Efficient pedigree recording for fast population genetics simulation

Jerome Kelleher, Kevin R. Thornton, Jaime Ashander, and Peter L. Ralph

November 15, 2017

Note: author order not determined

Abstract

The computational burden of tracking individual genomes in a population genetics simulation can be substantial. In this note we describe how to both (a) dramatically reduce this burden and (b) efficiently record the entire history of the population. We do this by simulating only those loci that may affect reproduction (those having non-neutral variants), and recording the entire history of genetic inheritance in an efficient representation of the ancestral recombination graph, on which neutral mutations can be quickly placed afterwards – the *tree sequence* introduced in the software package `msprime`. We implement the method in two popular fowards-time simulation frameworks, and show it leads to order-of-magnitude speedups.

previous **OUTLINE**, possibly out-of-date but left for comparison.

1. motivate need for whole-genome fwd-time simulations; point out that we only recently have the computing power to do this
2. explain ARG: explain that for forwards-time only need selected loci as by defn all others can be put on afterwards
3. review something about msprime methods for storing/traversing tree sequence
4. describe tables and write out conditions to have valid tables
5. write down algorithm used to do simple WF simulation
6. describe simplify algorithm
7. back-of-the-envelope calculation to compare cost of tracking whole genomes versus putting mutations on ARG
8. comparison of speed with simupop, fwdpp

Introduction

Since the 1980's, coalescent theory has enabled computer simulation of the results of population genetics models identical to that which would be produced by large, randomly mating populations over long periods of time without actually requiring simulation of so many generations or meioses. Coalescent theory thus had three transformative effects on population genetics: first, giving researchers better conceptual tools to describe *gene trees* and thus bringing within-population trees into better focus; second, producing analytical methods to estimate parameters of interest from genetic data (e.g. $\theta = 4N_e\mu$); and finally, providing a computationally feasible method to produce computer simulations of population genetics processes. However, these powerful advances came with substantial caveats: the backwards-in-time processes that are described by coalescent theory are only *Markovian*, and thus feasible to work with, thanks to the important assumptions

of (a) random mating, (b) neutrality, and (c) small sample size relative to the population size. The first two assumptions can be side-stepped to a limited extent, but it remains a challenge to map results of coalescent models onto species that are distributed across continuous geographical space and/or have large numbers of loci under various sorts of selection. Furthermore, the relationship between the life history of a species – fecundity and mortality schedules, allee effects, and demographic fluctuations – are all absorbed into a single compound parameter, the coalescence rate. The third assumption is no longer safe, either – for example, a recent study [Martin et al., 2017] simulated 600,000 samples of human chromosome 20 Several studies have now shown that in samples of size approaching that of the population, genealogical properties may be distorted relative to the coalescent expectation [Wakeley and Takahashi, 2003, Maruvka et al., 2011, Bhaskar et al., 2014]. These considerations, and increasing computational power, have led to a resurgence of interest in forwards-time, individual-based simulations.

Modern computing power easily allows simulations of birth, death and reproduction in a population having even hundreds of millions of individuals, even though theory tells us that a population of size N must be run for many multiples of N generations to produce stable genetic patterns. However, if our interest lies in the resulting genetic patterns of variation – and often, the point of such simulations is to compare to real data – then such simulations must somehow produce at the end data for each individual on a genomic scale. As samples of most species genomes harbor tens or hundreds of millions of variant sites, naively carrying full genotypes for even modest numbers of individuals through a simulation becomes quickly prohibitive.

However, it is thought that most genetic variation is selectively neutral (or nearly so). By definition, the alleles carried by individuals in a population at neutral sites do not affect the population process. For this reason, if one records the entire genealogical history of a population over the course of a simulation, one can lay down neutral mutations on top of that history afterwards, without loss of generality. Precisely, we would need to know the genealogical tree relating all sampled individuals at each position along the genome. In this paper, we show how to use algorithmic tools and data structures developed for the coalescent simulator `msprime` to efficiently record, and later process, this history.

In so doing we record the *population pedigree* – the entire history of parent-offspring relationships of an entire population going back to a remote time – as well as information encoding the genetic outcomes of each ancestral meiosis – who inherited which parts of which parental chromosomes. This embellished pedigree contains all the information necessary to construct the genealogical tree that relates each individual to each other at each position on the genome, i.e., the *tree sequence*. Combined with ancestral genotypes and the origins of new mutations, it also completely specifies the genomic sequence of any individual in the population at any time. This is much more than we need to know, however, so we discard all information irrelevant to the genetic history of the *sampled* individuals, which results in considerable savings. Another way of representing this same information is known as the *ancestral recombination graph*, or ARG [Griffiths and Marjoram, 1997], which has been the subject of substantial study under the assumptions of coalescent theory [Wiuf and Hein, 1997, 1999, Marjoram and Wall, 2006, Wilton et al., 2015].

this paragraph is supposed to cite the fwds time simulation methods we need to Forwards-time population genetics simulations are growing in importance, as the genome-wide impacts of hitchhiking and background selection are more widely appreciated. For instance, Harris and Nielsen [2016] used SLiM [Haller and Messer, 2017] to simulate tens of thousands of human exomes to assess the impact of genetic load and Neanderthal introgression on human genetic diversity. This is roughly state-of-the-art – the fastest simulation frameworks such as SLiM and fwdpp [**?**] can "only" simulate tens of megabases of sequence in tens of thousands of individuals for tens of thousands of generations – but still substantially short of whole genomes for most biological population sizes. There has been some progress at parallelization [Lawrie, 2017], which has substantial promise. The general idea of storing genealogical information to speed up simulations is not new – it was entirely implemented in AnA-FiTS [Aberer and Stamatakis, 2013], for instance – but this program did not implement the critical step of discarding irrelevant genealogical information. A similar but more limited method for discarding this information does appear in Padhukasahasram et al. [2008].

In this paper we discuss storage methods for genealogies (and hence, genome sequence), an algorithm for simplifying these, and their use in forwards-time simulation. Although we were motivated by a need for more efficient simulations, these tools may prove more widely useful.

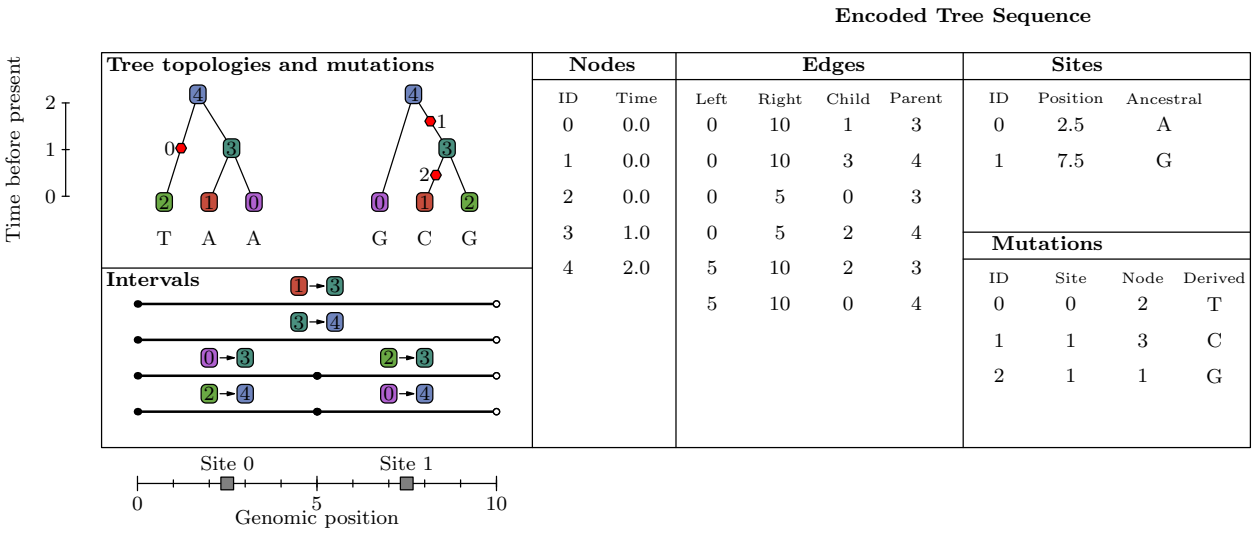


Figure 1: An example tree sequence with three samples over a chromosome of length 10. The left-hand panel shows the tree sequence pictorially in two different ways: (top) as a sequence of tree topologies and (bottom) the spatial extent of the edges that define these topologies. The right-hand panels show the specific encoding of this tree sequence in the four tables (nodes, edges, sites and mutations) defined by `msprime`. *Whoops! genotype of 1 in the second tree should be 'G'! Some minor alignment issues present here. turns out this is no fun to edit in inkscape. I can re-make to match the other figs but will leave alone for now: it's good!*

Results

Below, we first describe the conceptual and algorithmic foundations for the method: (a) the new *tables* format for storing tree sequences; (b) an algorithm to output tree sequence information on the fly from a forwards-time simulation; and (c) a method to *simplify* a tree sequence, i.e., remove redundant information. This is followed by benchmarking comparisons with real, forwards-time simulations.

Recording a tree sequence

A *tree sequence* is an encoding for the sequence of correlated trees, such as those that describe the history of a sexual population. It is efficient because branches that are shared by adjacent trees are stored once, rather than repeatedly for each tree. The topology of a tree sequence is defined via its *nodes* and *edges*, while information about variants are recorded as *sites* and *mutations*. The format for these four tables is given in more detail in the Methods; here we explain conceptually, using the example of Figure 1.

The *nodes* of a tree sequence correspond to the vertices in the individual genealogies along the sequence, and each node refers to a distinct ancestor. Since each node represents a specific ancestor, it has a unique "time", thought of as her birth time, which determines the height of any vertices she is associated with. The example of Figure 1 has five nodes: nodes 0, 1 and 2 occur at time 0 and are the *samples*, while nodes 3 and 4 represent those ancestors necessary to record their genealogy, who were born at time 1.5 and 2.5 respectively.

The *edges* define how nodes relate to each other over specific genomic intervals. Each edge records: the endpoints $[\ell, r)$ of the half-open genomic interval defining the spatial extent of the edge; and the identities p and c of the parent and child nodes of a single branch that occurs in all trees covering this interval. The spatial extent of the edges defining the topology of Figure 1 are shown in the bottom left panel. For example, the branch joining nodes 1 to 3 appears both trees, and so is recorded as a single edge extending over the

whole chromosome. It is this method of capturing the shared structure between adjacent trees that makes the tree sequence encoding very compact and algorithmically efficient.

Recovering the sequence of trees from this information is straightforward: each point along the genome that the tree topology changes is accompanied by the end of some *edges* and the beginning of others. Since each *edge* records the genomic interval over which a given node inherits from a particular ancestor, to construct the tree at a certain point in the genome we need only retrieve all edges overlapping that point and construct the corresponding tree. To modify the tree to reflect the genealogy at a nearby location, we need only remove those edges whose intervals do not overlap that location, and add those new edges whose intervals do. Incidentally, this property that edges naturally encode *differences* between nearby trees (e.g., as "subtree prune and regraft" moves) allows for efficient algorithms that take advantage of the highly correlated nature of nearby trees.

Given the topology defined by the nodes and edges, *sites* and *mutations* encode the sequence information for each sample in an efficient way. Each site is associated with a position on the genome and an ancestral state. For example, in Figure 1 we have two sites, one at position 2 with ancestral state 'A' and the other at position 7 with ancestral state 'G'. If no mutations occur, all samples inherit the ancestral state at a given site. Each mutation occurs above a specific node at a given site, and results in a specific derived state. Thus, all samples below the mutation node in the tree will inherit this state (unless further mutations are encountered). Three mutations are shown in Figure 1, illustrated by red hexagons. The first mutation occurs at site zero (in the left-hand tree), and is a simple mutation resulting in node 2 inheriting the state 'T'. The second site (in the right hand tree) has two mutations: one occurring over node 3 changing the state to 'C', and a back mutation over node 1 changing the state to 'G'.

This encoding of a sequence of trees and accompanying mutational information is very concise. To illustrate this, we ran a simulation of $500,000$ samples of a 200 megabase human-like chromosome ($N_e = 10^4$ and per-base mutation and recombination rates of 10^{-8} per generation) using `msprime`. This resulted in about 1 million distinct marginal trees and 1.1 million infinite-sites mutations. The HDF5 file encoding the nodes, edges, sites and mutations (as described above) for this simulation consumed 157MiB of storage space. Using the `msprime` Python API, the time required to load this file into memory was around 1.5 seconds, and the time required to iterate over all 1 million trees was 2.7 seconds. In contrast, recording the topological information in Newick format would require around 20 TiB and storing the genotype information in VCF would require about 1 TiB – a compression factor of 144,000. Working with either the Newick or VCF encoding of this dataset would likely require several days of CPU time simply to read the information info memory.

Recording the pedigree in forwards time

To record the genealogical history of a forwards in time simulation we need to record two things for each new chromosome: the birth time, and the endpoints and parental IDs of each distinctly inherited segment, which are naturally stored as the *nodes* and *edges* of a tree sequence. For concreteness, here we write out in pseudocode how to run a neutral Wright–Fisher simulation with overlapping generations that records genealogical history in this way. The simulation will run for T generations, and has N haploid individuals, each carrying a single chromosome of length L. For simplicity we sample exactly one crossover per generation. The probability of death per individual each generation is δ. We will use $\mathcal{R}_U(A)$ to denote an element of the set A chosen uniformly at random (and each such instance represents and independent draw).

Algorithm W. (*Forwards-time tree sequence*). Simulates a randomly mating population of N haploid individuals, with a probability of death per generation δ and a chromosome of length L for T generations. On termination, there will be n nodes with times recorded in the vector τ, and E will contain a set of (ℓ, r, p, c) tuples describing the recorded edges.

W1. [Initialisation.] For $0 \leq j < N$, set $P_j \leftarrow j$ and $\tau_j \leftarrow T$. Set $E \leftarrow \emptyset$, $t \leftarrow T$ and $n \leftarrow N$.

W2. [Generation loop.] If $t = 0$ terminate. Set $P' \leftarrow P$ and then set $t \leftarrow t - 1$ and $j \leftarrow 0$.

W3. [Individual loop.] If $j = N$ set $P \leftarrow P'$ and go to W2.

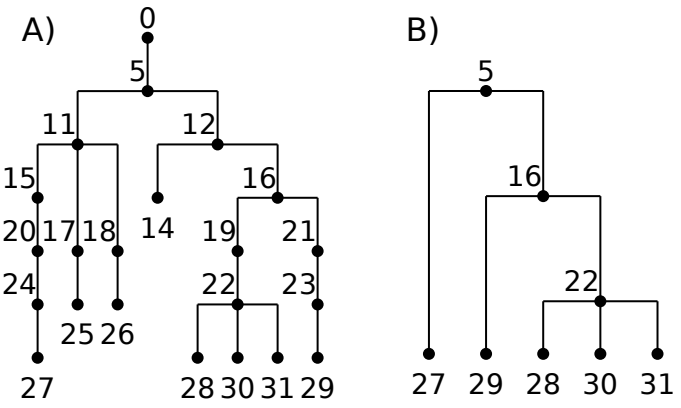


Figure 2: An example of a marginal genealogy from a Wright-Fisher simulation with $N = 5$. **(A)** the original tree including all intermediate nodes and dead-ends, and **(B)** the minimal tree relating all of the currently-alive individuals (27–31).

W4. [Mortality.] If $\mathcal{R}_U([0,1)) \geq \delta$ go to W8.

W5. [New node.] Set $P'_j \leftarrow n$ and $\tau_n \leftarrow t$.

W6. [Choose parents.] Set $a \leftarrow \mathcal{R}_U(\{0,\ldots,N-1\})$, $b \leftarrow \mathcal{R}_U(\{0,\ldots,N-1\})$ and $x \leftarrow \mathcal{R}_U([0,L))$.

W7. [Record edges] Set $E \leftarrow E \cup \{(0,x,P_a,n),(x,L,P_b,n)\}$ and $n \leftarrow n+1$.

W8. [Loop tail] Set $j \leftarrow j+1$ and go to W3.

Algorithm W is a very simple implementation of a Wright–Fisher model intended to illustrate how we can record the nodes and edges of a tree sequence forwards in time. We begin in W1 by allocating our initial population P and creating N nodes with birth time T generations ago, recorded in the vector τ. The set E is used to store the edges that we output during the simulation, and n is the number of nodes created so far (and so, the ID of the next node we create). Steps W2 and W3 simply loop over the generation clock t and individual index j. In W4 we check if an individual P_j has died in this generation. If it has, we replace it in steps W5–W7; if not, we proceed immediately to the next individual. When an individual in the population dies, we first allocate a new node with ID n in W5 and record its birth time. Then, in step W6 we choose two indexes a and b uniformly (giving us parents P_a and P_b) and choose a breakpoint x. We record the effects of this event by storing two new edges: one recording that the parent of node n from 0 to x is P_a, and another recording that the parent of n from x to L is P_b. We then complete the replacement event by incrementing n, ready to represent the next new node.

This algorithm records only the topological information resulting from the forwards in time Wright-Fisher process, but it is straightforward to add mutational information. This can be done in two different ways. We can record mutations that occur during the simulation quite simply. For example, in Algorithm W we would generate mutations after we have recorded the edges joining the new node n to its parents P_a and P_b in step W7. For example, if we assume that mutations occur according to the infinite sites model at rate μ per generation per unit length then there are Poisson $(\mu(r-\ell)(\tau_p - \tau_c))$ mutations on each edge (ℓ, r, p, c). Each of these mutations will occur on a distinct site x drawn uniformly from $(\ell, r]$ and be associated with node c. It is straightforward to record this information during the simulation, but it is significantly simpler and more efficient to generate these mutations after the simulation has completed.

Besides the cost of transmitting genotypes, generating mutations as a separate process after the topological simulation has completed avoids generating the many mutations that are lost in the population. Figure 2 shows an example of a marginal genealogy produced by a forwards-time Wright–Fisher process like Algorithm W. On the left is the tree showing all the edges output by the simulation, while on the right is the minimal tree representing the ancestry of the currently alive population. Clearly there is a great deal of

redundancy in the topological information output by the simulation.

There are two sources of redundancy here. The first type of redundancy arises from nodes in tree that have only one child. In Algorithm W we do not attempt to track coalescence events but simply record all parent-child relationships in the history of the population. As such, many of these edges will record the simple passing of genealogical information from parent to child and only some small subset will correspond to coalesences within the marginal trees. The second source of redundancy in the output of Algorithm W is due to the fact that lineages die out: a large number of individuals in the simulation leave no descendants in the present day population. Node 26 in Figure 2a, for example, leaves no ancestors in the current population, and so the entire path tracing back to the root is redundant.

A minimal genealogy such as in Figure 2 has fewer edges on which to place mutations. There are many more advantages to simplifying the genealogies to this point. Computing this minimal representation of the edges output by a forward-time simulation is an instance of a more general problem that we cover in the next section.

Tree sequence simplification

Suppose that we are only interested in a subset of the nodes (our 'samples') of a tree sequence, and so wish to reduce this input tree sequence into the minimum representation of the topologies that include the specified samples. The output tree sequence must have the following properties:

1. All marginal trees must match the subtree induced by the samples of the corresponding tree in the input tree sequence.

2. Within the marginal trees, all non-sample vertices must has at least two children (i.e., unary tree vertices are removed), unless the vertex is a sample.

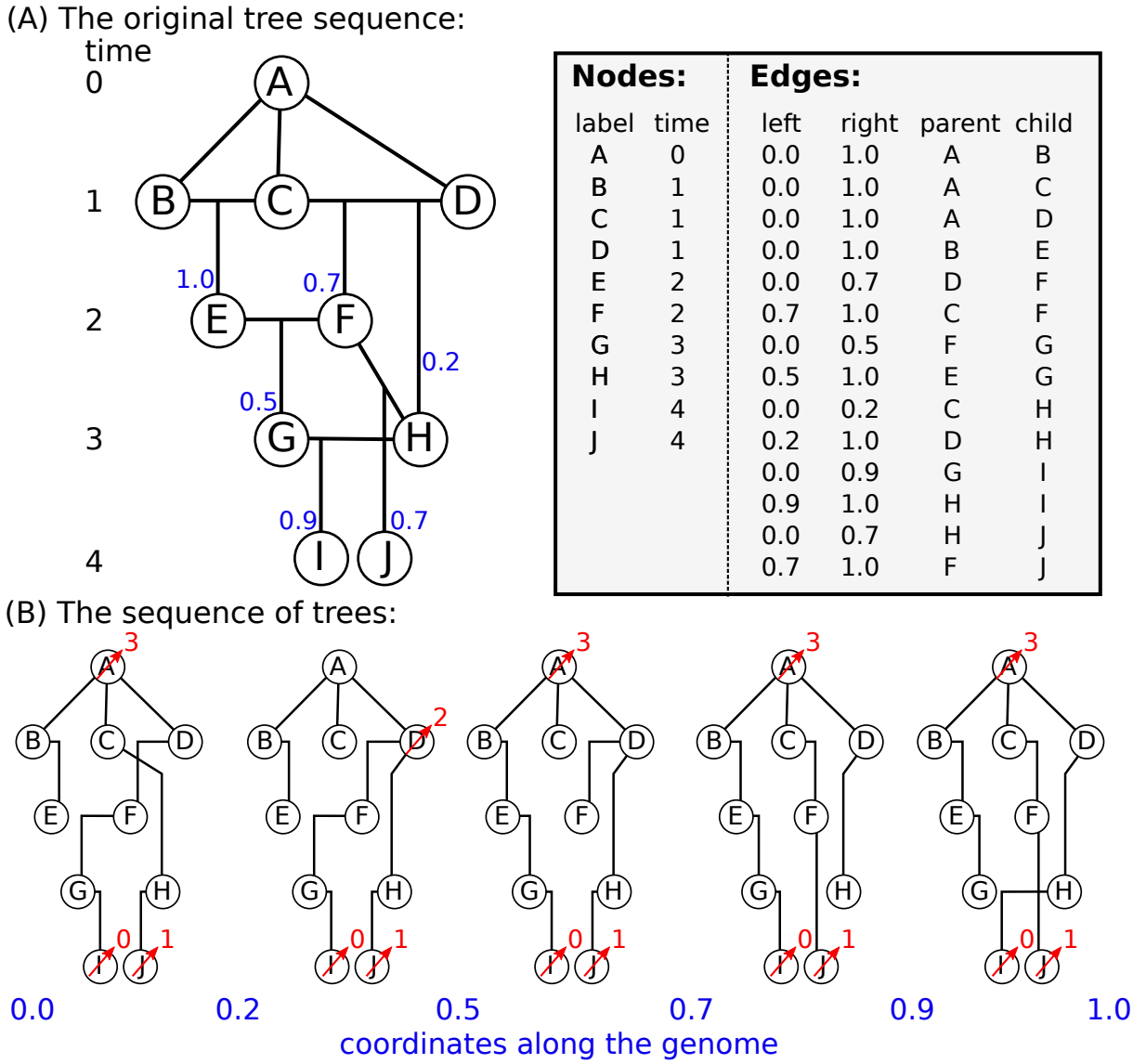
3. Any nodes and edges not ancestral to the sampled nodes are removed.

4. There are no adjacent redundant edges, i.e, pairs of edges (ℓ, x, p, c) and (x, r, p, c) which can be represented with a single edge (ℓ, r, p, c).

The tree sequences produced by forwards simulations, record all of history for everyone alive at any time through the simulation. This is much more than we need to reconstruct the genealogies and sequences of the current population. Simplification is essential to reduce this to a manageable quantity that still contains all the information that we are interested in. Simplification is also useful if we have a large tree sequence representing a large dataset and we wish to extract the information relevant to a subset of the samples.

The approach that we take is based on Hudson's algorithm for simulating the coalescent with recombination [Hudson, 1983, Kelleher et al., 2016]; the implementaiton of simplify parallels the implementation of Hudson's algorithm in `msprime`. Conceptually, this works by (a) beginning by painting the chromosome each sample a distinct color; (b) moving back through history, copying the colors of each chromosome to the portions of its' parental chromosomes from which it was inherited; (c) each time we would paint two colors in the same spot (a coalescence), record that information as an edge and instead paint a brand-new color; and (d) once all colors have coalesced on a given segment, stop propagating it. This "paint pot" description misses some details – for instance, we must ensure that all coalescing segments in a given individual are assigned the *same* new color – but is reasonably close. The process is depicted in Figure 4.

More concretely, the algorithm works by moving back through time, processing each parent in the input tree sequence in chronological order. The main state of the algorithm at each point in time is a set of ancestral lineages, and each lineage is a collection of ancestral segments. An ancestral segment (ℓ, r, u) is found in a lineage if the output node u inherits the genomic interval $[\ell, r)$ from that lineage. (These lineages are the "colors" above.) We also maintain a map A such that A_j is segment chain for node j in the input tree sequence. Crucially, the time required to run the algorithm is linear in the number of edges of the input tree sequence..



Nodes:

label	time
A	0
B	1
C	1
D	1
E	2
F	2
G	3
H	3
I	4
J	4

Edges:

left	right	parent	child
0.0	1.0	A	B
0.0	1.0	A	C
0.0	1.0	A	D
0.0	1.0	B	E
0.0	0.7	D	F
0.7	1.0	C	F
0.0	0.5	F	G
0.5	1.0	E	G
0.0	0.2	C	H
0.2	1.0	D	H
0.0	0.9	G	I
0.9	1.0	H	I
0.0	0.7	H	J
0.7	1.0	F	J

Nodes:

	label	time (ago)
I	0	0
J	1	0
D	2	3
A	3	4

Edges:

left	right	parent	child
0.0	0.2	3	0
0.0	0.2	3	1
0.2	0.5	2	0
0.2	0.5	2	1
0.5	1.0	3	0
0.5	1.0	3	1

Figure 3: A simple example of the method. **Top:** the diagram on the left relates ten haploid individuals to each other. It has 10 node records (one for each individual) and 14 edge records (one for each distinctly inherited segment). Blue numbers denote crossing over locations in each meiosis – for instance, C and D were parents to F, who inherited the left 70% of the chromosome from D and the remainder from C. The individuals B, C, and D inherit clonally from A. **Center:** the five distinct trees relating all individuals to each other found across the chromosome (blue numbers denote locations on the chromosome). Labels after simplification are shown in red. **Bottom:** tables recording the tree sequence after simplification with nodes I and J as samples. The mapping from labels in the forwards time simulation to nodes in the tree sequence is shown in red.

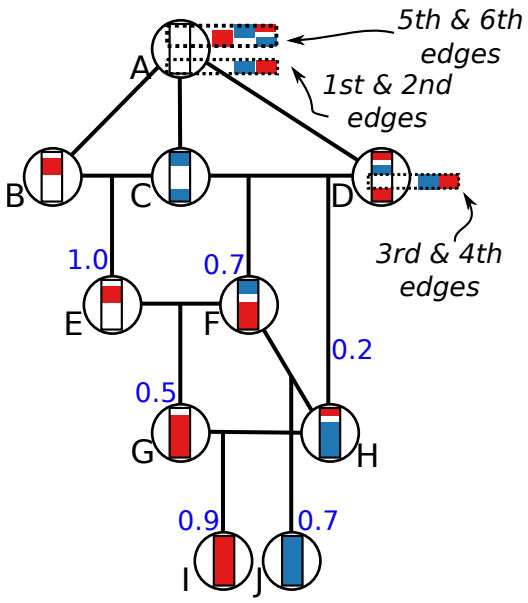


Figure 4: A depiction of the state of the simplification algorithm at each point in time, in the example of Figure 3.

Sequential simplification At any point the simulation scheme above, genealogical history is recorded in a tree sequence. This has two additional advantages. First, simplification can be run periodically through the simulation, taking the set of samples to be the entire currently alive population. This is important as it keeps memory usage from growing linearly (and quickly) with time. Second, the simulation can be *begun* with a tree sequence produced by some other method – for instance, by a coalescent simulation with `msprime`.

Overview of the API

leaving this alone for the moment: to stay here? move?

The `msprime` Python API provides a powerful platform for working with tree topology and mutation data. We refer to the part of `msprime` that is dedicated to tree sequence input and output as the 'Tables API', as the API is organised around simple tables of data. There are four key tables: nodes, edges, sites and mutations. Briefly, nodes and edges define the topology of a tree sequence (as defined above) and the sites and mutations define mutational processes on this topology. Figure 1 gives an informal depiction of this encoding in terms of these tables.

The tables API is primarily designed to facilitate efficient interchange of data between programs or between different modules of the same program. Following the current best-practises [citations: apache arrow, etc] data is stored in a columnar format. There are many advantages to this approach, but the principle advantage for our purposes is that it allows for very efficient interchange of large amounts of numerical data. In principle, this enables zero-copy semantics, where a data consumer can read the information directly from the memory of a producer without incurring the overhead of a copy [citation?] Our implementation uses the numpy C API [citation] to efficiently copy data from Python into the low-level C library used to manipulate tree sequences.

The tables API provides basic input and output operations via the numpy array interface, which provides a great deal of flexibility as well as efficiently. This makes transferring data from various sources such as HDF5 [citation], Dask, Zarr etc, straightforward. (For small scale data and debugging purposes, a simple text format is also supported.) Along with these operations we also provide a function to sort a set of tables

to ensure that the records are in the form required to input into an `msprime` tree sequence object. The `simplify_tables` function implements tree sequence simplification, as described in the previous subsection.

This interchange API is very efficient. [Describe a quick example where we generate a many-gigabyte tree sequence using fwdpp, and the time required to copy the node and edge data into the tables]. By using a simple numerical encoding of tree topologies and contiguous arrays of data to store this encoding, we can achieve data transfer rates that would be impossible under any text-based approach while retaining excellent portability.

Estimates of run-time complexity

Consider a simulation of a Wright-Fisher population of $2N$ haploid individuals using Algorithm W for T generations, with the probability of death, δ, set to 1 for simplicity. Since there is exactly one crossing over every generation, this produces tables of $2NT$ nodes and $4NT$ edges. Suppose that T is large enough that all samples coalesce within the simulation with high probability, so that $T \sim 20N$, say. After simplification, we are left with the tree sequence describing the history of only the current generation of $2N$ individuals. This tree sequence has $4N - 2$ edges to describe the leftmost tree, and one additional edge each time the marginal tree changes along the sequence. Coalescent theory tells us that the expected total length of edges in a marginal tree is approximately $4N \log(2N)$, which is also equal to the mean number of ancestral recombination events that occur on a branch of the marginal tree, since we have taken one crossover per generation. Not all such recombinations actually change the tree topology, and so this gives us an upper bound on the number of edges. Similarly, not every new edge derives from a never-before-seen node, but the number of nodes is at most equal to the number of edges plus the sample size. With $T = 20N$, this reduces the initial storage of $160N^2$ items to $2N(4 \log(2N) + 3)$; at $N = 10^4$, a factor of 20,000.

If we were to add mutations in forwards time under the infinite-sites model with total mutation rate per generation μ, there would also be around $\mu 2NT$ mutations (and the same number of sites). Since mutations fall as recombinations do on the marginal trees, adding them after simplification results in only around $\mu 4N \log(2N)$ mutations. This furthermore avoids the computational burden of propagating mutations forwards through the generations.

Our method stores genealogies, and so records substantially more information than would a method only recording genotypes. However, since the simplification algorithm requires some computational effort, it is still informative to compare computational complexity of the algorithm to one that propagates neutral genotypes. A typical individual will differ at $2N\mu$ sites from the population's consensus sequence, so propagating these to offspring by simple copying will take $4N^2\mu$ operations per generation. On the other hand, we must store $13N$ quantities per generation (two edges and one node per individual). The simplification algorithm is linear in the number of edges of the input tree sequence, and so multiplies this by a constant factor. Concretely, propagating neutral genotypes for T generations has time complexity $O(\mu T N^2)$, while our implementation is only $O(TN \log(2N))$.

Simulation benchmarks

Comparison of simulation with/without msprime, using simuPOP or maybe just a simple haploid simulation with 1000 QTL and stabilizing selection on a trait (say).

Maybe an estimate of how long *just* the ARG recording and simplification takes, so that then we can say how fast the simulator would have to be to do 10^6 whole chromosomes for 10^7 generations in a day.

Discussion

In this paper, we have shown that storing pedigrees and associated recombination events in a forwards-time simulation not only results in having available a great deal more information about the simulation, but also can speed up the simulation by orders of magnitude. To make this feasible, we have described how to efficiently store this information in tables, and have described a fundamental algorithm for simplificication of

tree sequences. Conceptually, recording of genealogical and recombination events can happen independently of the details of simulation; for this reason, we provide a well-defined and well-tested API in C and in python for use in other code bases.

The tree sequences produced by default by this method are very compact, storing genotype *and* genealogical information in a small fraction of the space taken by a compressed VCF file. The format is also highly efficient to process, leading to advantages for downstream analysis as well. This is because many algorithms to compute statistics of interest for population genetics are naturally expressed in terms of tree topologies, and so can be quickly computed from the trees underlying the tree sequence format. For example, pairwise nucleotide diversity π, is the average density of differences between sequences in the sample. To compute this directly from sequence data at m sites in n samples requires computing allele frequencies, taking $O(nm)$ operations. By using the locations of the mutations on the marginal trees, and the fact that these are correlated, dynamic programming algorithms similar to those in [Kelleher et al., 2016] can do this in roughly $O(n + m \log n)$ operations. The `msprime` API provides a method to compute π among arbitrary subsets of the samples in a tree sequence, which took about 1.2 seconds applied to the example tree sequence above with 1.1 million mutations in 200 megabases for 500,000 samples.

Another attractive feature of this set of tools is that it makes it easy to incorporate *prior history*, simply by seeding the simulation with a (relatively inexpensive) coalescent simulation. This allows for incorporation of deep-time history beyond the reach of individual-based simulations. Since geographic structure from times longer ago than the mixing time of migration across the range has limited effect on modern genealogies [Wilkins, 2004] (other than possibly changing effective population size Barton et al. [2002], Cox and Durrett [2002]), this may not negatively affect realism.

A final note: in preparing this manuscript, we debated a number of possible terms for the embellished pedigree, i.e., the "pedigree with ancestral recombination information". Current etymologica consensus [Liberman] has "pedigree" derived from the french "pied de grue" for the foot of a crane (whose branching pattern resembes the bifurcation of a single parent-offspring relationship). An analogous term for the embellished pedigree might be *nedigree*, from "nid de grue", as the nest of a crange is a large jumble of (forking) branches. We thought it unwise to use this term throughout the manuscript, but perhaps it will prove useful elsewhere.

Methods

some of this can move back to an appendix, but since they want methods at the end, it's the sort of thing they want here.

Simuation methods

Jaime and Kevin to describe simulation methods here (separately).

Data structures:

Moved from above.

Here we describe in more detail the data structures underlying a tree sequence in `msprime`. These derive from those described by Kelleher et al. [2016], but have been generalised and modified to remove redundancy.

To clarify terminology, below a *tree* refers to a genealogical tree describing how a collection of individuals are related to each other. A *tree sequence* contains information sufficient to reconstruct the genealogical tree relating all samples to each other at any point along the genome. In the context of a tree sequence, *nodes* refer to distinct ancestors, and correspond to the vertices in the trees of a tree sequence. Since each node represents a certain ancestor, it has a unique "time", thought of as her birth time, which determines the height of any branching points she is associated with. A given node will be associated with branching points of all trees across a region if that node is the most recent common ancestor to the subtending tips across that region. This information is stored in the columns of a **Node Table**, having columns *time*,

population, and *flags*. The column "flags" records other information (e.g., a binary mask of '1' indicates the node is a sample). Importantly, the **node ID** of a node is given implicitly by the (zero-based) index of its corresponding row in the Node Table.

Tree sequences are constructed by specifying over which segments of genome which nodes inherit from which other nodes. This information is stored by recording the endpoints of each distinctly inherited ancestral segment, the parental node, and the child node who inherited that segment. As each such record describes particular edge across a swatch of trees in the tree sequence, we call these *edges* and store them in the columns of an **Edge Table**, having columns *left*, *right*, *parent*, and *child*.

To record information about genetic variants we need to also record each mutation and which nodes have inherited that mutation. The tree structure takes care of inheritance – all we need to do is to record the highest node in the tree at the mutated site that inherited that mutation. As more than one mtuation may occur at a given site, we separate this information into two tables, first, the **Site Table** records for each variant site its *position* and *ancestral state*. Here *position* is a (floating point) position along the chromosome, and *ancestral state* is the genotype of the root of the tree at that site. As for nodes, **site IDs** are given implicitly by the (zero-based) index of the rows. Then, we record in a **Mutation Table** the *site*, *node*, and *derived state* for each mutation. Here "site" is the ID of the site at wihch this mutation occurred, "node" is the ID of the highest node that has inherited this mutation, and "derived state" is the genotype at this site of any individuals inheriting this mutation, unless another mutation occurs.

Definition of valid tables Here are the formal requirements for a set of nodes and edges to make sense, and to allow "msprime"'s algorithms to work properly.

To disallow time travel and multiple inheritance:

1. Offspring must be born after their parents (and hence, no loops).
2. The set of intervals on which each individual is a child must be disjoint.

For algorithmic reasons, we also require:

3. The leftmost endpoint of each chromosome is 0.0.
4. Node times must be strictly greater than zero.
5. Edges must be sorted in nondecreasing time order.

Note that since each node time is equal to the amount of time since the *birth* of the corresponding parent, time is measured in clock time, not in meioses.

A forwards-time simulation does **not** naturally emit genealogical information satisfying all these requirements. However, `msprime` implements two algorithms that will take a set of tables satisfying only 1–4 and produce tables satisfying all requirements. These are `sort_tables`, to enforce sortedness requirements (without renumbering nodes), and `simplify`, which works on sorted tables and is guarenteed to produce a valid, `msprime`-ready tree sequence.

Simplification

Here we describe the simplification algorithm. The flow is close to our implementation, but the description below uses set operations, while in the implementation, all ancestors are maintained as linked lists of segments ordered by left endpoint.

Algorithm S. (*Simplify a tree sequence*). Simplifies the input tree sequence: input consists of a list of sample IDs, S, of length n, a list of edges, $\mathcal{E}_\mathcal{I}$, and a list of nodes $\mathcal{N}_\mathcal{I}$. The genome length is L. The output will be stored in a list of edges $\mathcal{E}_\mathcal{O}$ and nodes $\mathcal{N}_O$. This also maintains the list N that gives the mapping from $\mathcal{N}_\mathcal{I}$ to $\mathcal{N}_\mathcal{O}$, so that if $M[p] = u$ then the p^{th} ancestor in N_I is the same as the u^{th} ancestor in N_0. The internal state is $\mathcal{A}$, the list of ancestors (which are collections of ancestral segments).

S0. [Initialize.] Set $\mathcal{A}[j] \leftarrow (0, L, j)$ and $\mathcal{N}_O[j] \leftarrow \mathcal{N}_I[S[j]]$ for $0 \le j < N$. Set $Q \leftarrow \emptyset$, and $M[p] \leftarrow -1$ for $0 \le p < \text{length}(N_I)$.

S1. [Input parent loop.] Loop over parents, chronologically.

S2. [Remove ancestry.] Call Algorithm R on each edge corresponding to the current parent.

S3. [Merge ancestors.] Merge the resulting segments into a new ancestor for this parent.

The first step is, for each edge (ℓ, r, p, c), simply to replace any ancestry segment $[a, b)$ held by ancestor c with $[a, b) \setminus [\ell, r)$ (deleting the segment or splitting in two if necessary), and adding the removed segment $[a, b) \cap [\ell, r)$ to the merge queue Q:

Algorithm R. (*Remove ancestry*). Given an edge (ℓ, r, p, c), and a list $\mathcal{A}[c]$ of the m ancestral segments corresponding to c, remove segments overlapping $[\ell, r)$ from $\mathcal{A}[c]$, and add those overlapping segments to Q.

R1. [Segment loop.] Loop over ancestral segments in $\mathcal{A}[c]$: if $j = m$, terminate; else let $(a, b, u) = \mathcal{A}[c][j]$.

R2. [Add to merge queue.] If $[a, b) \cap [\ell, r) \neq \emptyset$, append $(\max(a, \ell), \min(b, r), u)$ to Q.

R3. [Remove ancestry.] If $[a, b) \cap [\ell, r)! = \emptyset$, delete $\mathcal{A}[c][j]$ and insert in its place $(\min(a, \ell), \max(a, \ell), u)$ (if $a < \ell$) and/or $(\min(b, r), \max(b, r), u)$ (if $r < b$).

Next we must merge these ancestry segments, creating the new ancestor p, outputting edges and a node if coalescence has occurred.

Algorithm M. (*Merge ancestors*). Merge a collection of segments and create a new ancestor. This operates on the merge queue Q above, which is a list of ancestral segments, and creates the new ancestor $\mathcal{A}[p]$.

M1. [Segment loop.] While $\text{length}(Q) > 0$:

M2. [Find next segment overlap boundaries] Let $\ell \leftarrow \min\{x.\ell : x \in Q\}$ and $H \leftarrow \{x \in Q : x.\ell = \ell\}$. Also let $\ell_+ \leftarrow \min\{x.\ell : x \in Q \setminus H\}$ and $r = \min(\ell_+, \min\{x.r : x \in H\})$.

M3. [Copy single segment.] If $\text{length}(H) == 1$, set $\alpha \leftarrow (\ell, r, H[0].n)$. If also $p \in S$, then append $(\alpha.\ell, \alpha.r, N[p], \alpha.c)$ to $\mathcal{E}_O$ and then set $\alpha.c \leftarrow M[p]$. Go to M7.

M4. [Coalescence head.] If $(h \leftarrow \text{length}(H)) > 1$, if $M[p] == -1$ set $M[p] \leftarrow \text{length}(N_O)$ and append $N_I[p]$ to N_O. Set $\alpha \leftarrow (\ell, r, M[p])$.

M6. [Record edges.] For each $x \in H$, append $(\ell, r, M[p], x.n)$ to $\mathcal{E}_O$.

M7. [Update Q.] Set $Q \leftarrow Q \setminus \{x \in Q : x.r == r\}$ and then set $x.\ell = \ell$ for all remaining $x \in Q$.

M8. [Add segment to ancestor.] Append α to $\mathcal{A}[p]$ and return to M1.

Acknowledgements

Thanks to Brad Shaffer and Evan McCartney-Melstad for useful discussions. Funding for this project cam from XXX

References

Andre J. Aberer and Alexandros Stamatakis. Rapid forward-in-time simulation at the chromosome and genome level. *BMC Bioinformatics*, 14(1):216, July 2013. ISSN 1471-2105. doi: 10.1186/1471-2105-14-216. URL `https://doi.org/10.1186/1471-2105-14-216`.

Nick H. Barton, Frantz Depaulis, and Alison M. Etheridge. Neutral evolution in spatially continuous populations. *Theoretical Population Biology*, 61(1):31–48, February 2002. URL http://dx.doi.org/10.1006/tpbi.2001.1557.

A. Bhaskar, A. G. Clark, and Yun S. Song. Distortion of genealogical properties when the sample is very large. *Proc Natl Acad Sci U S A*, 111(6):2385–2390, 2014.

J. Theodore Cox and Richard Durrett. The stepping stone model: New formulas expose old myths. *Ann. Appl. Probab.*, 12(4):1348–1377, 11 2002. doi: 10.1214/aoap/1037125866. URL http://dx.doi.org/10.1214/aoap/1037125866.

Robert C. Griffiths and Paul Marjoram. An ancestral recombination graph. In *Progress in population genetics and human evolution (Minneapolis, MN, 1994)*, volume 87 of *IMA Vol. Math. Appl.*, pages 257–270. Springer, New York, 1997. URL http://www.math.canterbury.ac.nz/~r.sainudiin/recomb/ima.pdf.

Benjamin C. Haller and Philipp W. Messer. SLiM 2: Flexible, interactive forward genetic simulations. *Molecular Biology and Evolution*, 34(1):230–240, 2017. doi: 10.1093/molbev/msw211. URL /brokenurl#+http://dx.doi.org/10.1093/molbev/msw211.

Kelley Harris and Rasmus Nielsen. The genetic cost of Neanderthal introgression. *Genetics*, 203(2):881–891, June 2016. URL http://www.genetics.org/content/203/2/881.

R. R. Hudson. Properties of a neutral allele model with intragenic recombination. *Theor Popul Biol*, 23: 183–201, 1983.

Jerome Kelleher, Alison M Etheridge, and Gilean McVean. Efficient coalescent simulation and genealogical analysis for large sample sizes. *PLoS computational biology*, 12(5):e1004842, 2016.

David S. Lawrie. Accelerating Wright–Fisher forward simulations on the graphics processing unit. *G3: Genes—Genomes—Genetics*, 7(9):3229–3236, August 2017. doi: 10.1534/g3.117.300103. URL https://doi.org/10.1534%2Fg3.117.300103.

Anatoly Liberman. Little triumphs of etymology: "pedigree". https://blog.oup.com/2014/05/pedigree-etymology-word-origin/. Accessed: 2017-11-11.

P Marjoram and J D Wall. Fast "coalescent" simulation. *BMC Genet*, 7:16–16, 2006. doi: 10.1186/1471-2156-7-16. URL http://www.ncbi.nlm.nih.gov/pubmed/16539698.

Alicia R Martin, Christopher R Gignoux, Raymond K Walters, Genevieve L Wojcik, Benjamin M Neale, Simon Gravel, Mark J Daly, Carlos D Bustamante, and Eimear E Kenny. Human demographic history impacts genetic risk prediction across diverse populations. *The American Journal of Human Genetics*, 100(4):635–649, 2017.

Yosef E Maruvka, Nadav M Shnerb, Yaneer Bar-Yam, and John Wakeley. Recovering population parameters from a single gene genealogy: an unbiased estimator of the growth rate. *Mol Biol Evol*, 28(5):1617–1631, 2011.

B. Padhukasahasram, P. Marjoram, J. D. Wall, C. D. Bustamante, and M. Nordborg. Exploring population genetic models with recombination using efficient forward-time simulations. *Genetics*, 178(4): 2417–2427, April 2008. doi: 10.1534/genetics.107.085332. URL https://doi.org/10.1534%2Fgenetics.107.085332.

John Wakeley and Tsuyoshi Takahashi. Gene genealogies when the sample size exceeds the effective size of the population. *Mol Biol Evol*, 20(2):208–213, 2003.

Jon F. Wilkins. A Separation-of-Timescales Approach to the Coalescent in a Continuous Population. *Genetics*, 168(4):2227–2244, 2004. doi: 10.1534/genetics.103.022830. URL http://www.genetics.org/cgi/content/abstract/168/4/2227.

P R Wilton, S Carmi, and A Hobolth. The SMC' is a highly accurate approximation to the ancestral recombination graph. *Genetics*, 200(1):343–355, May 2015. doi: 10.1534/genetics.114.173898. URL `http://www.ncbi.nlm.nih.gov/pubmed/25786855`.

C Wiuf and J Hein. On the number of ancestors to a DNA sequence. *Genetics*, 147(3):1459–1468, November 1997. URL `http://www.ncbi.nlm.nih.gov/pubmed/9383085`.

C Wiuf and J Hein. The ancestry of a sample of sequences subject to recombination. *Genetics*, 151(3): 1217–1228, March 1999. URL `http://www.ncbi.nlm.nih.gov/pubmed/10049937`.