_lines, v_pjuvs, SN,
             rngtype);
      }
```

```
        printf("%lu\n", tree_size); }
```

This code is used in chunk 13.

```
        /* § 4.3
        */
  ⟨ includes 7 ⟩      /* § 4.1
        */
  ⟨ GSL rng 5 ⟩
  ⟨ constants 6 ⟩      /* § 4.4
        */
  ⟨ cdf 8 ⟩      /* § 4.5
        */
  ⟨ sample number juveniles 9 ⟩      /* § 4.6
        */
  ⟨ pool 10 ⟩      /* § 4.7
        */
  ⟨ new number lines 11 ⟩      /* § 4.8
        */
  ⟨ estimatesize 12 ⟩
        int main(int argc, char *argv[ ])
        {      /* estimate the tree size § 4.8
                */      /* initialise the GSL random number generator § 4.1
                */
          setup_rng(static_cast⟨unsigned long⟩(atol(argv[1])));
          tree_size(rngtype);
          gsl_rng_free(rngtype);
          return (0);      /* at end of main function
                */      /* See § 2 for compiling and running.
                */
```

```
}
```

5 references

?⟨SEC:refs⟩?

References

n-diggle_beta [1] JA Chetwyn-Diggle, Bjarki Eldon, and Alison M Etheridge. Beta-coalescents when sample size is large. no idea.

Index

List of Refinements

⟨ GSL rng 5 ⟩ Used in chunk 13.

⟨ cdf 8 ⟩ Used in chunk 13.

⟨ constants 6 ⟩ Used in chunk 13.

⟨ estimatesize 12 ⟩ Used in chunk 13.

⟨ includes 7 ⟩ Used in chunk 13.

⟨ new number lines 11 ⟩ Used in chunk 13.

⟨ pool 10 ⟩ Used in chunk 13.

⟨ sample number juveniles 9 ⟩ Used in chunk 13.