# Efficient pedigree recording for fast population genetics simulation

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

November 20, 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previous **OUTLINE**, possibly out-of-date but left for comparison.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, simply laying down neutral mutations on top of that history afterwards is equivalent to having generated them during the simulation. Precisely, we would need to know the genealogical tree relating all sampled individuals at each position along the genome. In this paper, we describe how algorithmic tools and data structures developed for the coalescent simulator `msprime` can be used 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* — the ARG [Griffiths and Marjoram, 1997], where nodes are common ancestor or recombination events — 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].

*Is this the right place for this paragraph? should it maybe be integrated with graf beginning 'Modern computing' above? this paragraph is supposed to cite the fwds time simulation methods we need to - @molpopgen fix this up?* 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 [Messer, 2013] to simulate ten thousand human exomes to assess the impact of genetic load and Neanderthal introgression on human genetic diversity. Sanjak et al. [2017] used fwdpp [Thornton, 2014] to simulate a series of models of quantiative traits under mutation-selection balance with population sizes of $2times10^4$ diploids in stable populations and populations growing up to $\approx 5 \times 10^5$ individuals, using the output to explore the relationship between the genotype/phenotype model and GWAS outcomes.

Due to tracking neutral mutations, the fastest simulation frameworks such as SLiM, SLiM2 Haller and Messer [2017] and fwdpp [Thornton, 2014] can “only” simulate tens of megabases of sequence in tens of thousands of individuals for tens of thousands of generations, and there are still near-order-of-magnitude performance differences between these implementations Haller and Messer [2017]. In practice, current state-of-the-art simulation software may take on the order of weeks to simulate models without selectionThornton [2014], Hernandez and Uricchio [2015], and different simulation engines differ in how efficiently they calculate fitnesses in models with selection Thornton [2014]. These population and region sizes are still substantially short of whole genomes (hundreds to thousands of megabases) for many biological population sizes of interest. Simulating very large popluations and entire genomes will likely require parallelization, which Lawrie [2017] used to improve performance for the "Poisson Random Field" family of models Sawyer and Hartl [1992], in which mutations are unlinked and there is no effect of genetic background on fitness. However, little, and perhaps even no, attention has been paid to parallelization of simulations involving linkage. Here, we show that recording genealogical information instead of explicitly tracking neutral mutations results in order-of-magnitude performance improvements to simulations based on `fwdpp`. The idea of storing genealogical information to speed up simulations is not new. AnA-FiTS implemented it completely [Aberer and Stamatakis, 2013], 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.

## Results

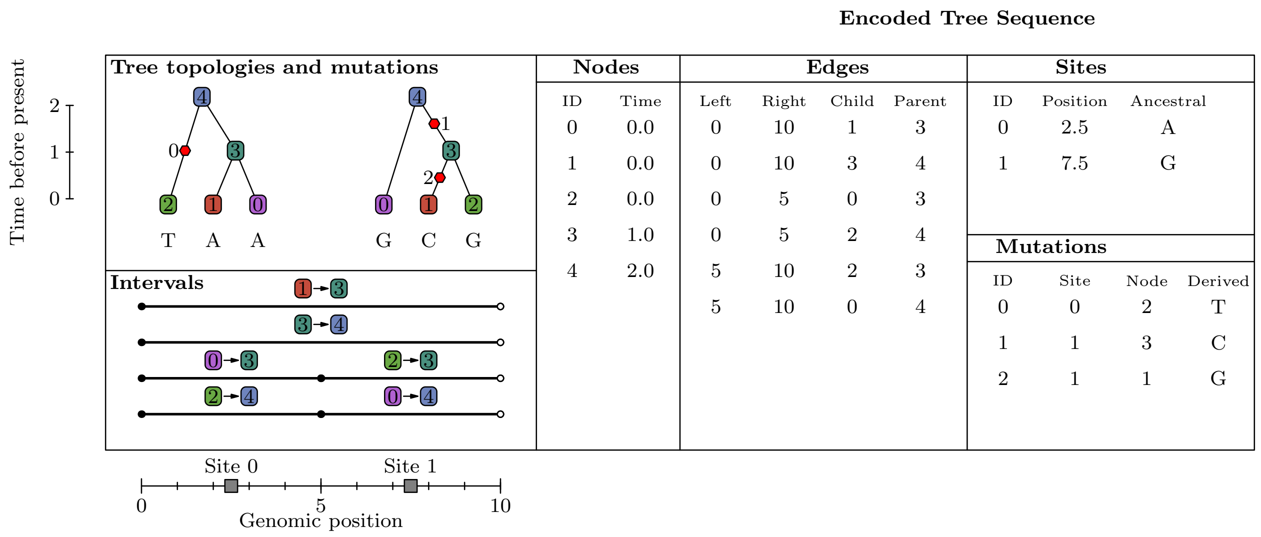
Below, we first describe the conceptual and algorithmic foundations for the method: (a) a format, implemented in the `msprime` Python API, for recording tree sequences in several *tables*; (b) an algorithm, W, to record tree sequence information into these tables on the fly from a forwards-time simulation; and (c) an algorithm to *simplify* a tree sequence, i.e., remove redundant information. We then describe a general-purpose method that uses these tools (as implemented in `msprime`) to efficiently record simplified a tree sequence, and analyze of its run time complexity. This is followed by benchmarking the performance improvement achieved by connecting two previously published forwards-time simulation libraries, `simuPOP` [**?**] and `fwdpp` [Thornton, 2014] to the tools in `msprime`.

### Working with a tree sequence through tables

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.



**Nodes**

| ID | Time |
|---|---|
| 0 | 0.0 |
| 1 | 0.0 |
| 2 | 0.0 |
| 3 | 1.0 |
| 4 | 2.0 |

**Edges**

| Left | Right | Child | Parent |
|---|---|---|---|
| 0 | 10 | 1 | 3 |
| 0 | 10 | 3 | 4 |
| 0 | 5 | 0 | 3 |
| 0 | 5 | 2 | 4 |
| 5 | 10 | 2 | 3 |
| 5 | 10 | 0 | 4 |

**Sites**

| ID | Position | Ancestral |
|---|---|---|
| 0 | 2.5 | A |
| 1 | 7.5 | G |

**Mutations**

| ID | Site | Node | Derived |
|---|---|---|---|
| 0 | 0 | 2 | T |
| 1 | 1 | 3 | C |
| 2 | 1 | 1 | G |

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!*

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 [Kelleher et al., 2016].

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 (but 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 $\delta$. 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).

*TODO: rewrite with tables* We begin in W1 by allocating our initial population $P$ (a vector of $N$ node IDs), and creating $N$ nodes with birth time $T$ generations ago, recorded in the vector $\tau$. 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 parental IDs $P_a$ 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.

**Algorithm W.** (*Forwards-time tree sequence*). Simulates a randomly mating population $P$ of $N$ haploid individuals, with a probability of death per generation $\delta$ and a chromosome of length $L$ for $T$ generations. Events are recorded in a node table $\mathcal{N}$ and edge table $\mathcal{E}$. Simplify is run every $s$ generations.

```
W1. [Initialisation.] Set $\mathcal{N} \leftarrow$ NodeTable() and $\mathcal{E} \leftarrow$ EdgeTable() and $t \leftarrow T$. For $0 \leq j < N$, set $P_j \leftarrow \text{add}(\mathcal{N}, T)$.
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.
W4. [Mortality.] If $\mathcal{R}_U([0, 1)) \geq \delta$ go to W8.
W5. [New node.] Set $u \leftarrow \text{add}(\mathcal{N}, t)$ and $P'_j \leftarrow u$.
W6. [Choose parents.] Set $a \leftarrow \mathcal{R}_U(\{0, \dots, N-1\})$, $b \leftarrow \mathcal{R}_U(\{0, \dots, N-1\})$ and $x \leftarrow \mathcal{R}_U([0, L))$.
W7. [Record edges] Call $\text{add}(\mathcal{E}, (0, x, P_a, u))$ and $\text{add}(\mathcal{E}, (x, L, P_b, u))$.
W8. [Loop tail] Set $j \leftarrow j + 1$ and go to W3.
W9. [Simplify] TODO
```

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 $\mu$ per generation per unit length then there are Poisson $(\mu(r - \ell)(\tau_p - \tau_c))$ mutations on each edge $(\ell, r, p, c)$.

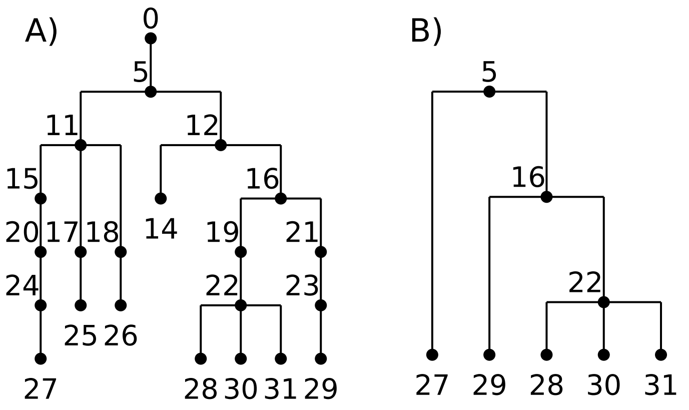


Figure 2: An example of a marginal genealogy from a Wright-Fisher simulation with $N = 5$. **(A)** the original tree including all intermediate nodes and dead-ends, and **(B)** the minimal tree relating all of the currently-alive individuals (27–31).

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 $(\ell, x, p, c)$ and $(x, r, p, c)$ which can be represented with a single edge $(\ell, r, p, c)$.

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 Figures 3 and 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 $(\ell, r, u)$ 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.

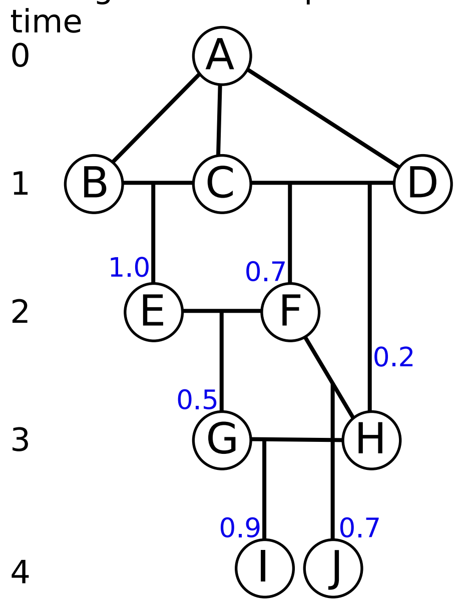
## Forward simulation and sequential simplification

At any point in a simulation scheme that, like Algorithm W, records data into tables the genealogical history is recorded in a tree sequence. This has two additional advantages, as seen in Algorithm W. 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`. We provide results for simulation timing from two specific implementations of this method (forward simulation coupled with sequential simplification) in the next section, but first we analyze the expected resource use of the general Algorithm W.

### Estimates of run-time complexity

*It'd be good to outline what readers should expect here. and note we're doing the space analysis for with reference to alg W with $s = 1$* Consider a simulation of a Wright-Fisher population of $2N$ haploid individuals using Algorithm W for $T$ generations, with the probability of death, $\delta$, 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 each time the marginal tree changes along the sequence, three edges end and three new edges begin (except for changes affecting the root; see Kelleher et al. [2016]). 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 three times this gives us an

(A) The original tree sequence:



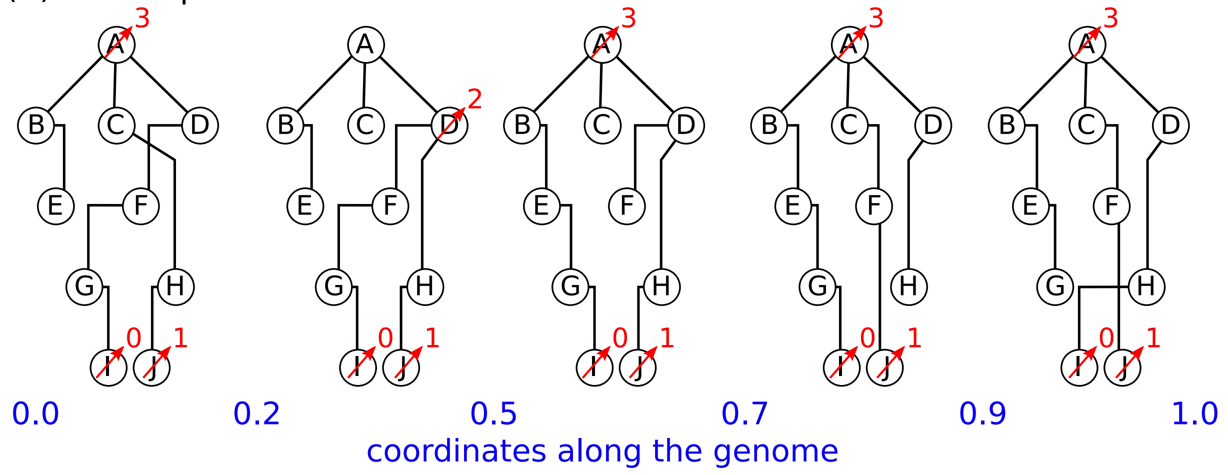
**Nodes:**

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

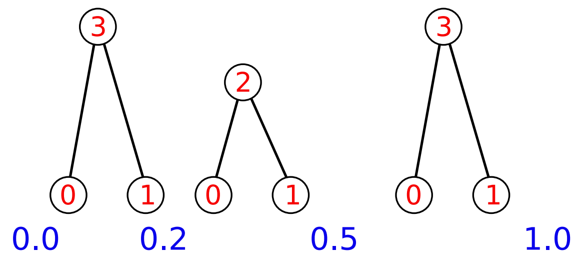
**Edges:**

| 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 |

(B) The sequence of trees:



(C) Simplified for I,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 simplify 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.

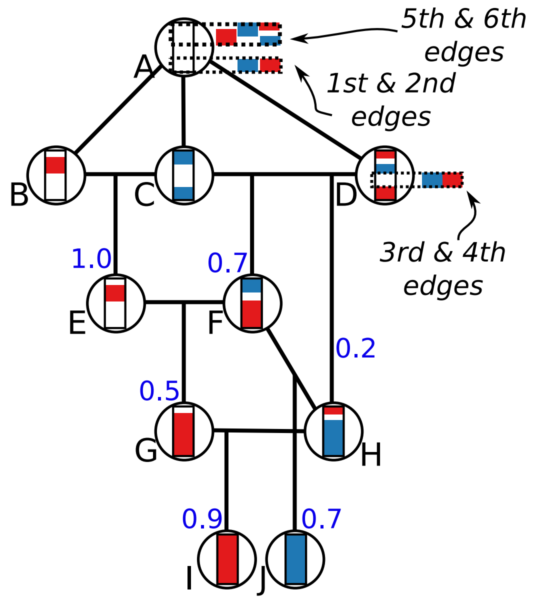


Figure 4: A depiction of the state of the simplification algorithm at each point in time, in the example of Figure 3.

upper bound on the expected 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 $8N(4\log(2N)+3)$; at $N = 10^4$, a factor of 4,700.

To get an idea of how required space depends on the length of time, suppose instead that we have run a simulation of $2N$ individuals for $T$ generations, begun with no prior history. This produces $4NT$ edges; how many are required after simplification? As above, the expected number of edges is bounded by $2N-2$ plus four times the mean marginal tree length. Roughly, the length of time for which a marginal tree has k tips is $2N/k(k-1) = 2N(1/(k-1) - 1/k)$, and so the time to go from N to n tips is $2N(1/(n-1) - 1/(N-1))$. This implies that after running for time $T$ we expect to have around $n(T)$ roots, where $n(T) \approx 1 + 2N/(T+2)$. The total tree length over this time is $2N\sum_{k=n(T)+1}^{N-1} 1/k$, which leads to an upper bound on the number of edges of

$$2N\left(1 + 4\log\left(\frac{NT}{T+2N}\right)\right).$$

This holds up quite well in practice – the number of edges actually required for 50 independent Wright–Fisher simulations, as a function of time (obtained by running simplify every generation) is shown in 5, compared to this prediction.

If we were to add mutations in forwards time under the infinite-sites model with total mutation rate per generation $\mu$, 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

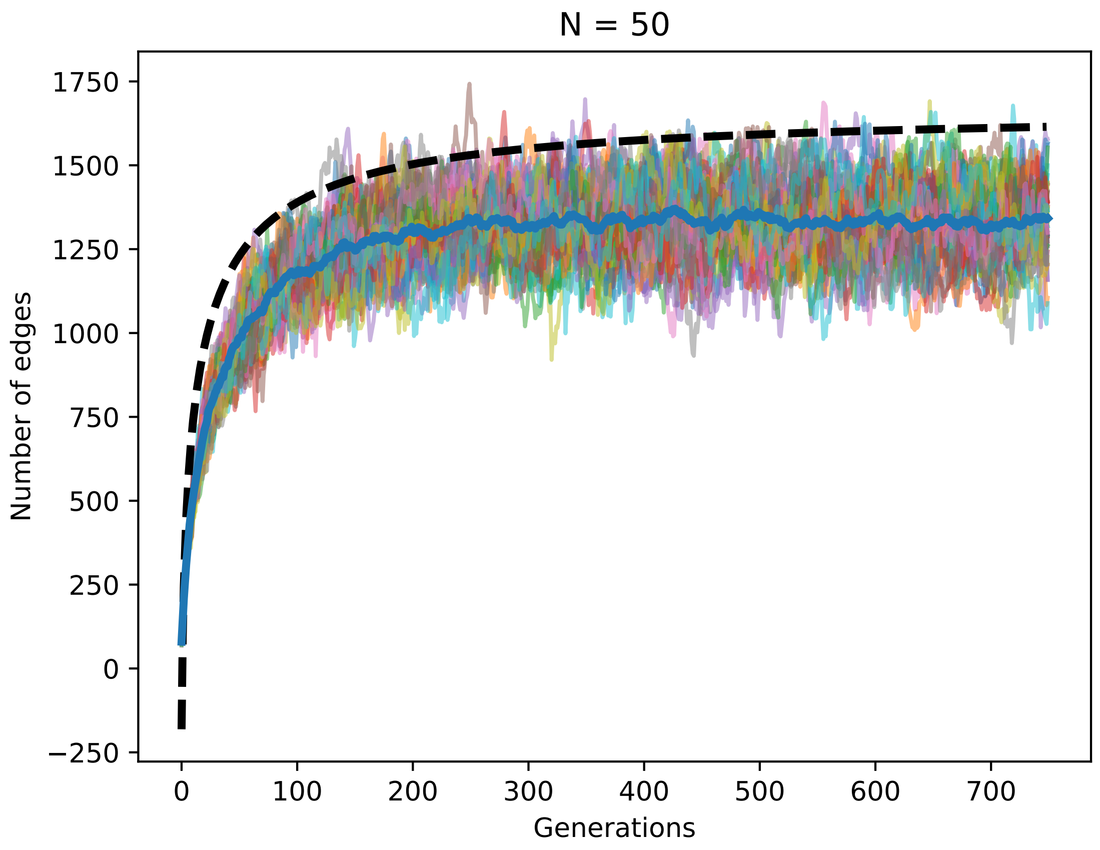


Figure 5: Number of edges in the simplified tree sequence for 50 Wright-Fisher simulations begun with no prior history as a function of number of generations. Each line is one simulation, the black line gives the average, and the dotted line is the upper bound of the text.

$13N$ 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 $\pi$, 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 $\pi$ 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 crane 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).

## Overview of the API

The `msprime` Python API provides a powerful platform for working with tree topology and mutation data. The new portions of `msprime` that we cover here are dedicated to tree sequence input and output using simple tables of data, so we refer to this as the 'Tables API'. The four key tables: nodes, edges, sites and mutations, are described above.

The Tables API is primarily designed to facilitate efficient interchange of data between programs or between different modules of the same program. *should capitalize Tables, or not, as above* Following the current best-practises [citations: apache arrow, etc] data is stored in a columnar format. The principal advantage of this 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. Thus, 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.

The `msprime` tables API is independent of the details of a particular simulation engine. In other words, one simply needs to fill the relevant columns and send them to `msprime` at regular intervals for simplification. For example, using the `fwdpp` C++ API Thornton [2014], we fill arrays of simple structures representing the node and edge data. Doing so requires no modification of the `fwdpp` code base. Rather, we simply need to book-keep the parent/offspring labels and perform simple processing of the recombination breakpoints from each mating event. Using `pybind11` (`https://github.com/pybind/pybind11/`), the node/edge arrays on the C++ side are made visible, without copy, to Python as a multi-column numpy array. This array is in turn processed by the `msprime` tables API. In order to create a standalone user interface, our implementation uses `pybind11` to create a Python package extending `fwdpy11` (`https://github.com/molpopgen/fwdpy11`), which is a Python package based on `fwdpp`. (A neat aside, that we may wish to skip here, is that it may be possible to use the tables API from a command-line C/C++ application via an embedded Python interpreter.) *@molpopgen: note on how this works with fwdpp?* *@plr: done, but perhaps too verbose?*

The tables API provides basic input and output operations via the numpy array interface, which provides a great deal of flexibility as well as efficiency, and makes it straightforward to transfer data from sources such as HDF5 [citation], Dask, or Zarr. (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. (Simplification may also be required.)

*Maybe we don't need this paragraph? Already said it, more or less, above, esp if we insert something about how fwdpp works.* *agree re the next two lines.* *propose moving the other above* 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].

## The table format

Given a set of node and edge tables as described above, there are a small set of requirements that ensure the tables describe a valid tree sequence. (There are essentially no such requirements on the site and mutation tables.) These are:

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.

A pair of node and edge tables that satisfy these two requirements is guarenteed to uniquely describe at each point on the genome a collection of directed, acyclic graphs – in other words, a forest of trees. For some applications it is necessary to check that at every point there is only a *single* tree. Checking this is more difficult, but is implemented in `msprime`'s API. *TODO add function name*

For efficiency, `msprime` makes several other sortedness requirements on the tables, that are not necessarily satisfied by tables emitted by a forwards-time simulation. `msprime`'s API includes tools to rectify this by first sorting (using `sort_tables`) and then using `simplify`, which works on sorted tables and is guarenteed to produce a valid, `msprime`-ready tree sequence.

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. *is this the right place? I'd put it in discussion of Figure 1*

## 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. As it goes, the algorithm records a list $M$ that gives the mapping from input nodes to output nodes, so that if $M[p] = u$ then the $p^{\text{th}}$ node in the input is the same as the $u^{\text{th}}$ node in the output. The current state of the algorithm is recorded in $\mathcal{A}$, a list of ancestors each of which is a collections of ancestral segments. An ancestral segment is a triple $(\ell, r, x)$, indicating that the output node $x$ inherits from this ancestor on the genomic segment $[\ell, r)$.

**Algorithm S.** (*Simplify a tree sequence*). Input consists of a list $S$ of sample IDs, a list $\mathcal{N}_I$ of nodes (ordered by birth time), a list $\mathcal{E}_I$ of edges, and the genome length, $L$. The output is a nodes $\mathcal{N}_O$, and list of edges $\mathcal{E}_O$.

**S0.** [Initialize.] Set $\mathcal{A}[j] \leftarrow (0, L, j)$ and $\mathcal{N}_O[j] \leftarrow \mathcal{N}_I[S[j]]$ for $0 \le j < \text{length}(S)$. Set the current input parent index to $p = 0$ and $M[u] \leftarrow -1$ for $0 \le u < \text{length}(\mathcal{N}_I)$.

**S1.** [Input parent loop.] Set $Q \leftarrow \emptyset$.

**S2.** [Remove ancestry.] Call Algorithm R on each edge $(\ell, r, p, c)$ in $\mathcal{E}_I$ corresponding to parent $p$ (which stores the resulting segments in the merge queue $Q$).

**S3.** [Merge ancestors.] Call Algorithm M on $Q$ (which adds to $\mathcal{N}_O$ and $\mathcal{E}_O$).

**S4.** [Next parent.] Set $p \leftarrow p + 1$; if $p = \text{length}(\mathcal{N}_I)$, terminate and return $\mathcal{N}_O$ and $\mathcal{E}_O$; otherwise, return to S1.

The algorithm steps back up through the pedigree, since nodes are ordered by time-since-birth. First, for each input node $p$, we must find all ancestral segments inheriting from $p$ – so, for every edge $(\ell, r, p, c)$, Algorithm R replaces any ancestry segment $[u, v)$ held by $c$ with $[u, v) \setminus [\ell, r)$ (deleting the segment or splitting in two if necessary), and adding the removed segment $[u, v) \cap [\ell, r)$ to the merge queue $Q$:

**Algorithm R.** (*Remove ancestry*). Given an edge $(\ell, r, p, c)$, 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 $Q$.

**R0.** [Initialize.] Set $j \leftarrow 0$.

**R1.** [Segment loop.] If $j = m$, terminate; else let $(u, v, x) = \mathcal{A}[c][j]$. (Here $x$ is the output node that inherits from this segment.)

**R2.** [Add to merge queue.] If $[u, v) \cap [\ell, r) = \emptyset$, skip to R4. Otherwise, append $(\max(u, \ell), \min(v, r), x)$ to $Q$.

**R3.** [Remove ancestry.] Delete $\mathcal{A}[c][j]$ and insert in its place $(\min(u, \ell), \max(v, \ell), x)$ (if $u < \ell$) and(/or) $(\min(v, r), \max(v, r), x)$ (if $r < v$).

**R4.** [Iterate.] Set $j \leftarrow j + 1$ and return to R1.

Next we must merge the ancestry segments in $Q$, creating the new ancestor corresponding to input node $p$, adding a node and edges to the output if any segments overlap, which implies that an output coalescence has occurred. Below we index elements of an ancestral segment $z = (\ell, r, u)$ by writing $z$.left $= \ell$ and $z$.right $= r$ for the endpoints, and $z$.id $= u$ for the output node ID.

**Algorithm M.** (*Merge ancestors*). Merge the ancestral segments in $Q$, and create the new ancestor $\mathcal{A}[p]$. This algorithm also appends to $\mathcal{N}_O$ and $\mathcal{E}_O$, and needs to know about $S$, to check if input nodes are samples.

**M0.** [Initialize.] Set $z \leftarrow (0, 0, -1)$ and $d =$ False.

**M1.** [Segment loop.] If length$(Q) = 0$, go to M9.

**M2.** [Find next segment overlap boundaries] Let $\ell_0 \leftarrow \min\{x.\text{left} : x \in Q\}$ and $H \leftarrow \{x \in Q : x.\text{left} = \ell_0\}$. Also let $\ell_+ \leftarrow \min\{x.\text{left} : x \in Q \setminus H\}$ and $r = \min(\ell_+, \min\{x.\text{right} : x \in H\})$.

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

**M4.** [New output node.] If $M[p] = -1$ set $M[p] \leftarrow \text{add}(\mathcal{N}_O, \mathcal{N}_I[p])$ Set $\alpha \leftarrow (\ell, r, M[p])$.

**M6.** [Record edges and update $Q$.] For each $x \in H$, do add$(\mathcal{E}_O, (\ell, r, M[p], x.\text{id}))$.

**M7.** [Update $Q$.] Remove from $Q$ any segment $x \in H$ with $x.\text{right} = r$ and then set $x.\text{left} = \ell$ for all remaining $x \in H$.

**M8.** [Add segment to ancestor.] If $z.\text{right} = \alpha.\text{left}$ and $z.\text{id} = \alpha.\text{id}$ set $d =$ True. Append $\alpha$ to $\mathcal{A}[p]$, set $z \leftarrow \alpha$, and return to M1.

**M9.** [Defragment.] If $d$ is True, find and merge any adjacent segments in $\mathcal{A}[p]$ with the same id. Terminate.

## 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`.

Ryan D Hernandez and Lawrence H Uricchio. SFS_CODE: More efficient and flexible forward simulations. August 2015.

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.

Philipp W Messer. SLiM: simulating evolution with selection and linkage. *Genetics*, 194(4):1037–1039, August 2013.

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`.

Jaleal S Sanjak, Anthony D Long, and Kevin R Thornton. A model of compound heterozygous, Loss-of-Function alleles is broadly consistent with observations from Complex-Disease GWAS datasets. *PLoS Genet.*, 13(1):e1006573, January 2017.

S A Sawyer and D L Hartl. Population genetics of polymorphism and divergence. *Genetics*, 132(4):1161–1176, December 1992.

Kevin R Thornton. A c++ template library for efficient forward-time population genetic simulation of large populations. *Genetics*, 198(1):157–166, September 2014.

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`.