Gene genealogies in a haploid population evolving according to sweepstakes reproduction – approximating $\mathbb{E}[R_i(n)]$ for the time-changed δ_0-Poisson-Dirichlet$(\alpha, 0)$-coalescent

Bjarki Eldon[1][2]

Fix $0 < \alpha < 2$ and $0 < \gamma \leq 1$. Let $\{\xi^n\} \equiv \{\xi^n(t) : t \geq 0\}$ be the continuous-time time-changed δ_0-Poisson-Dirichlet$(\alpha, 0)$-coalescent with time-change function $G(t) = \int_0^t e^{\rho s} ds$. Write $\#A$ for the number of elements in a given finite set A, $L_i(n) \equiv \int_0^{\tau(n)} \# \{\xi \in \xi^n(t) : \#\xi = i\}\, dt$ and $L(n) \equiv \int_0^{\tau(n)} \#\xi^n(t)dt$ and $\tau(n) \equiv \inf\{t \geq 0 : \#\xi^n(t) = 1\}$ for $i \in \{1, 2, \ldots, n-1\}$. We then have $L(n) = L_1(n) + \cdots + L_{n-1}(n)$. Define $R_i(n) \equiv L_i(n)/\sum_j L_j(n)$ for $i = 1, 2, \ldots, n-1$. Interpreting $\{\xi^n\}$ as 'trees' we may view $L_i(n)$ as the random total length of branches supporting $i \in \{1, 2, \ldots, n-1\}$ leaves, with the length measured in coalescent time units, and n sample size. With this C++ code we sample the time-changed δ_0-Poisson-Dirichlet$(\alpha, 0)$-coalescent with $0 < \alpha < 2$ and exponential growth parameter $\rho \geq 0$ fixed, and approximate $\mathbb{E}[R_i(n)]$

Contents

1 Copyright

[1] beldon11@gmail.com

[2] compiled @ 3:57pm on Tuesday 6$^{\text{th}}$ January, 2026; using
kernel 6.18.3+deb14-amd64 GNU/Linux
GNU bash, version 5.3.3(1)-release (x86_64-pc-linux-gnu)
CTANGLE 4.12.1 (TeX Live 2025/Debian)
g++ (Debian 15.2.0-12) 15.2.0
GSL 2.8
CWEAVE 4.12.1 (TeX Live 2025/Debian)
This is LuaHBTeX, Version 1.22.0 (TeX Live 2025/Debian) Development id: 7673
SpiX 1.3.0
GNU Awk 5.3.2, API 4.0, PMA Avon 8-g1, (GNU MPFR 4.2.2, GNU MP 6.3.0)
GNU Emacs 30.2

2 compilation and output

This CWEB [KL94] document (the `.w` file) can be compiled with `cweave` to generate a `.tex` file, and with `ctangle` to generate a `.c` [KR88] C++ code file.

Use the shell tool `spix` on the script appearing before the preamble (the lines starting with %$); simply

```
spix /path/to/the/sourcefile
```

where `sourcefile` is the `.w` file

One may also copy the script into a file and run `parallel` [Tan11] :

```
parallel --gnu -j1 :::: /path/to/scriptfile
```

3 intro

We consider the continuous-time time-changed δ_0-Poisson-Dirichlet$(\alpha, 0)$-coalescent, with $A \equiv 1/\left(\prod_{j=2}^{n}\left(\sum_i \mathbb{1}_{\{k_i=j\}}\right)!\right)$, transition rates

$$\lambda_{n;k_1,\ldots,k_r;s} = A\binom{n}{k_1 \ldots k_r\ s}\left(\mathbb{1}_{\{r=1,k_1=2\}}\frac{C_\kappa}{C_\kappa + c(1-\alpha)} + \frac{cp_{n;k_1,\ldots,k_r;s}}{C_\kappa + c(1-\alpha)}\right) \tag{1}$$

where $0 < \alpha < 1$, $\kappa \geq 2$, $c \geq 0$ all fixed, and

$$\begin{aligned} p_{n;k_1,\ldots,k_r;s} &= \frac{\alpha^{r+s-1}\Gamma(r+s)}{\Gamma(n)}\prod_{i=1}^{r}(k_i - \alpha - 1)_{k_i-1} \\ C_\kappa &= \mathbb{1}_{\{\kappa=2\}}\frac{2}{m_\infty^2} + \mathbb{1}_{\{\kappa>2\}}\frac{c_\kappa}{2^\kappa(\kappa-2)(\kappa-1)} \end{aligned} \tag{2}$$

and $\kappa + 2 < c_\kappa < \kappa^2$ for $\kappa > 2$.

4 code

¶ **includes**.
the included libraries

5 ⟨includes 5⟩ ≡

```
#include <iostream>
#include <cstdlib>
#include <iterator>
#include <random>
#include <fstream>
#include <iomanip>
#include <vector>
#include <numeric>
#include <functional>
#include <algorithm>
#include <cmath>
#include <unordered_map>
#include <assert.h>
#include <float.h>
#include <fenv.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_math.h>
#include "headerfile.hpp"
```

This code is used in chunk 17.

¶ **the random number generators**.
the random number generators

6 ⟨rngs 6⟩ ≡

```
std::random_device randomseed;    /*
    Standard mersenne twister random number engine */
std::mt19937_64 rng(randomseed());
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 17.

4.1 the coalescent rate

the coalescent rate $\lambda_{n;k_1,\ldots,k_r;s}$

7 $\langle\, \lambda_{n;k_1,\ldots,k_r;s}$ 7 $\rangle \equiv$

```
static double lambdanks (const double n, const std::vector < unsigned int > &v_k ) {
  assert(v_k[0] > 1);
  double d
  { }
  ;
  double k
  { }
  ;
  double f
  {1};
  const double r = static_cast⟨double⟩(v_k.size( )); std::unordered_map < unsigned int ,
          unsigned int > counts
  { }
  ;
  const auto descending_factorial = [ ](const double x, const double m)
  {
    double p = 1;
    for (double i = 0; i < m; ++i) {
      p *= (x − i);
    }
    return p;
  }
  ;
  const auto veldi = [ ](const double x, const double y)
  {
    feclearexcept(FE_ALL_EXCEPT);
    const double d = pow(x, y);
    return (fetestexcept(FE_UNDERFLOW) ? 0. : (fetestexcept(FE_OVERFLOW) ? FLT_MAX : d));
  }
  ;
  for (std::size_t i = 0; i < v_k.size( ); ++i) {
    f *= descending_factorial(static_cast⟨double⟩(v_k[i]) − 1. − ALPHA,
        static_cast⟨double⟩(v_k[i]) − 1);      /*
         count occurrence of each merger size */
    ++counts[v_k[i]];
    k += static_cast⟨double⟩(v_k[i]);
    d += lgamma(static_cast⟨double⟩(v_k[i] + 1));
  }
  assert(k < n + 1);
  const double s = n − k;
  const double p = static_cast⟨double⟩ ( std::accumulate (counts.begin( ), counts.end( ), 0,
          [ ](double a, const auto &x)
  {
    return a + lgamma((double) x.second + 1);
  }
  ) ) ;
```

const double $l = ((v\_k.size(\,) < 2\ ?\ (v\_k[0] < 3\ ?\ 1.:0):0) * \texttt{CKAPPA}) + (\texttt{CEPS} * veldi(\texttt{ALPHA},$
$r + s - 1) * tgamma(r + s) * f / tgamma(n));$

return $(veldi(exp(1),$
$(lgamma(n + 1.) - d) - lgamma(n - k + 1) - p) * l / (\texttt{CKAPPA} + (\texttt{CEPS} * (1 - \texttt{ALPHA})))); \}$

This code is used in chunk 17.

4.2 partitions summing to a given number

generate the partitions of a given size summing to a given number *myInt*

8 ⟨GenPartitions 8⟩ ≡

```
static double GenPartitions (const unsigned int m, const unsigned int myInt, const
      unsigned int PartitionSize, unsigned int MinVal, unsigned int MaxVal,
      std :: vector < std :: pair < double , std :: vector < unsigned int >>> &v_l_k, std :: vector
      < double > &lrates_sorting ) {      /*
       m is the given number of blocks; PartitionSize is the size of the partitions; the
      partitions sum to myInt */
double lrate
{ }
;
double sumrates
{ }
; std :: vector < unsigned int > partition(PartitionSize);
unsigned int idx_Last = PartitionSize − 1;
unsigned int idx_Dec = idx_Last;
unsigned int idx_Spill = 0;
unsigned int idx_SpillPrev;
unsigned int LeftRemain = myInt − MaxVal − (idx_Dec − 1) ∗ MinVal;
partition[idx_Dec] = MaxVal + 1;
do {
  unsigned int val_Dec = partition[idx_Dec] − 1;
  partition[idx_Dec] = val_Dec;
  idx_SpillPrev = idx_Spill;
  idx_Spill = idx_Dec − 1;
  while (LeftRemain > val_Dec) {
    partition[idx_Spill−−] = val_Dec;
    LeftRemain −= val_Dec − MinVal;
  }
  partition[idx_Spill] = LeftRemain;
  const char a = (idx_Spill) ? ∼((−3 ≫ (LeftRemain − MinVal)) ≪ 2) : 11;
  const char b = (−3 ≫ (val_Dec − LeftRemain));
  switch (a & b) {
  case 1: case 2: case 3: idx_Dec = idx_Spill;
    LeftRemain = 1 + (idx_Spill − idx_Dec + 1) ∗ MinVal;
    break;
  case 5:
    for (++idx_Dec, LeftRemain = (idx_Dec − idx_Spill) ∗ val_Dec;
          (idx_Dec ≤ idx_Last) ∧ (partition[idx_Dec] ≤ MinVal); idx_Dec++)
      LeftRemain += partition[idx_Dec];
    LeftRemain += 1 + (idx_Spill − idx_Dec + 1) ∗ MinVal;
    break;
  case 6: case 7: case 11: idx_Dec = idx_Spill + 1;
    LeftRemain += 1 + (idx_Spill − idx_Dec + 1) ∗ MinVal;
    break;
  case 9:
    for (++idx_Dec, LeftRemain = idx_Dec ∗ val_Dec;
          (idx_Dec ≤ idx_Last) ∧ (partition[idx_Dec] ≤ (val_Dec + 1));
          idx_Dec++) LeftRemain += partition[idx_Dec];
    LeftRemain += 1 − (idx_Dec − 1) ∗ MinVal;
```

```
      break;
    case 10:
      for (LeftRemain += idx_Spill * MinVal + (idx_Dec − idx_Spill) * val_Dec + 1, ++idx_Dec;
            (idx_Dec ≤ idx_Last) ∧ (partition[idx_Dec] ≤ (val_Dec − 1));  idx_Dec++)
        LeftRemain += partition[idx_Dec];
      LeftRemain −= (idx_Dec − 1) * MinVal;
      break;
    }
    while (idx_Spill > idx_SpillPrev)  partition[−−idx_Spill] = MinVal;
    assert(static_cast⟨unsigned int⟩(std::accumulate(partition.begin( ), partition.end( ),
        0)) ≡ myInt);       /*
        § 4.1  */
    lrate = lambdanks(static_cast⟨double⟩(m), partition);
    assert(lrate ≥ 0);
    v_l_k.push_back(std::make_pair(lrate, partition));
    lrates_sorting.push_back(lrate);
    sumrates += lrate;
  } while (idx_Dec ≤ idx_Last);
  assert(sumrates ≥ 0);
  return sumrates; }
```
This code is used in chunk 17.

4.3 all partitions summing to a given number

get all partitions (merger sizes) summing to a given number

9 ⟨allmergers_sum_m 9⟩ ≡

```
static double allmergers_sum_m (const unsigned int n, const unsigned
      int m, std::vector < std::pair < double , std::vector < unsigned
      int >>> &v___l_k, std::vector < double > &v_lrates_sort ) {      /*
       n is the number of blocks; the partitions sum to m */
const std::vector < unsigned int > v___m
{m};      /*
      § 4.1 */
double sumr = lambdanks(static_cast⟨double⟩(n), v___m);
v___l_k.push_back(std::make_pair(sumr, v___m));
v_lrates_sort.push_back(sumr);
if (m > 3) {
  for (unsigned int s = 2; s ≤ m/2; ++s) {
    assert(m > 2 ∗ (s − 1));      /*
        § 4.2 */
    sumr += GenPartitions(n, m, s, 2, m − (2 ∗ (s − 1)), v___l_k, v_lrates_sort);
  }
}
assert(sumr ≥ 0);
return sumr; }
```

This code is used in chunk 17.

¶ **write mergers to file.**

10 ⟨ratesmergersfile 10⟩ ≡

```
static void ratesmergersfile (const unsigned int n, const std::vector < unsigned
            int > &v___indx, const std::vector < std::pair < double , std::vector <
            unsigned int >>> &vlk, const double s, std::vector < std::vector <
            double >> &a___cmf ) {
        assert(s > 0);
        double cmf
        { }
        ;
        std::ofstream f;
        f.open("gg_" + std::to_string(n) + "_.txt", std::ios::app);
        a___cmf[n].clear( );
        for (const auto &i:v___indx) {
        cmf += (vlk[i].first)/s;
        assert(cmf ≥ 0);
        a___cmf[n].push_back(cmf);
        assert((vlk[i].second).size( ) > 0);
        for (const auto &x:vlk[i].second)
        {
          f ≪ x ≪ ' ';
        }
        f ≪ '\n'; }
        f.close( );
```

$$assert(abs(cmf - 1.) < 0.999999); \}$$

This code is used in chunk 17.

4.4 get all partitions

11 ⟨allmergers_when_n_blocks 11⟩ ≡

```
static void allmergers_when_n_blocks (const unsigned int n, std::vector <
        double > &v___lambdan, std::vector < std::vector < double >> &a___cmf ) {
        std::vector < std::pair < double , std::vector < unsigned int >>> vlk
{ }
; std::vector < double > ratetosort
{ }
;
ratetosort.clear( );
double lambdan
{ }
;
vlk.clear( );
assert(n > 1);
for (unsigned int k = 2; k ≤ n; ++k) {    /*
        the partition sums to k; the number of blocks is n */
  lambdan += allmergers_sum_m(n, k, vlk, ratetosort);
}    /*
        record the total rate when n blocks */    /*
        use for sampling time */
assert(lambdan > 0);
v___lambdan[n] = lambdan;
std::vector < unsigned int > indx(ratetosort.size( ));
std::iota(indx.begin( ), indx.end( ), 0);
std::stable_sort (indx.begin( ), indx.end( ), [&ratetosort](const unsigned int x, const
        unsigned int y)
{
  return ratetosort[x] > ratetosort[y];
}
) ;    /*
      merger rates sorted in descending order; print the cmf and rates to file */
ratesmergersfile(n, indx, vlk, v___lambdan[n], a___cmf); }
```

This code is used in chunk 17.

¶ **get all partitions.**
generate all mergers up to sample size

12 ⟨all possible partitions 12⟩ ≡

```
static void allmergers ( std::vector < double > &vlmn, std::vector < std::vector <
    double >> &acmf )
{
  for (unsigned int tmpn = 2; tmpn ≤ SAMPLE_SIZE; ++tmpn) {    /*
        § 4.4 */
    allmergers_when_n_blocks(tmpn, vlmn, acmf);
  }
}
```

This code is used in chunk 17.

4.5 tree modules

13 ⟨ updatetree 13 ⟩ ≡

```
static void updatetree ( std::vector < unsigned int > &tre, const std::vector < unsigned
        int > &mergersizes ) {
    assert(mergersizes.size( ) > 0); std::vector < unsigned int > newblocks
    { }
    ;
    newblocks.clear( );
    std::shuffle(tre.begin( ), tre.end( ), rng);
    std::size_t s = tre.size( ); for (const auto &k:mergersizes)
    {
      assert(k > 1);
      assert(k ≤ s);
      s −= k;     /*
           record the size of the merging blocks */
      newblocks.push_back(std::accumulate(tre.rbegin( ), tre.rbegin( ) + k, 0));     /*
           remove the blocks that merged */
      tre.resize(s);
    }
    assert(newblocks.size( ) > 0);
    assert(static_cast⟨unsigned int⟩(std::accumulate(newblocks.begin( ), newblocks.end( ),
        0)) ≤ SAMPLE_SIZE);
    tre.insert(tre.end( ), newblocks.begin( ), newblocks.end( ));
    assert(static_cast⟨unsigned int⟩(std::accumulate(tre.begin( ), tre.end( ),
        0)) ≡ SAMPLE_SIZE); }
```

This code is used in chunk 17.

¶ **read partitions.**

14 ⟨ readmergersizes 14 ⟩ ≡

```
static void readmergersizes (const unsigned int n, const unsigned int j, std::vector <
    unsigned int > &v___mergers ) {
std::ifstream f("gg_" + std::to_string(n) + "_.txt");
std::string line { }
;
v___mergers.clear( );
for (unsigned int i = 0; std::getline (f, line ) ∧ i < j; ++i ) { if (i ≥ j − 1) {
std::stringstream ss ( line ) ;
v___mergers = std::vector < unsigned int > ( std::istream_iterator < unsigned int > (ss),
{ } ) ; } }
assert(v___mergers.size( ) > 0);
assert(v___mergers[0] > 1);
assert(v___mergers.back( ) > 1);
f.close( ); }
```

This code is used in chunk 17.

4.6 one experiment

15 ⟨ onexperiment 15 ⟩ ≡

```
static void onexperiment ( std::vector < double > &v___ri, std::vector < double > &vl,
        const std::vector < double > &v___lambda___n, const std::vector <
        std::vector < double >> &a___cmf ) {
    std::vector < unsigned int > v___tre(SAMPLE_SIZE, 1);
    std::fill(vl.begin( ), vl.end( ), 0);
    unsigned int lina
    { }
    ;
    double tau
    { }
    ;
    double new_time
    { }
    ; std::vector < unsigned int > v_merger_sizes(SAMPLE_SIZE/2);
    v_merger_sizes.reserve(SAMPLE_SIZE/2); auto newtime = [ ](const double
            lambdam, const double tau)
    {
      return (RHO > DBL_EPSILON ? log1p(−(RHO ∗ exp(−RHO ∗ tau)/lambdam) ∗ log(1. −
          gsl_rng_uniform_pos(rngtype)))/RHO : gsl_ran_exponential(rngtype,
          1./lambdam));
    }
    ; auto updatelengths = [&v___tre, &vl](const double t) {
        for (const auto &b:v___tre)
        {
          assert(b > 0);
          assert(b < SAMPLE_SIZE);
          vl[0] += t;
          vl[b] += t;
        }
        } ;
        auto samplemerger = [ ] (const unsigned int n, const std::vector <
            double > &v___cmf )
        {
          unsigned int j
          { }
          ;
          const double u = gsl_rng_uniform(rngtype);
          while (u > v___cmf[j]) {
            ++j;
          }
          return j;
        }
        ;
        while (v___tre.size( ) > 1) {
          new_time = newtime(v___lambda___n[v___tre.size( )], tau);
          tau += new_time;
```

```
    updatelengths(new_time);
    lina = samplemerger(v___tre.size(), a___cmf[v___tre.size()]);
    readmergersizes(v___tre.size(), 1 + lina, v_merger_sizes);      /*
        § 4.5 */
    updatetree(v___tre, v_merger_sizes);
  }
  assert(v___tre.back() ≡ SAMPLE_SIZE);
  assert(vl[0] > DBL_EPSILON);
  const double d = vl[0]; std::transform (vl.cbegin(), vl.cend(), v___ri.begin(),
          v___ri.begin(), [&d](const auto &x, const auto &y)
  {
    return y + (x/d);
  }
  ) ; }
```

This code is used in chunk 17.

¶ **go ahead – approximate** $\mathbb{E}[R_i(n)]$.
approximate $\mathbb{E}[R_i(n)]$

16 ⟨approximate 16⟩ ≡

```
static void approximate() {
    std::vector < double > vri(SAMPLE_SIZE);
    vri.reserve(SAMPLE_SIZE); std::vector < double > v___l(SAMPLE_SIZE);
    v___l.reserve(SAMPLE_SIZE); std::vector < double > v_l_n(SAMPLE_SIZE + 1);
    v_l_n.reserve(SAMPLE_SIZE + 1); std::vector < std::vector < double >> a___cmfs
        (SAMPLE_SIZE + 1, std::vector < double > { } ) ;
    allmergers(v_l_n, a___cmfs);
    int r = EXPERIMENTS + 1;
    while (−−r > 0) {      /*
          § 4.6 */
      onexperiment(vri, v___l, v_l_n, a___cmfs);
    }
    double x
    { }
    ;
    for (unsigned int i = 1; i < SAMPLE_SIZE; ++i) {
      x = vri[i]/static_cast⟨double⟩(EXPERIMENTS);
      std::cout ≪ log(x) − log1p(−x) ≪ '\n';
    }
    }
```

This code is used in chunk 17.

¶ **main**.
the *main* module

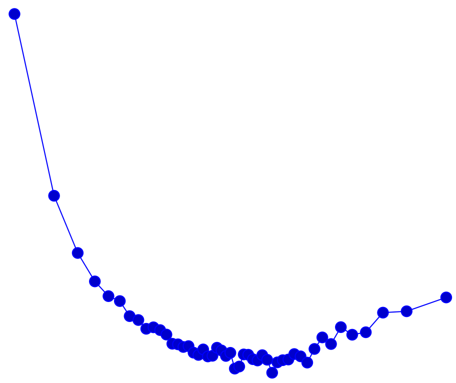
17 ⟨includes 5⟩

⟨ rngs 6 ⟩ /* § 4.1 */
⟨ $\lambda_{n;k_1,\ldots,k_r;s}$ 7 ⟩ /* § 4.2 */
⟨ GenPartitions 8 ⟩ /* § 4.3 */
⟨ allmergers_sum_m 9 ⟩
⟨ ratesmergersfile 10 ⟩ /* § 4.4 */
⟨ allmergers_when_n_blocks 11 ⟩
⟨ all possible partitions 12 ⟩ /* § 4.5 */
⟨ updatetree 13 ⟩
⟨ readmergersizes 14 ⟩ /* § 4.6 */
⟨ onexperiment 15 ⟩
⟨ approximate 16 ⟩

```
int main(int argc, const char *argv[])
{
  setup_rng(static_cast<unsigned long>(atoi(argv[1])));   /* § 4.6 */
  approximate();
  gsl_rng_free(rngtype);
  return GSL_SUCCESS;
}
```

5 conclusions and bibliography

We approximate $\mathbb{E}\left[R_i(n)\right]$ when the coalescent is the time-changed δ_0-Poisson-Dirichlet$(\alpha, 0)$-coalescent. An example in Figure 1



$\log(i/n) - \log(1 - i/n)$

Figure 1: *An example approximation of* $\mathbb{E}\left[R_i(n)\right]$ *for the given parameter values and graphed as logits against* $\log(i/n) - \log(1 - i/n)$ *for* $i = 1, 2, \ldots, n-1$ *where* n *is sample size*

References

[D2024] Diamantidis, Dimitrios and Fan, Wai-Tong (Louis) and Birkner, Matthias and Wakeley, John. Bursts of coalescence within population pedigrees whenever big families occur. Genetics Volume 227, February 2024.
https://dx.doi.org/10.1093/genetics/iyae030.

[CDEH25] JA Chetwyn-Diggle and Bjarki Eldon Beta-coalescents when sample size is large. 2026 bioRxiv https://doi.org/10.64898/2025.12.30.697022

[DK99] P Donnelly and T G Kurtz. Particle representations for measure-valued population models. *Ann Probab*, 27:166–205, 1999.
https://dx.doi.org/10.1214/aop/1022677258

[KL94] Donald Ervin Knuth and Silvio Levy. *The CWEB system of structured documentation: version 3.0.* Addison-Wesley Longman Publishing Co., Inc., Reading, Massachusetts, 1994.

[KR88] Brian W Kernighan and Dennis M Ritchie. The C programming language, 1988.

[Pit99] J Pitman. Coalescents with multiple collisions. *Ann Probab*, 27:1870–1902, 1999.
https://dx.doi.org/10.1214/aop/1022874819

[Sag99] S Sagitov. The general coalescent with asynchronous mergers of ancestral lines. *J Appl Probab*, 36:1116–1125, 1999.
https://doi.org/10.1239/jap/1032374759

[Sch03] J Schweinsberg. Coalescent processes obtained from supercritical Galton-Watson processes. *Stoch Proc Appl*, 106:107–139, 2003.
https://doi.org/10.1016/S0304-4149(03)00028-0

[S00] J Schweinsberg. Coalescents with simultaneous multiple collisions. *Electronic Journal of Probability*, 5:1–50, 2000.
https://dx.doi.org/10.1214/EJP.v5-68

[Tan11] O Tange. GNU parallel – the command-line power tool. The USENIX Magazine, 2011.

Index

List of Refinements