bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

Efficient pedigree recording for fast population genetics simulation

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

January 23, 2018

Abstract

In this paper we describe how to efficiently record the entire genetic history of a population during a forwards-time, individual-based population genetics simulation. This dramatically reduces the computational burden of tracking individual genomes by simulating only those loci that may affect reproduction (those having non-neutral variants), and recording history as a *succinct tree sequence* as introduced in the software package `msprime`, on which neutral mutations can be quickly placed afterwards. We implement the method in two popular forwards-time simulation frameworks, which speeds up large, whole-genome simulations by one to two orders of magnitude. In addition to speed, the method has several advantages: (1) All marginal genealogies of the simulated individuals are recorded, rather than just genotypes. (2) A population of N individuals with M polymorphic sites can be stored in $O(N \log N + M)$ space, making it feasible to store a simulation's entire final generation (and its' history) rather than only a subset. (3) A simulation can easily be initialized with a more efficient coalescent simulation of deep history. To make these possible, we introduce the `msprime` Tables API, which allows for the efficient interchange of tree sequences between modular program components, and provide an algorithm to quickly simplify a tree sequence by removing history irrelevant to a given set of genomes.

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 [Hudson, 1990, Neuhauser and Krone, 1997], but it remains a challenge to map results of coalescent models onto species that are distributed across continuous geographical space [Barton et al., 2010, Kelleher et al., 2014] 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, density-dependent effects on fitness, 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 to examine biases in GWAS. Several studies have now shown that in samples approaching the size 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 large forwards-time, individual-based simulations. 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,

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

2014] to simulate a series of models of quantitative traits under mutation-selection balance with population sizes of 2×10^4 diploids in stable populations and populations growing up to $\sim 5 \times 10^5$ individuals, using the output to explore the relationship between the genotype/phenotype model and GWAS outcomes.

Modern computing power easily allows simulations of birth, death and reproduction in a population having even hundreds of millions of individuals. 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 record each individual's genome. As samples of most species' genomes harbor tens or hundreds of millions of variant sites, carrying full genotypes for even modest numbers of individuals through a simulation can quickly become prohibitive. To make matters worse, a population of size N must be simulated across many multiples of N generations to produce stable genetic patterns [Wright, 1931, Wakeley, 2005]. Because of this computational burden, even the fastest simulation frameworks such as SLiM 2 [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. In practice, current state-of-the-art simulation software may take on the order of weeks to simulate models of large genomic regions without selection [Thornton, 2014, Hernandez and Uricchio, 2015], and existing 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.

However, it is thought that most genetic variation is selectively neutral (or nearly so). By definition, neutral alleles carried by individuals in a population 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: it does not matter if we generate each generation's mutations during the simulation, or afterwards. To add mutations after the fact, we need to know the genealogical trees relating all sampled individuals at each position along the genome. Combined with ancestral genotypes and the origins of new mutations, these trees completely specify the genomic sequence of any individual in the population at any time. To obtain this information, we record from forward simulation the *population pedigree* – the complete history of parent-offspring relationships of an entire population going back to a remote time – and the genetic outcomes of each ancestral meiosis, periodically discarding all information irrelevant to the genetic history of the extant population. The information in this embellished pedigree is stored as a *succinct tree sequence* (or, for brevity, "tree sequence"), which contains all the information necessary to construct the genealogical tree that relates each individual to every other at each position on the genome.

The idea of storing genealogical information to speed up simulations is not new. It was implemented in AnA-FiTS [Aberer and Stamatakis, 2013], but without the critical step of discarding irrelevant genealogical information. Padhukasahasram et al. [2008] obtained impressive speedups for a Wright–Fisher simulation by keeping track of genealogies over the preceding 8 generations and only tracking neutral genotypes for those segments having descendants across this window. Our approach is similar, but uses genealogies across the entire duration of the simulation. The embellished pedigree is equivalent to the *ancestral recombination graph*, or ARG [Griffiths, 1991, Griffiths and Marjoram, 1997], which has been the subject of substantial study [Wiuf and Hein, 1997, 1999, Marjoram and Wall, 2006, Wilton et al., 2015]. However, it is unclear if an ARG-based approach would share the computational advantages of the data structures we use here [Kelleher et al., 2016].

In this paper, we describe a storage method for *succinct tree sequences* (and hence, genome sequence) as well as an algorithm for simplifying these. The data structure is *succinct* in the sense that its' space usage is close to optimal, while still allowing efficient retrieval of information (see, e.g., Gog et al. [2014]). The storage method and simplification algorithm are both implemented as improvements to the algorithmic tools and data structures in `msprime`. We also describe how these tools can efficiently record, and later process, the embellished population pedigree from a forwards-time simulation. While providing substantial savings in computational time and space, our methods provide much more information than simply simulating the genomes – the tree sequence encodes all marginal genealogies of individuals living at the end of the simulation. Although we were motivated by a need for more efficient genomic simulations, these tools may prove more widely useful.

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

Results

The strategy described above is only of interest if it is computationally feasible. Therefore, we begin by benchmarking the performance improvement achieved by this method, implemented using the forwards-time simulation library `fwdpp` [Thornton, 2014] and tree sequence tools implemented in `msprime`. Then, we describe the conceptual and algorithmic foundations for the method: (a) a format, implemented in the `msprime` Python API, for recording tree sequences efficiently in several *tables*; (b) an algorithm to record these tables during a forwards-time simulation; and (c) an algorithm to *simplify* a tree sequence, i.e., remove redundant information. Finally, we analyze the run time and space complexity of our general-purpose method.

Simulation benchmarks

To measure the performance gains from recording the pedigree we ran simulations both with and without recording. (Although we record more than just the parent–offspring relationships of the pedigree, for brevity we refer to the method as "pedigree recording".) All simulations used `fwdpp` to implement a discrete-time Wright-Fisher population of N diploid individuals, simulated for $10N$ generations (details below). Simulations without pedigree recording introduced neutral mutations at a rate equal to the recombination rate, so $\mu = r$, where μ and r are the expected per-generation number of mutations per gamete and recombination breakpoints per diploid, respectively. Simulations with pedigree recording introduced neutral mutations at the same rate retrospectively, as described below, resulting in statistically identical simulation results. We ran simulations with different values of N and varied the size of the genomic region according to the scaled recombination parameter $\rho = 4Nr$.

Deleterious mutations were introduced at rate $\rho/100$ per generation, drawing scaled selection coefficients ($2Ns$) from a Gamma distribution with a mean of -5 and a shape parameter of 1. This distribution of fitness effects results in thousands of weakly-deleterious mutations segregating in the population, many of which drift to intermediate frequencies. The case of many mutations with selection is a non-trivial violation of exchangeability assumptions of the coalescent [Neuhauser and Krone, 1997]. Therefore, these selected mutations must be explicitly tracked in our forward simulation and the time savings due to pedigree recording come from not having to record *neutral* mutations.

Pedigree tracking dramatically reduced runtimes, as shown in Figure 1, producing a relative speedup of up to ≈ 50 fold relative to standard simulations that track neutral mutations (Figure 2). Pedigree tracking results in greater relative speedups for larger N and we observe increasing relative speedups as $4Nr$ increases for a given N (Figure 2). Importantly, runtimes are approximately linear in region size ρ when pedigree tracking (partially obscured by the log scale of the horizontal axis in Figure 1). In a more limited set of neutral simulations we found the same qualitative behavior, and a numerically larger speedup by using pedigree tracking (see Appendix D).

In our implementation, simulations with pedigree recording used substantially more RAM than simple forward simulations (see Appendix B). This is unsurprising: unsimplified tree sequences grow quickly, and so storing history can use arbitrarily much memory. However, this is not a requirement of the method, only a straightforwards consequence of a speed–memory tradeoff: the amount of required memory is mostly determined by the interval between simplification steps, but less frequent simplification reduces overall computation time (see Appendix C). In fact, our method could in some situations *reduce* the amount of memory required, if memory usage in the forwards simulation was dominated by the cost of maintaining neutral genetic variants.

Tables for succinct tree sequences

We now explain what we actually did to achieve this $50\times$ speedup. The "pedigree recording" simulations above recorded information about each new individual in a collection of tables that together define a *succinct tree sequence* (or, simply "tree sequence"). A tree sequence is an encoding for a sequence of correlated trees, such as those describing the history of a sexual population. Tree sequences are efficient because branches

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

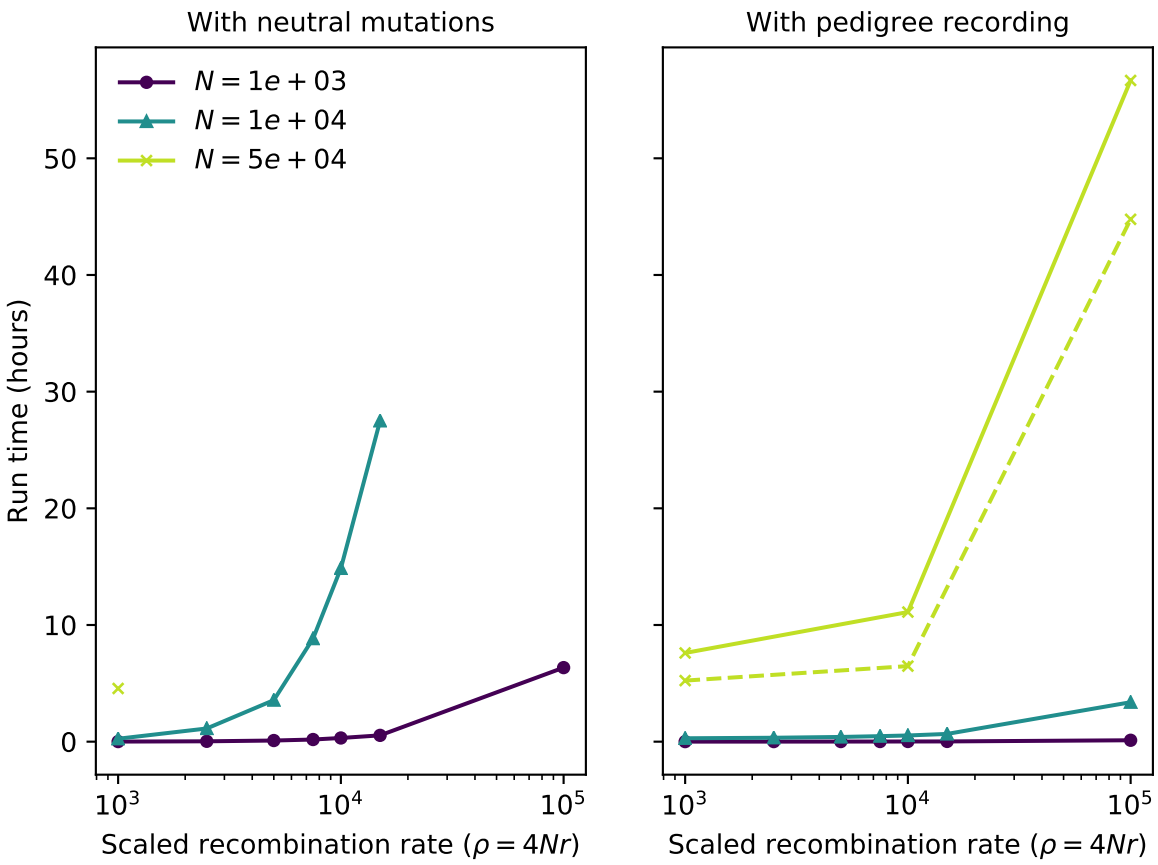


Figure 1: Total run time per single simulation replicate as a function of region length. Line color represents different diploid population sizes (N). The left figure shows run times for standard simulations including neutral mutations. The right column of panels shows run times of simulations that recorded the pedigree and added neutral mutations afterwards. The dashed line in the right panel shows results for an implementation using `fwdpy11` where the pedigree simplification steps were handled in a separate thread of execution and fitness calculations were parallelized. Simulations with $N = 5 \times 10^4$ timed out for region sizes larger than 10^3.

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

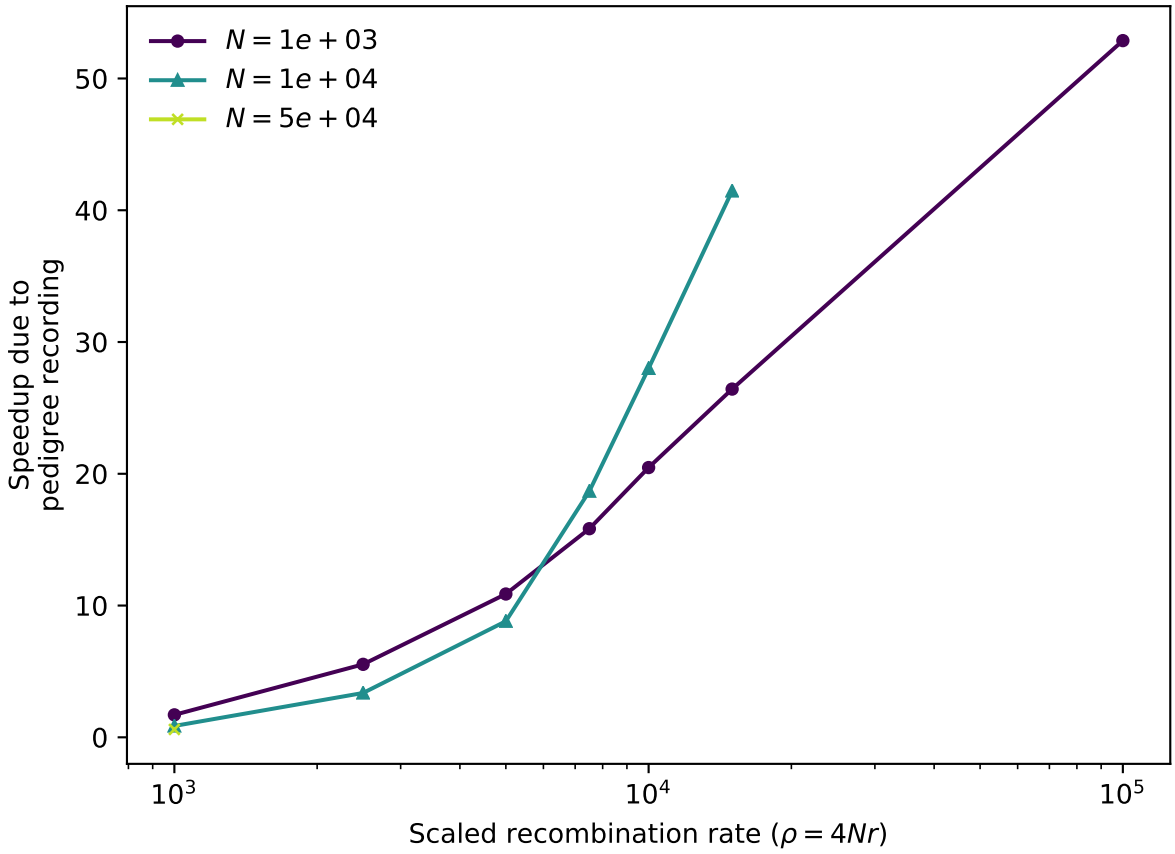


Figure 2: Relative speedup of simulations due to pedigree recording. Each line shows the ratio of total run times of standard simulations to those of simulations with pedigree recording. Data points are taken from Figure 1 for simulations that ran to completion in both cases.

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

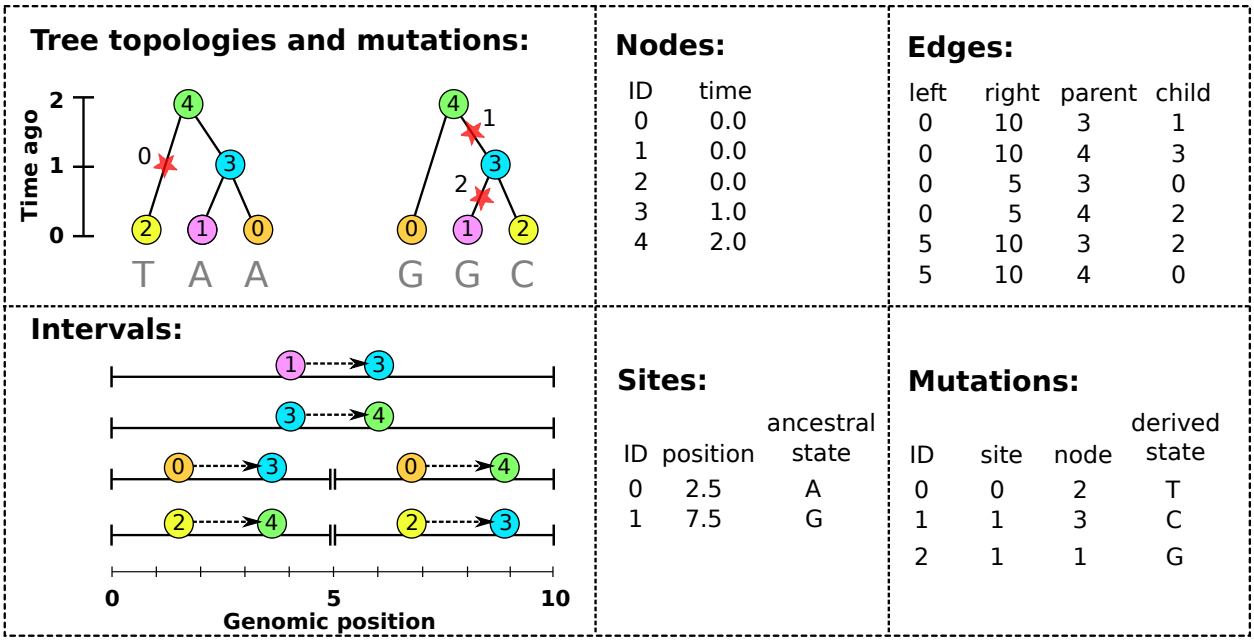


Figure 3: An example tree sequence with three samples over a chromosome of length 10. The leftmost panels show the tree sequence pictorially in two different ways: (top) a sequence of tree topologies; the first tree extends from genomic position 0 to 5, and the second from 5 to 10; and (bottom) the edges that define these topologies, displayed over their corresponding genomic segment (for instance, the edge from node 2 to node 4 is present only on the interval from 0 to 5). The remaining panels show the specific encoding of this tree sequence in the four tables (nodes, edges, sites and mutations) defined by `msprime`.

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 is recorded as *sites* and *mutations*; we give an example in Figure 3. This formulation is derived from the "coalescence records" encoding of tree sequences [Kelleher et al., 2016], normalised to remove redundancy and generalised to include a more general class of tree topologies.

The *nodes* of a tree sequence correspond to the vertices in the individual genealogies along the sequence. Each node refers to a specific, distinct ancestor, and so has a unique "time", thought of as the node's birth time, which determines the height of any vertices the node is associated with. (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.) The example of Figure 3 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 one and two units of time in the past, 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 in this interval. The spatial extent of the edges defining the topology of Figure 3 are shown in the bottom left panel. For example, the branch joining nodes 1 to 3 appears in 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 compact and algorithmically efficient.

Recovering the sequence of trees from this information is straightforward: each point along the genome at which 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

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

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 simply 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 to compute statistics of the genome sequence 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 records two things: its position on the genome and an ancestral state. For example, in Figure 3 we have two sites, one at position 2.5 with ancestral state 'A' and the other at position 7.5 with ancestral state 'G'. If no mutations occur at a given site, all nodes inherit the ancestral state. Each mutation records three things: the site at which it occurs, the first node to inherit the mutation, and the derived state. Thus, all nodes below the mutation's node in the tree will inherit this state, unless further mutations are encountered. Three mutations are shown in Figure 3, illustrated by red stars. The first site, in the left-hand tree, has a single mutation, which results 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 used `msprime` to simulate 500,000 samples of a 200 megabase chromosome with human-like parameters: $N_e = 10^4$ and per-base mutation and recombination rates of 10^{-8} per generation. This resulted in about 1 million distinct marginal trees and 1.1 million infinite-sites mutations. The HDF5 file encoding the node, edge, site and mutation tables (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 (giving a compression factor of 144,000 in this instance). Working with either the Newick or VCF encoding of this dataset would likely require several days of CPU time just to read the information into memory.

Validity of a set of tables Given a set of node and edge tables as described above, there are only two requirements that ensure the tables describe a valid tree sequence. 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 guaranteed 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. 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 and then using the `simplify` algorithm described below, which works on sorted tables and is guaranteed to produce a valid, `msprime`-ready tree sequence.

The `msprime` Tables API

The facilities for working with succinct tree sequences are implemented as part of the `msprime` Python API, which provides a powerful platform for processing tree topology and mutation data. The new portions of `msprime` that we discuss here are dedicated to tree sequence input and output using simple tables of data, as described above, so we refer to this as the "Tables API".

The Tables API is primarily designed to facilitate efficient interchange of data between programs or between different modules of the same program. We adopted a 'columnar' design, where all the values for a particular column are stored in adjacent memory locations. There are many advantages to columnar storage – for example, since adjacent values in memory are from the same column, they tend to compress well, and

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

suitable encodings can be chosen on a per-column basis [Abadi et al., 2006]. A particular advantage of this approach is that it enables very efficient copying of data, and in principle zero-copy data access (where a data consumer reads directly from the memory of a producer). Our implementation efficiently copies data from Python as a NumPy array [Walt et al., 2011] into the low-level C library used to manipulate tree sequences. This architecture allows for data transfer rates of gigabytes per second (impossible under any text-based approach), while retaining excellent portability. NumPy's array interface provides a great deal of flexibility and efficiency, and makes it straightforward to transfer data from sources such as HDF5 [The HDF Group, 1997-2018] or Dask [Dask Development Team, 2016]. For small scale data and debugging purposes, a simple text based format is also supported.

The `msprime` Python Tables API provides a general purpose toolkit for importing and processing succinct tree sequences. Interoperation with Python simulators is then straightforward. The implementation we benchmark here uses `pybind11` (`https://github.com/pybind/pybind11/`) to interface with the `fwdpp C++` API [Thornton, 2014]. No modifications were required to the `fwdpp` code base; rather, we simply need to bookkeep parent/offspring labels, and perform simple processing of the recombination breakpoints from each mating event to generate node and edge data. This information is then periodically copied to the `msprime` Tables API, where it is sorted and simplified.

Flexibility. To demonstrate the flexibility provided by the Tables API and provide an implementation that decouples forward simulation internals from transfer of data to `msprime`, we also implemented a version of the simulations described in "Simulation benchmarks" separately in Python, described in Appendix E. In this proof-of-concept implementation, the simulation engine (we use `simuPOP`, Peng and Kimmel [2005]) invokes callbacks at critical points of the simulation, and we infer nodes and edges from the information that is provided. Rows are appended to the tables one-by-one, and the tables are periodically sorted and simplified to control memory usage. Benchmarking results from this implementation are shown (alongside results from `fwdpp`) for simulations without selection in Appendix D: a relatively modest speedup of around $5\times$ is achieved, likely due to increased overhead.

Recording the pedigree in forwards time

To record the genealogical history of a forwards 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. These are naturally stored as the *nodes* and *edges* of a tree sequence. To demonstrate the idea, we write out in pseudocode how to run a neutral Wright–Fisher simulation 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.

We use $\mathcal{R}_U(A)$ to denote an element of the set A chosen uniformly at random (and all such instances are independent). Given a node table $\mathcal{N}$, the function $\mathcal{N}.\mathbf{addrow}(t)$ adds a new node to the table $\mathcal{N}$ with time t and returns the ID of this new node. Similarly, the function $\mathcal{E}.\mathbf{addrow}(\ell, r, p, c)$ adds a new edge $(\ell\text{eft}, r\text{ight}, p\text{arent}, c\text{hild})$ to the edge table $\mathcal{E}$. The function $\mathbf{simplify}(P, \mathcal{N}, \mathcal{E})$ (described below) simplifies the history stored in the tables $\mathcal{N}$ and $\mathcal{E}$ to the minimal information required to represent the genealogies of the list of node IDs P; after simplification the nodes appearing in P are relabeled $(0, 1, \ldots, |P| - 1)$. A step-by-step explanation follows the pseudocode.

Algorithm W. (*Forwards-time tree sequence*). Simulates a randomly mating population of N haploid individuals with chromosome of length L for T generations, and returns the node and edge tables ($\mathcal{N}$ and $\mathcal{E}$) recording the simulated history. In each generation, the current population is stored in P, while produced offspring are placed in P'. The tables are simplified every s generations, removing genealogical information from $\mathcal{N}$ and $\mathcal{E}$ irrelevant to the current population.

W1. [Initialisation.] Set $\mathcal{N} \leftarrow \mathbf{NodeTable}()$, $\mathcal{E} \leftarrow \mathbf{EdgeTable}()$, $t \leftarrow T$, and $j \leftarrow 0$. For $0 \leq k < N$, set $P_k \leftarrow \mathcal{N}.\mathbf{addrow}(T)$.

W2. [Generation loop head: new node.] Set $u \leftarrow \mathcal{N}.\mathbf{addrow}(t)$ and $P'_j \leftarrow u$.

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

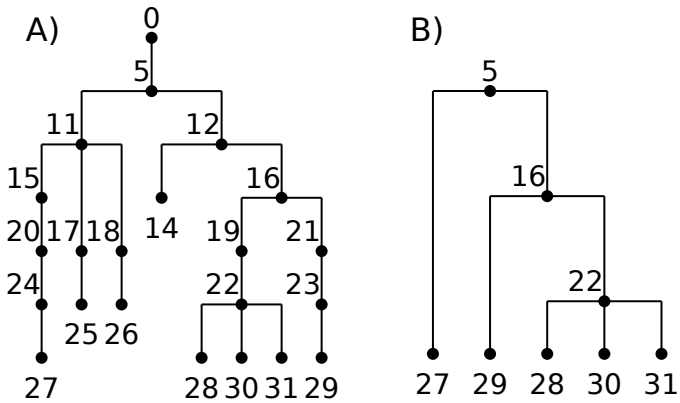


Figure 4: An example of a marginal genealogy from a Wright-Fisher simulation with $N = 5$. **(A)** the original tree including all intermediate nodes and dead-ends, and **(B)** the minimal tree relating all of the currently-alive individuals (27–31).

W3. [Choose parents.] Set $a \leftarrow \mathcal{R}_U(\{0, \ldots, N-1\})$, $b \leftarrow \mathcal{R}_U(\{0, \ldots, N-1\})$ and $x \leftarrow \mathcal{R}_U((0, L))$.

W4. [Record edges.] Call $\mathcal{E}.\mathbf{addrow}(0, x, P_a, u)$ and $\mathcal{E}.\mathbf{addrow}(x, L, P_b, u)$.

W5. [Individual loop.] Set $j \leftarrow j + 1$. If $j < N$ go to W2. Otherwise, if $t \bmod s \neq 0$ go to W7.

W6. [Simplify.] Call $\mathbf{simplify}(P', \mathcal{N}, \mathcal{E})$, and set $P'_k \leftarrow k$ for $0 \leq k < N$.

W7. [Generation loop.] Set $t \leftarrow t - 1$. If $t = 0$ terminate. Set $P \leftarrow P'$, $j \leftarrow 0$, and go to W2.

We begin in W1 by creating new node and edge tables, and setting our population P (a vector of N node IDs) to the initial population. This initial population is a set of N nodes with birth time T generations ago. We also initialise our generation clock t and individual index j. Step W2 replaces the jth individual (with node ID P_j) by creating a new node with birth time t (and ID u). In step W3 we determine the new node's ancestry by choosing two indexes a and b uniformly, giving us parental IDs P_a and P_b, and choose a chromosomal breakpoint x. We record the effects of this event by storing two new edges: one recording that the parent of node u from 0 to x is P_a, and another recording that the parent of u from x to L is P_b. Step W5 then iterates these steps for each of the N individuals for each generation. At the end of a generation, we then check if we need to simplify (done every s generations). If simplification is required, we do this in step W6 by calling the simplify function on the node and edge tables with the current set of population IDs P' as the samples. This updates the tables in-place to remove all redundant information, and remaps the specified sample IDs to $0, \ldots, N-1$ in the updated tables. Hence, we set our current population IDs to $0, \ldots N-1$ after simplify has completed. Step W7 loops these steps until the required number of generations have been simulated.

This algorithm records only topological information about the simulated genealogies, but it is straightforward to add mutational information. Mutations that occur during the simulation can be recorded by simply storing the node in which they first occur, the derived state, and (if not already present) the genomic position of the site at which it occurs. This allows selected mutations, that the forwards time simulation must generate, to be recorded in the tree sequence. Neutral mutations can be generated after the simulation has completed, thus avoiding the cost of generating the many mutations that are lost in the population. This is straightforward to do because we have access to the marginal genealogies.

Figure 4 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 current set of samples. Clearly there is a great deal of redundancy in the topological information output by the simulation. This redundancy comes from two sources. First, there are a large number of nodes in the tree that have only one child. In Algorithm W we do

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

not attempt to identify 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 coalescences within the marginal trees. The second source of redundancy in the (unsimplified) 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 4a, for example, leaves no ancestors in the current population, and so the entire path tracing back to its common ancestor with 27 is redundant.

Essential steps in tree sequence recording To record a tree sequence, including mutations, a simulator must record:

- the birth time of each individual in the Node Table,
- the genetic segments each individual inherits from its parents in the Edge Table, and
- the individual each new mutation occurs in, and the derived state in the Mutation Table, adding also to the Site Table if the site is previously unmutated.

Recall that here we mean "*haploid* individual", i.e., gamete. Before simplification, time must be translated to "time ago" if it is not already. (This was avoided in Algorithm W since there t denoted "time until the end of the simulation".) Furthermore, since **simplify**$(P', \mathcal{N}, \mathcal{E})$ results in the N individuals in P' being relabeled in `msprime`'s tables as $0, 1, \ldots, N-1$, there must be bookkeeping that keeps in sync the individual IDs as recorded by the simulator with the node IDs recorded in the tables.

Tree sequence simplification

It is desirable for many reasons to remove redundant information from a tree sequence. To formalize this: suppose that we are only interested in a subset of the nodes of a tree sequence (which we refer to as our 'samples'), and wish to reduce this input tree sequence to the smallest one that still completely describes the history of the specified samples, having the following properties:

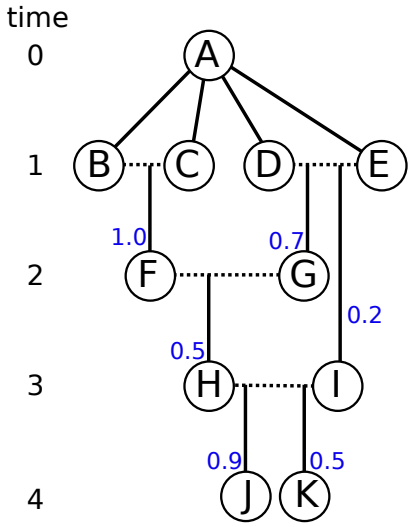
1. All marginal trees must match the subtree of the corresponding tree in the input tree sequence that is induced by the samples.
2. Within the marginal trees, all non-sample vertices must have at least two children (i.e., unary tree vertices are removed).
3. Any nodes and edges not ancestral to any of 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).

Simplification is essential not only for keeping the information recorded by forwards simulation manageable, but also is useful for extracting subsets of a tree sequence representing a very large dataset.

Our approach to simplification is based on Hudson's algorithm for simulating the coalescent with recombination [Hudson, 1983], paralleling the implementation in `msprime` [Kelleher et al., 2016]; an implementation in pseudocode is provided in Appendix A, and a python implementation as supplementary information. Conceptually, this works by (a) beginning by painting the chromosome in 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. Figure 5 shows an example tree sequence, before and after simplification, and Figure 6 depicts the "paint pot" state of the algorithm during the process of simplifying this tree sequence. Since the method

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

(A) The original tree sequence:



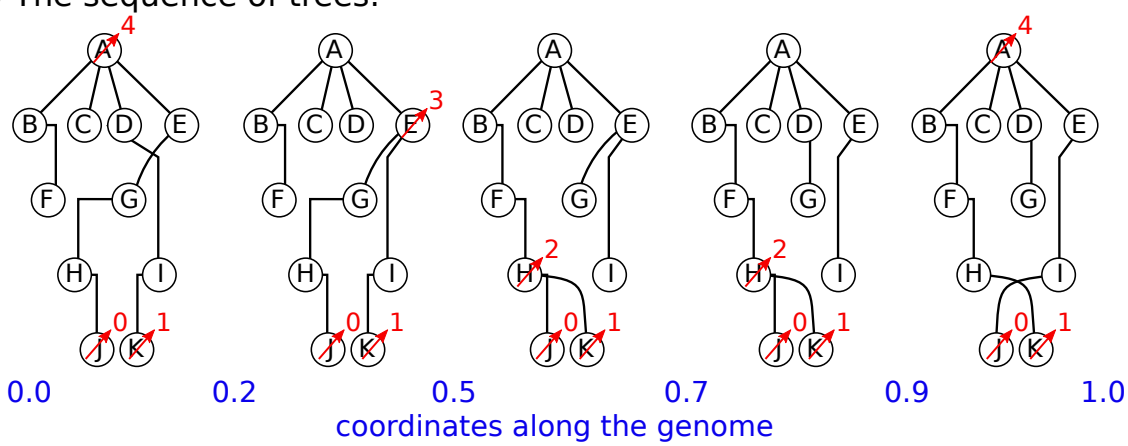
Nodes:

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

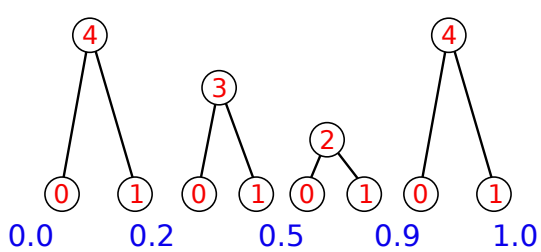
Edges:

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	A	E
0.0	1.0	B	F
0.0	0.7	E	G
0.7	1.0	D	G
0.0	0.5	G	H
0.5	1.0	F	H
0.0	0.2	D	I
0.2	1.0	E	I
0.0	0.9	H	J
0.9	1.0	I	J
0.0	0.5	I	K
0.5	1.0	H	K

(B) The sequence of trees:



(C) Simplified for J,K:



Nodes:

	label	time (ago)
J	0	0
K	1	0
H	2	1
E	3	3
A	4	4

Edges:

left	right	parent	child
0.5	0.9	2	0
0.5	0.9	2	1
0.2	0.5	3	0
0.2	0.5	3	1
0.0	0.2	4	0
0.0	0.2	4	1
0.9	1.0	4	0
0.9	1.0	4	1

Figure 5: An example of tree sequence simplification. **(A)** The augmented pedigree diagram on the left relates the ten homologous chromosomes of five diploid individuals (BC, DE, FG, HI, and JK) to each other and to a common ancestral chromosome (A); dotted lines connect the two chromosomes of each individual, and solid lines lead to the products of their meioses. The corresponding tables (right) have 11 node records (one for each chromosome) and 15 edge records (one for each distinctly inherited segment). Blue numbers denote crossing over locations – for instance, D and E were parents to G, who inherited the left 70% of the chromosome from E and the remainder from D. B, C, D, and E inherit clonally from A. **(B)** The five distinct trees found across the chromosome (blue numbers denote locations on the chromosome). Labels after simplification are shown in red (see text). **(C)** Tables recording the tree sequence after simplification with nodes J and K as samples. The mapping from labels in the forwards time simulation to nodes in the tree sequence is shown in red.

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

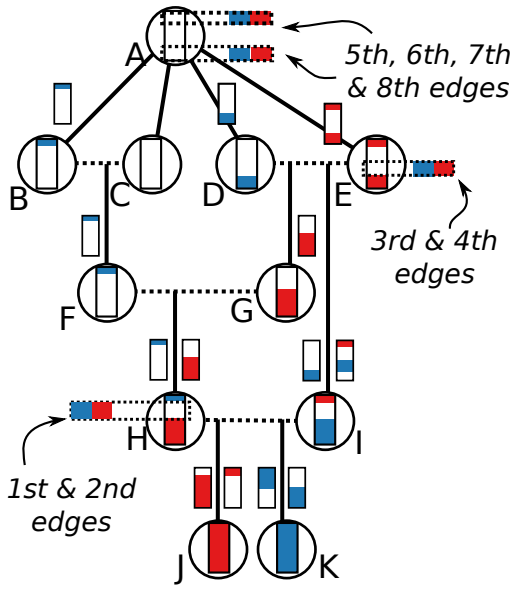


Figure 6: A depiction of each state of the simplification algorithm as it moves up through the embellished pedigree in the example of Figure 5A. Following the "paint pot" description in the text, we begin by coloring J and K's genomes in red and blue respectively, then trace how these colors were inherited back up through the pedigree until they coalesce. To aid in this, the smaller colored chromosomes on either side of each solid arrow show the bits inherited from each of the two parental chromosomes, with genomic position 0.0 on the bottom and 1.0 at the top. Each time a red and a blue segment overlap, a coalescence occurs, two edges are output, and we stop propagating that segment. For instance, both J and K inherit from H between 0.5 and 0.9, which resulted in the first two edges of the simplified table of Figure 5C. Later, both inherit from E between 0.2 and 0.5, along the paths J-H-G-E and K-I-E respectively, resulting in the next two edges.

begins with the samples and moves back through time, in the output tree sequence, the n samples will be numbered $0, 1, \ldots, n-1$ and subsequent nodes will be ordered by time since birth. This is seen in the red labels of Figure 5.

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 linked list 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 (and so u corresponds to a "color" in the description above). We also maintain a map from input nodes to lineages. Crucially, the time required to run the algorithm is linear in the number of edges of the input tree sequence.

Sequential simplification and prior history

Any simulation scheme that records data into tables, as Algorithm W does, has its genealogical history available at any time as a tree sequence. This has two additional advantages: First, simplification can be run periodically through the simulation, if we take the set of samples to be the entire currently alive population. This is important in practice 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`, providing an easy, efficient way to specify prior history. A natural question is now: how often should simplification occur? Limited testing (described in Appendix C) found that different simplification intervals affect run times by approximately 25%, with the lowest run time occurring when simplifying every 10^3 generations. Thus, there is a memory-versus-speed tradeoff –

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

simplifying more often would keep fewer extinct nodes and edges in memory.

Computational complexity

Figures 1 and 2 show that this method can dramatically improve simulation performance in practice – but, how does it perform in theory? Both computational time and storage space are depend mostly on the number of *mutations* and *edges* in the tree sequence. The key quantity to understand for this will be the total "area" of the tree sequence, which is the sum of the lengths of all ancestral genomic segments that some, but not all, of the present population has inherited. It can be found by summing the product of segment length (left minus right coordinates) and edge length (difference in birth times between parent and child), across all edges. This area is also equal to the sum of the total lengths of all marginal trees (i.e., the trees describing inheritance at each position on the genome), so can be computed as the sequence length multiplied by the mean marginal tree length. Since we analyze tree sequences arising from a Wright–Fisher model, statistical properties of a marginal tree are fairly well-described by coalescent theory. Similar arguments to those below go back at least to Watterson [1975], who also explicitly computed smaller order corrections relevant to whole-population genealogies of the Wright–Fisher model. For an introduction to coalescent theory, see Wakeley [2005].

First: how much memory do simplified tree sequences require? Consider a simulation of a Wright–Fisher population of N haploid individuals using Algorithm W for T generations. Since every chromosome inherits material from both parents, without simplification this would produce tables of NT nodes and $2NT$ edges. After simplification, we are left with the tree sequence describing the history of only the current generation of N individuals, back to either T generations ago, or their common ancestor, whichever comes first. The tree sequence must store edges describing the leftmost marginal tree, which requires at most $2N - 2$ edges. Then, each time the marginal tree changes along the sequence, four edges end and four new edges begin (except for changes affecting the root, which require fewer; see Kelleher et al. [2016]). Suppose that $\mathcal{T}_x$ is the marginal tree at genomic position x, and write $|\mathcal{T}_x|$ for the total length of the tree. For the tree at nearby position $x + dx$ to be different, there must have been a crossing-over between x and $x + dx$ in one of the $|\mathcal{T}_x|$ meioses that gave birth to those individuals. (Recall that these "individuals" are haploid.) Since we measure distance along the genome so that length is equal to the expected number of crossing-overs per generation, the expected distance until the next crossing-over is $1/|\mathcal{T}_x|$. If every crossing-over changed the marginal tree, then this would imply, for consistency, that the expected total number of times that the marginal tree changes along the genome is equal to the total area of the tree sequence (as defined above). Not every such crossing-over changes the marginal tree, but most do, so the total area of the tree sequence, multiplied by four, gives an upper bound on the expected number of edges in the tree sequence beyond those describing the leftmost tree. Since we are considering a chromosome of length 1, the expected total area is equal to the mean marginal tree length, as above.

If T is large relative to N, so that all marginal trees have a single root with high probability, then coalescent theory tells us that the expected total length of the branches of a marginal tree back to the most recent common ancestor is approximately $2N \log(N)$. Therefore, the tree sequence describing the entire population is expected to need no more than $2N + 8N \log(N)$ edges. 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. Therefore, we would need $O(N^2)$ space to store the complete history of the simulation, but only $O(N \log N)$ to store the history that is relevant to the extant population.

What if T is smaller: how many of the resulting $2NT$ edges are required after simplification? In other words, how fast does the information in the pedigree become irrelevant? Now, we need to compute the expected total length of all branches in a coalescent tree up until time T (or the common ancestor, whichever comes first). Again, coalescent theory tells us that the expected length of time for which a coalescent tree has k lineages is $2N/(k(k-1)) = 2N(1/(k-1) - 1/k)$ generations. Since the tree has k branches over this period, it is expected to contribute $2N/(k-1)$ to the total tree length. By summing over $n < k \leq N$, the N tips of a tree are expected to descend from only n lineages around $2(N/n - 1)$ generations ago. Inverting this relationship between time and number of roots implies that a marginal tree cut T units of time ago is expected to have around $r(T)$ roots, where $r(T) = 2N/(T + 2)$. The total tree length over this time is

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

$2N \sum_{k=r(T)+1}^{N} 1/(k-1) \approx 2N \log(N/r(T))$, since $\sum_{k=1}^{x} 1/k \approx \log(x)+\gamma$. This is a crude estimate for several reasons: first, we should not count the branch leading to the root of the tree (i.e., when $r(T) = 1$), and second, this does not account for the discrete nature of the Wright–Fisher model. Nonetheless, this leads as above to an upper bound on the number of edges of

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

This implies that the number of edges required to store the last T generations of history for the entire chromosome of a population of size N grows as $O(N \log T)$ – proportionally to N at first but rapidly tapering off. The bound holds up reasonably well in practice, as shown in Figure 7.

What about mutations? Forwards-time generation of infinite-sites mutations with total mutation rate per generation μ would produce around μNT mutations (and the same number of sites), simply because there was a total of NT meioses. Since mutations that are retained after simplification are precisely those that fall on the marginal trees, the number of mutations is proportional to the total area of the tree sequence. By the same argument as for the number of edges, this will result in around $\mu 2N \log(N)$ mutations. With $N = 2\times10^4$ and $T = 10N$, this implies that adding neutral mutations to the simplified tree sequence reduces the number of mutations that must be generated by a factor of 10,000. This could result in substantial time savings, even without considering the computational burden of propagating mutations forwards across generations.

Since each mutation is stored only as a single row in the mutation table, and at most one row in the site table, the space required for M mutations is $O(M)$; combined with the $O(N \log N)$ storage for edges and nodes of a simplified tree sequence, this implies that the full results of a simulation of N individuals having M mutations can be stored in $O(N \log N + M)$ space. In simulations of whole chromosomes, there are typically a small, bounded number of mutations per chromosome per generation, so M will be $O(N \log N)$ as well.

How does the computation *time* required for simplification scale? Simply because it must process each edge, the simplification algorithm is at least linear in the number of edges of the input tree sequence. Empirically the algorithm is exactly linear, as seen in Figure 7B, which shows the time required to simplify increasingly large subsets of a large tree sequence. When simplifying the result of a forwards-time sequence, the number of edges is the main contributing factor. Suppose on the other hand we want to simplify an already-minimal but large tree sequence with N nodes to a subsample of size n. How does the required time scale with n? In this case, the computation is dominated by the size of the output tree sequence, which grows with $\log(n)$, as shown in Figure 7C,

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 simulated population, 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 numerical tables, and have described a fundamental algorithm for simplification 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 Python for use in other code bases (a C API is also planned).

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 also allows highly efficient processing for downstream analysis. Efficient processing is possible because many 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, sequential tree algorithms similar to those in [Kelleher et al., 2016] can do this in roughly $O(n + m + t \log n)$ operations, where t

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

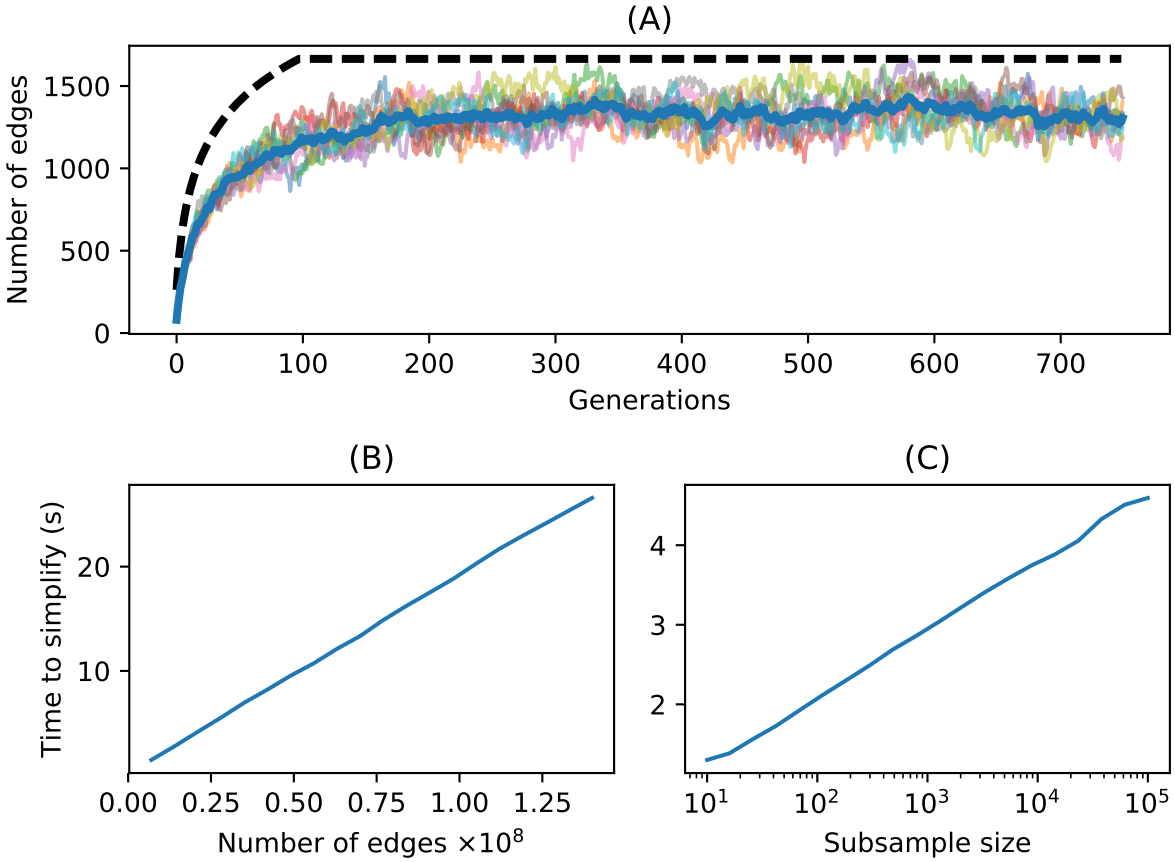


Figure 7: Time and space complexity of simplify. **(A)** Number of edges in the simplified tree sequence for 10 replicate Wright–Fisher simulations with $N = 100$ as a function of number of generations. Each line is one simulation, the heavy blue line gives the average, and the dashed line is the upper bound of equation (1). **(B)** Time required to simplify the first k edges of a large (4.2GiB) unsimplified tree sequence produced by a forwards-time simulation plotted against k. The time scales linearly with the number of input edges. **(C)** Time required to simplify the tree sequence resulting from a coalescent simulation of 500,000 samples of a 200 megabase human chromosome to a random subsample of n samples, plotted against n (note the log scale; the time scales logarithmically with n).

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

is the number of distinct trees. The `msprime` API provides a method to compute π among arbitrary subsets of the samples in a tree sequence, which took about 0.7 seconds when applied to an example simulation of 100 megabases of human-like sequence for 200,000 samples (about 500K sites). The corresponding numeric genotype matrix required about 95GiB of RAM, and calculating π took about 66 seconds with NumPy.

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. This may not even negatively affect realism, 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].

Other applications The methods described here for efficiently storing tree sequences may prove useful in other fields. We have focused on the interpretation of tree sequences as the outcome of the process of recombination, but in principle, we can efficiently encode any sequence of trees which differ by subtree-prune-and-regraft operations. Since each such operation requires a constant amount of space to encode, the total space required is $O(n + t)$ for t trees with n leaves [Kelleher et al., 2016]. For instance, the large numbers of large, correlated trees produced by MCMC samplers used in Bayesian phylogenetics [e.g., Drummond et al., 2012] might be compactly stored as a tree sequence, which would then allow highly efficient computation of properties of the posterior distribution.

In this article, we applied our methods for storing trees to the problem of pedigree recording in a forward-time simulation. However, the method applies to any simulation scheme generating nodes and edges. For example, one could use the methods described here to generate succinct tree sequences under coalescent processes not currently implemented in `msprime`, such as the coalescent with gene conversion [Wiuf and Hein, 2000], using the structured coalescent to model various forms of natural selection [Kaplan et al., 1988, 1989, Braverman et al., 1995], or the coalescent within a known pedigree. For such models, one could in principle generate tables of nodes and edges to be simplified in `msprime`. The resulting succinct tree sequence object would be in the same format as those generated by `msprime`'s built-in simulate function, and therefore compatible with existing methods for downstream analyses.

Another application of our methods would be the case of simulating coalescent histories conditional on known pedigrees. The standard description of the Wright-Fisher coalescent averages over pedigrees. However, conditional on a realized pedigree, the distribution of coalescent times in the recent past differs from that of the unconditional coalescent [Wakeley et al., 2012]. For populations with known pedigrees [*e.g.* Aguillon et al., 2017], it may be of use to simulate transmission along such pedigrees for the purpose of inference.

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". Etymological consensus [Liberman, 2014] has "pedigree" derived from the french "pied de grue" for the foot of a crane (whose branching pattern resembles the bifurcation of a single parent-offspring relationship). An analogous term for the embellished pedigree might then 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

We implemented simulations and the connection to `msprime` in `C++`, using `fwdpp` library functions and interface code using a continuum-sites model for both mutation and recombination. Simulations were run using `fwdpy11` (version 0.13.a0), a Python package based on `fwdpp` (version 0.5.7). The majority of results are presented based on a single-threaded implementation. However, we also implemented a parallelized version using Python's `queue.Queue` to run the simplification step in a separate Python thread. Our implementation allows a maximum of four simplification intervals to be in the queue at once. This parallelized version also performed fitness calculation in parallel using two threads of execution in C++.

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

Code for all simulations and figures is available at `https://github.com/petrelharp/ftprime_ms`. These made use of the GNU Scientific Library [version 1.16 Galassi, 2018], pybind11 [version 2.2.1 Jakob et al., 2016], and GCC (version 4.8.5). We ran all benchmarks on an Ubuntu Linux (version 16.04) system with two 2.6 GHz Intel E5-2650 CPU with hyperthreading enabled. We ran one simulation at a time and the machine was under minimal load otherwise. We used GNU parallel [Tange, 2011] to kill any simulation that did not finish within 72 hours, and the Linux `time` command to record run time and peak memory usage of each replicate.

Acknowledgements

Thanks to Gil McVean, Jared Galloway, Brad Shaffer, and Evan McCartney–Melstad for useful discussions. Work on this project was supported by funding from the Sloan Foundation and the NSF (under DBI-1262645) to PR; the Wellcome Trust (grant 100956/Z/13/Z) to Gil McVean; the NIH (R01GM115564) to KRT; and the USF&WS to H. Bradley Shaffer.

References

Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682. ACM, 2006.

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`.

Stepfanie M Aguillon, John W Fitzpatrick, Reed Bowman, Stephan J Schoech, Andrew G Clark, Graham Coop, and Nancy Chen. Deconstructing isolation-by-distance: The genomic consequences of limited dispersal. *PLoS Genet.*, 13(8):e1006911, August 2017.

Nicholas H Barton, Jerome Kelleher, and Alison M Etheridge. A new model for extinction and recolonization in two dimensions: quantifying phylogeography. *Evolution*, 64(9):2701–2715, 2010.

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`.

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

J M Braverman, R R Hudson, N L Kaplan, C H Langley, and W Stephan. The hitchhiking effect on the site frequency spectrum of DNA polymorphisms. *Genetics*, 140(2):783–796, June 1995.

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`.

Dask Development Team. *Dask: Library for dynamic task scheduling*, 2016. URL `http://dask.pydata.org`.

A J Drummond, M A Suchard, D Xie, and A Rambaut. Bayesian phylogenetics with BEAUti and the BEAST 1.7. *Mol Biol Evol*, 29(8):1969–1973, August 2012. doi: 10.1093/molbev/mss075. URL `https://www.ncbi.nlm.nih.gov/pubmed/22367748`.

M. et al Galassi. Gnu scientific library reference manual, 2018. URL `https://www.gnu.org/software/gsl/`.

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *International Symposium on Experimental Algorithms*, pages 326–337. Springer, 2014.

Robert C Griffiths. The two-locus ancestral graph. In *Selected Proceedings of the Sheffield Symposium on Applied Probability*, volume 18, pages 100–117, 1991.

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 `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.

R.R. Hudson. Gene genealogies and the coalescent process. *Oxford surveys in evolutionary biology*, 7(1):44, 1990. URL `http://web.eve.ucdavis.edu/pbg298/pdfs/Hudson_OxfordSurveysEvolBiol_1991.pdf`.

Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11 — seamless operability between c++11 and python, 2016. https://github.com/pybind/pybind11.

N L Kaplan, T Darden, and R R Hudson. The coalescent process in models with selection. *Genetics*, 120 (3):819–829, November 1988.

N L Kaplan, R R Hudson, and C H Langley. The "hitchhiking effect" revisited. *Genetics*, 123(4):887–899, December 1989.

J Kelleher, AM Etheridge, and NH Barton. Coalescent simulation in continuous space: Algorithms for large neighbourhood size. *Theoretical population biology*, 95:13–23, 2014.

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.

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

Paul Marjoram and Jeff 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.

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

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

Claudia Neuhauser and Stephen M Krone. The genealogy of samples in models with selection. *Genetics*, 145(2):519–534, February 1997.

Badri Padhukasahasram, Paul Marjoram, Jeff D. Wall, Carlos D. Bustamante, and Magnus 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`.

Bo Peng and Marek Kimmel. simupop: a forward-time population genetics simulation environment. *Bioinformatics*, 21(18):3686–3687, 2005.

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.

Ole Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011. URL `http://www.gnu.org/s/parallel`.

The HDF Group. Hierarchical Data Format, version 5, 1997-2018. http://www.hdfgroup.org/HDF5/.

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. *Coalescent Theory, an Introduction*. Roberts and Company, Greenwood Village, CO, 2005. URL `http://www.coalescentheory.com/`.

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.

John Wakeley, Léandra King, Bobbi S Low, and Sohini Ramachandran. Gene genealogies within a fixed pedigree, and the robustness of kingman's coalescent. *Genetics*, 190(4):1433–1445, April 2012.

Stéfan van der Walt, S Chris Colbert, and Gael Varoquaux. The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

G A Watterson. On the number of segregating sites in genetical models without recombination. *Theor Popul Biol*, 7(2):256–276, April 1975. URL `https://www.ncbi.nlm.nih.gov/pubmed/1145509`.

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`.

Peter R Wilton, Shai Carmi, and Asger 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`.

Carsten Wiuf and Jotun 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`.

Carsten Wiuf and Jotun 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`.

Carsten Wiuf and Jotun Hein. The coalescent with gene conversion. *Genetics*, 155(1):451–462, May 2000.

Sewall Wright. Evolution in mendelian populations. *Genetics*, 16(2):97–159, 1931.

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

A Simplification algorithm

Here we provide a concrete implementation of the simplification algorithm. The algorithm uses a few simple structures besides the node and edge tables discussed earlier. As the algorithm needs to track the mapping between input and output node IDs over specific genomic intervals, we use a **Segment**() type to track these individual mappings. Thus, a segment x has three attributes: x.left and x.right represent the genomic interval in the usual way, and x.node represents the output node ID that is assigned to this interval. These output node IDs correspond to colors in the "paint pot" analogy. The per-interval segments are assigned to input IDs via the $\mathcal{A}$ mapping, which is the main state required by the algorithm. Each element $\mathcal{A}[u]$ is a list of ancestral segments, describing the output node IDs mapped to input node ID u over a set of intervals. For each input node, we need to process the ancestral segments to find overlaps (and hence coalescences), and to update $\mathcal{A}$. We do this using a **PriorityQueue**() type, which maintains a list of ancestral segments sorted by left coordinate. For a priority queue Q, the operation $Q.\mathbf{push}(x)$ inserts the segment x into the queue, and $Q.\mathbf{pop}()$ removes and returns the segment in Q with the smallest left coordinate. To obtain the segment currently in the queue with the smallest left value we write $Q[0]$. Finally, note that to keep the exposition simple, the algorithm described here requires that the input node table is sorted by time since the parent's birth; but the version implemented in `msprime` makes different sortedness assumptions (described in the documentation).

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 $\mathcal{E}_O$.

S1. [Initialisation.] Set $\mathcal{A}_u \leftarrow \mathbf{List}()$ for $0 \leq u < |\mathcal{N}_I|$. Then, for each u in S, set $v \leftarrow \mathcal{N}_O.\mathbf{addrow}(\mathcal{N}_I[u])$ and call $\mathcal{A}[u].\mathbf{append}(\mathbf{Segment}(0, L, v))$. Then set $Q \leftarrow \mathbf{PriorityQueue}()$ and $u \leftarrow 0$.

S2. [Select parent edges.] Set $E \leftarrow \{e \in \mathcal{E}_I : e.\text{parent} = u\}$. Set $v \leftarrow -1$. If $|E| = 0$, go to S9.

S3. [Insert edge/ancestry intersections.] For each $e \in E$ and each $x \in \mathcal{A}[e.\text{child}]$, if $x.\text{right} > e.\text{left}$ and $e.\text{right} > x.\text{left}$, set $\ell \leftarrow \max(x.\text{left}, e.\text{left})$, $r \leftarrow \min(x.\text{right}, e.\text{right})$, $y \leftarrow \mathbf{Segment}(\ell, r, x.\text{node})$ and call $Q.\mathbf{push}(y)$.

S4. [Find segments with minimum left coordinate.] Set $\ell \leftarrow Q[0].\text{left}$, $r \leftarrow L$ and $X \leftarrow \text{List}()$. Then, while $|Q| > 0$ and $Q[0].\text{left} = \ell$, set $x \leftarrow Q.\mathbf{pop}()$, $r \leftarrow \min(r, x.\text{right})$ and call $X.\mathbf{append}(x)$. Afterwards, if $|Q| > 0$, set $r \leftarrow \min(r, Q[0].\text{left})$. If $|X| > 1$ go to S6.

S5. [No overlapping segments.] Set $x \leftarrow X[0]$ and $\alpha \leftarrow x$. If $|Q| > 0$ and $Q[0].\text{left} < x.\text{right}$, set $\alpha \leftarrow \mathbf{Segment}(x.\text{left}, Q[0].\text{right}, x.\text{node})$, $x.\text{left} \leftarrow Q[0].\text{left}$ and call $Q.\mathbf{push}(x)$. Go to S8.

S6. [Overlap; new output node.] If $v = -1$, set $v \leftarrow \mathcal{N}_0.\mathbf{addrow}(\mathcal{N}_I[u])$.

S7. [Record edges.] Set $\alpha \leftarrow \mathbf{Segment}(\ell, r, v)$. Then for each $x \in X$ call $\mathcal{E}_O.\mathbf{addrow}(l, r, v, x.\text{node})$ and, if $x.\text{right} > r$, set $x.\text{left} \leftarrow r$ and call $Q.\mathbf{push}(x)$.

S8. [Left coordinate loop.] Call $\mathcal{A}[u].\mathbf{append}(\alpha)$. If $|Q| > 0$ return to S4.

S9. [Parent loop.] Set $u \leftarrow u + 1$. If $u < |\mathcal{N}_I|$, go to S2.

The algorithm begins in S1 by allocating an empty list to store ancestral segments for each of the $|\mathcal{N}_I|$ input nodes, and then creating the initial state for the samples in S. For each input sample with node ID u, we add a row in the output node table $\mathcal{N}_O$, and create a mapping for this new node v. After this initial state has been created, we then allocate our priority queue Q, and set the current input ID u to 0. Recall that the function $\mathcal{N}.\mathbf{addrow}(t)$ adds a new node to the node table $\mathcal{N}$ with birth time t, and returns the ID (i.e., index) of that new node in $\mathcal{N}$. So, the operation $v \leftarrow \mathcal{N}_O.\mathbf{addrow}(\mathcal{N}_I[u])$ looks up the birth time for input node ID u in $\mathcal{N}_I$, adds a new node with this time to $\mathcal{N}_O$, and stores the resulting (output) node ID in v.

To process ancestral information moving back up through the pedigree, steps S2–S9 consider each node u in turn (note we are assuming that nodes are ordered by time-since-birth). In S2 we first find all edges in

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

the input where the parent is equal to our current input ID u, and then set the corresponding output ID v to -1. Then, in S3 we find all intersections between these input edges and the mapped ancestry segments of their child nodes. For each of these intersections we create a new segment y and insert it into the priority queue. (In the paint pot analogy, this stores in Q all segments of color that must be transferred to the ancestor with input ID u.)

After finding all overlaps and filling the priority queue with the resulting ancestry segments, we then need to process these segments. Steps S4–S8 loop over the segments in the queue, considering each in turn until the queue is empty. Step S4 builds a list X of all segments covering the current left-most coordinate (along with some bookkeeping to keep track of where our next right coordinate r is), removing them from Q. If there is only one element in X this means that we have no overlapping ancestral segments at this coordinate and we proceed to S5.

In this case, only a single segment covers the leftmost region of the segments, so there is no change in the mapping between input and output IDs, and the ancestry segment α is not changed (this is assigned to $\mathcal{A}_u$ in S8). (The color in this segment of the offspring can simply be transferred to the parent.) However, if other segments in Q intersect with x (since x is the current left-most segment, these have left coordinate less then x.right), then we can only transfer the portion of x up until it overlaps with other segments. Therefore, we change the left coordinate of x to this coordinate and reinsert it into Q where it will be processed again later, effectively cutting off (and propagating) the leftmost segment of x. After doing this, we proceed to S8, update the ancestry mapping for the input node u, and loop back to S4 if Q is not empty.

On the other hand, if there is more than one segment in X, we know that an overlap among the ancestral segments has occurred and we must update the output nodes and edges accordingly. In step S6 we first allocate a new output node v for input node u, if it has not been done already (we may have many overlapping segments along the genome). Step S7 then continues by creating a new ancestry segment α mapping the current interval to the output node v. We then iterate over all segments in X, adding the appropriate output edges and editing and reinserting the ancestry segment x into the priority queue if necessary, as before.

B Memory usage with selection

The peak RAM use in gigabytes (GB) is shown for the simulations with selection in Figure B1. For our parameters, which involved simplifying every 10^3 generations, simulation while recording pedigrees uses more RAM than a standard simulation. This is because with pedigree recording, a non-recombinant individual still requires one new edge and two new nodes to be allocated. Without pedigree recording, no new memory is allocated for such an offspring by `fwdpp`. Thus, pedigree recording accumulates a large amount of data in RAM until simplification. This extra RAM consumption is the trade off for reduced run times and the degree of extra RAM needed can be reduced by simplifying more often. The simulations where we run the simplification step in a separate thread require even more RAM (dashed line in Figure B1) because up to four sets of unsimplified data are stored in RAM. In practice, the RAM consumption can be minimized by simplifying more often.

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

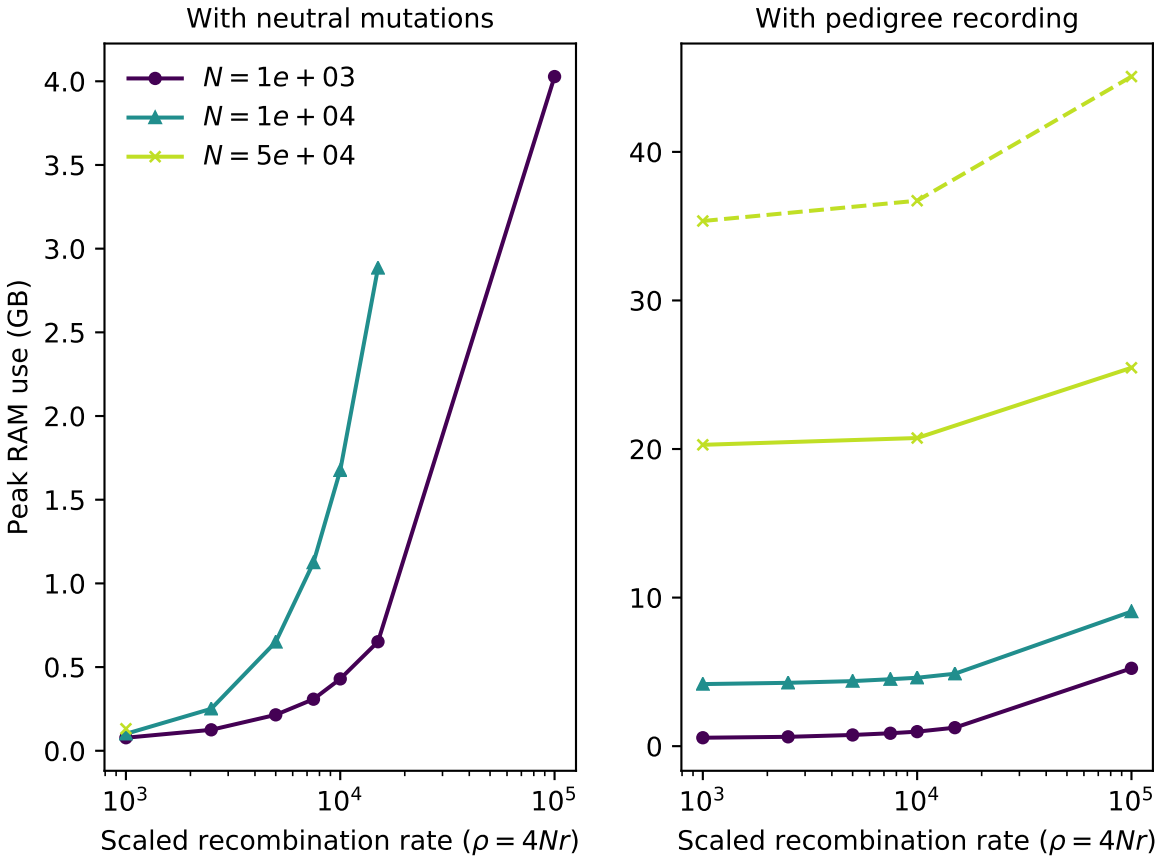


Figure B1: Peak RAM use in gigabytes (GB) for simulations with selection. The plots are from the same data as in Figure 1. Note that the y-axis scales differ considerably across panels.

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

C Effect of simplification interval on run time and memory use

We performed a limited set of simulations to explore the effect of the simplification interval on run times and memory use. Figure C2 shows that run times increase for simplification intervals less than 100 generations. For intervals longer than 100 generations, run times are very similar, with no appreciable difference between 10^3 and 10^4 generations. As expected, the simplification interval has a near-linear effect on peak RAM use (Figure C3). Taken together Figures C2 and C3 suggest that one can tune the simplification interval to available RAM with little effect on run times provided that simplification doesn't occur too often.

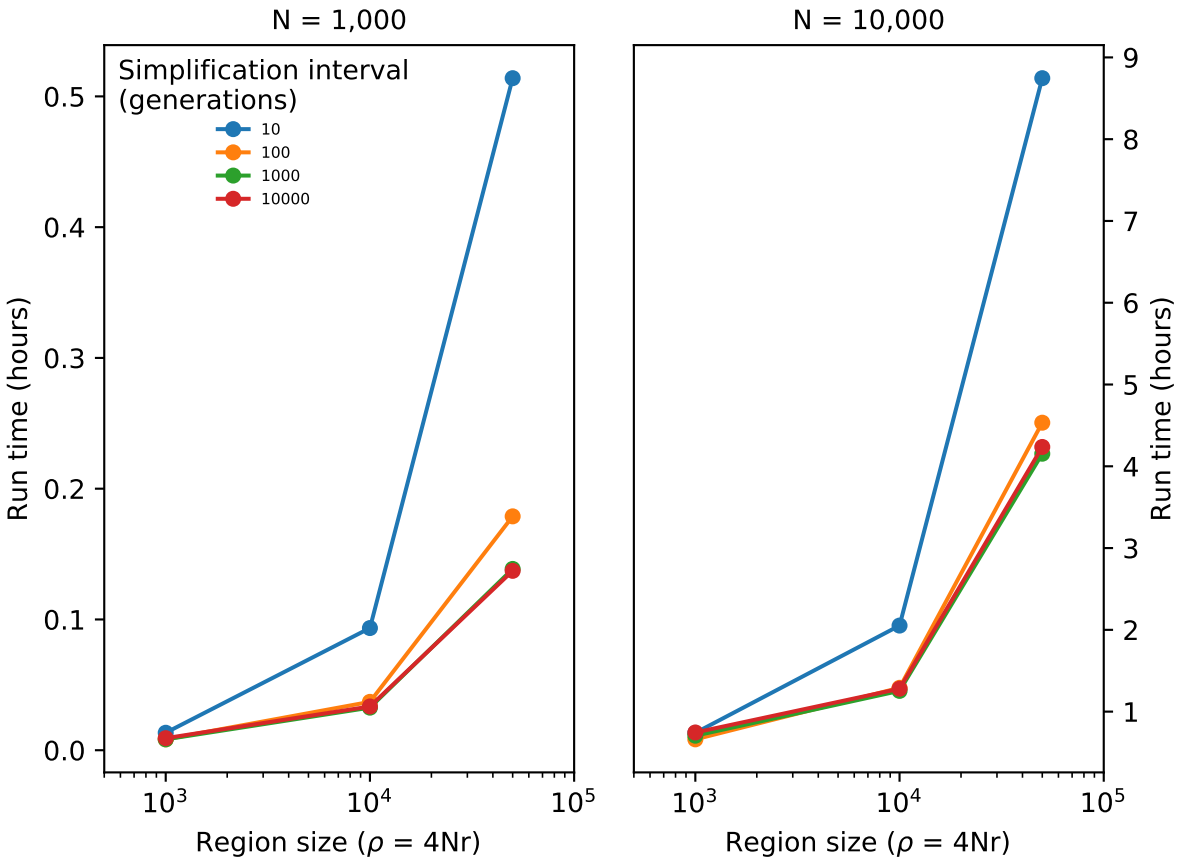


Figure C2: The effect of simplification interval on run times. A single replicate was done for the parameters shown in the figure. These simulations were done with the same selection parameters as in Figure 1.

D Timing and memory use with no selection

We also performed a more limited set of simulations without natural selection. For these we used `C++` as in the main text, but also provide results from our proof-of-concept implementation that uses `simuPOP` (see Appendix E).

The total run times are shown in Figure D1 and for `C++` show the same qualitative behavior as simulations with selection (Figure 1). The relative improvement due to pedigree tracking is again substantial in these simulations (Figure D2). In fact, we see more of a benefit to pedigree tracking here with `fwdpy11` than we

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

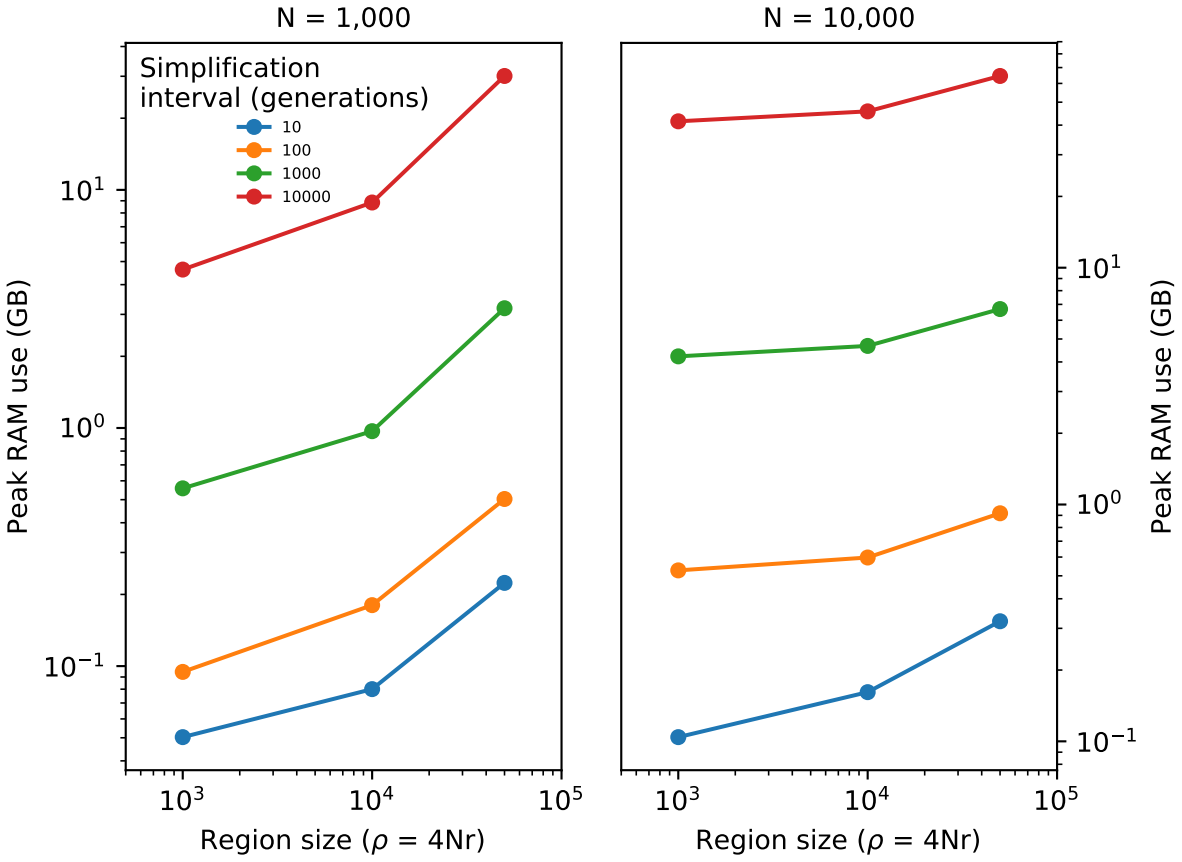


Figure C3: Peak RAM use for the simulations shown in Figure C2.

did with selection (Figure 2). The reason is that the fitness function exits close to instantly in simulations without selection that are based on `fwdpp`, meaning that `fwdpy11` is doing little more than generating random numbers and book-keeping in Figures D1 and D2. For `simuPOP`, the results show a qualitatively similar pattern but (1) the simulations for $N = 10000$ or greater did no finish in the 72 hour limit or exceed memory limits (more than 500 GB), and (2) the magnitude of the speedup is much less. The total RAM use for simulations without selection is shown in Figure D3.

E The `simuPOP` implementation

To use the methods with `simuPOP`, we created a python module (available at `https://github.com/ashander/ftprime`) that manages communication between the forward simulator and `msprime`. We implemented simulations in a Python script, usable as a command-line utility, taking input parameters for population size N and scaled recombination rate ρ. As for the `fwdpp` simulations above, we set scaled mutation rate $\theta = \rho$, used a simplify interval of 1000 generations and simulated for $T = 10N$ generations in each simulation.

We coupled `simuPOP` to `msprime` using a Python class `RecombCollector` that 1) tracks time, and 2) collects and parses recombination information provided by the `Recombinator` class of `simuPOP`, adding them to a **NodeTable**() and **EdgeTable**() upon which simplification is performed every 1000 generations (using `msprime.sort_tables()` and `msprime.simplify_tables`). The tables are initialized using

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

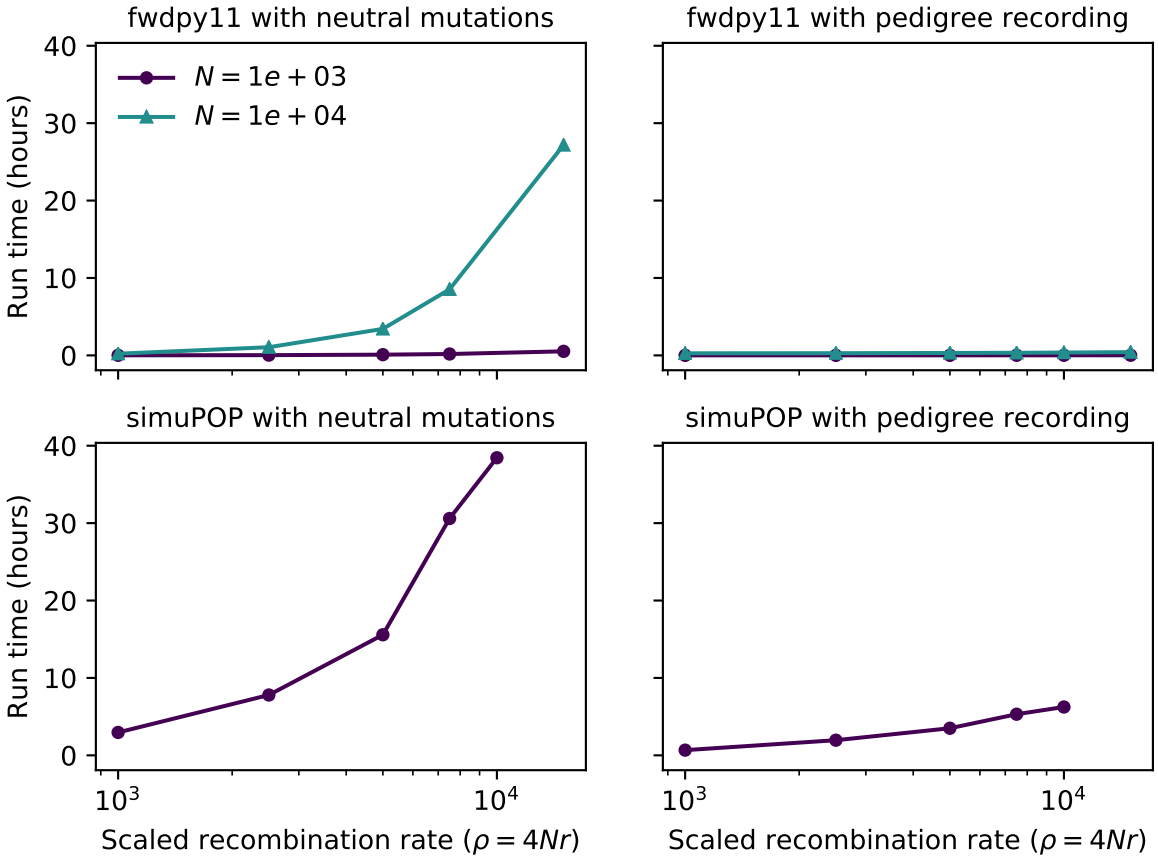


Figure D1: Total run time for a single simulation replicate as a function of region length, measured as the scaled recombination parameter $\rho = 4Nr$. The simulations here involve no natural selection.

the `TreeSequence` returned from `msprime.simulate(`$2N$`, Ne=`N`, recombination_rate=`$r/2$`, length=`n`)`. The `simuPOP.Population()` is initialized and tagged with individual IDs, a mapping of these IDs to the **NodeTable**() IDs is passed to an instance of `RecombCollector` and used internally to update the tables. The `simuPOP.Population()` is then evolved. Before mating: the internal time of the `RecombCollector` instance is updated. During mating: male and female parents are drawn randomly, recombination data is output by `simuPOP` to the `RecombCollector` instance. After mating: every 1000 generations the tables are simplified.

`RecombCollector` uses the Python class `ARGrecorder` internally to add data to the tables and perform simplification. This class contains the **NodeTable**() and **EdgeTable**() and also the mapping from (haploid) ids to ids in the tables. It also provides methods (`add_individual` and `add_record`) to append data to the tables. The only functions of `RecombCollector` not delegated to this internal class are computing haploid IDs from `simuPOP`'s diploid IDs and parsing recombination data into data suitable for use with use with `add_individual` and `add_record`; these functions are used to stream data into the tables as it is received by an instance of `RecombCollector`.

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

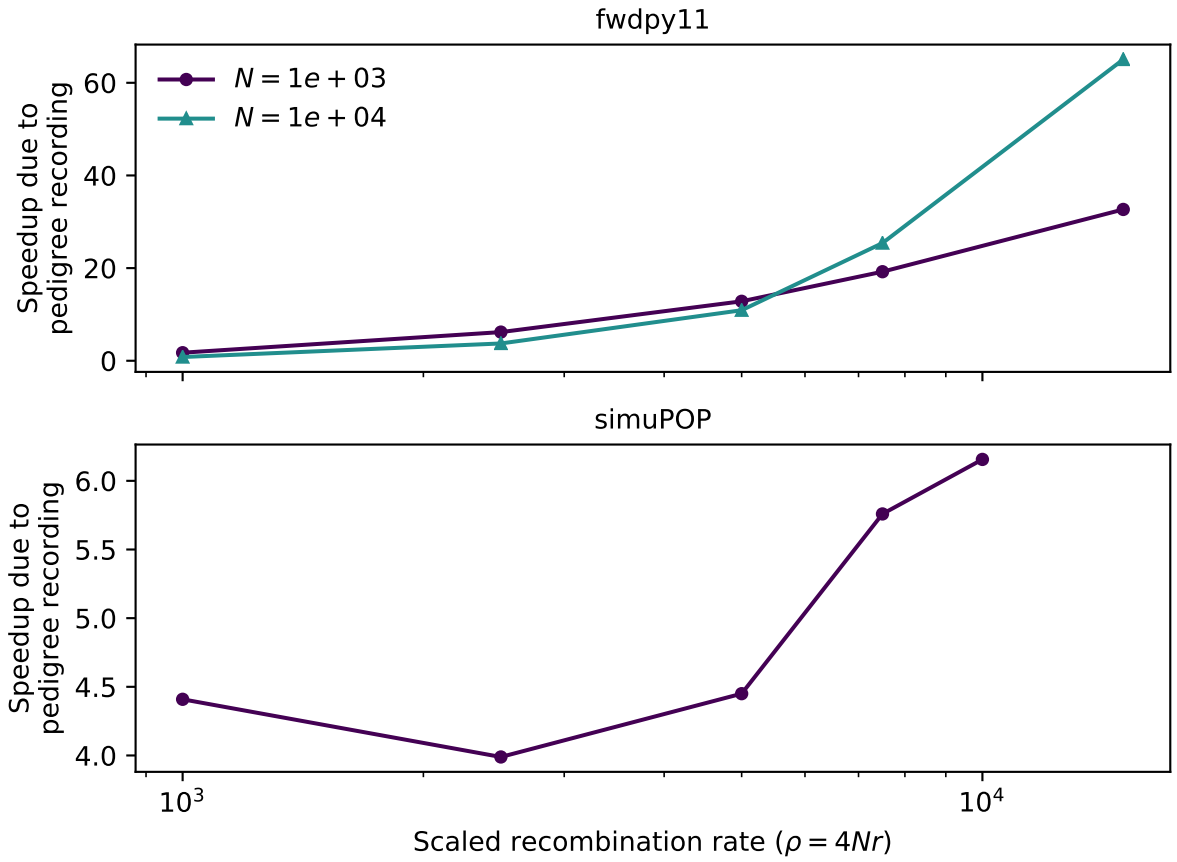


Figure D2: Relative performance improvement due to pedigree tracking for simulations without selection. Data points are taken from Figure D1 and show the ratio of run times tracking neutral variation over the run times tracking the pedigree.

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

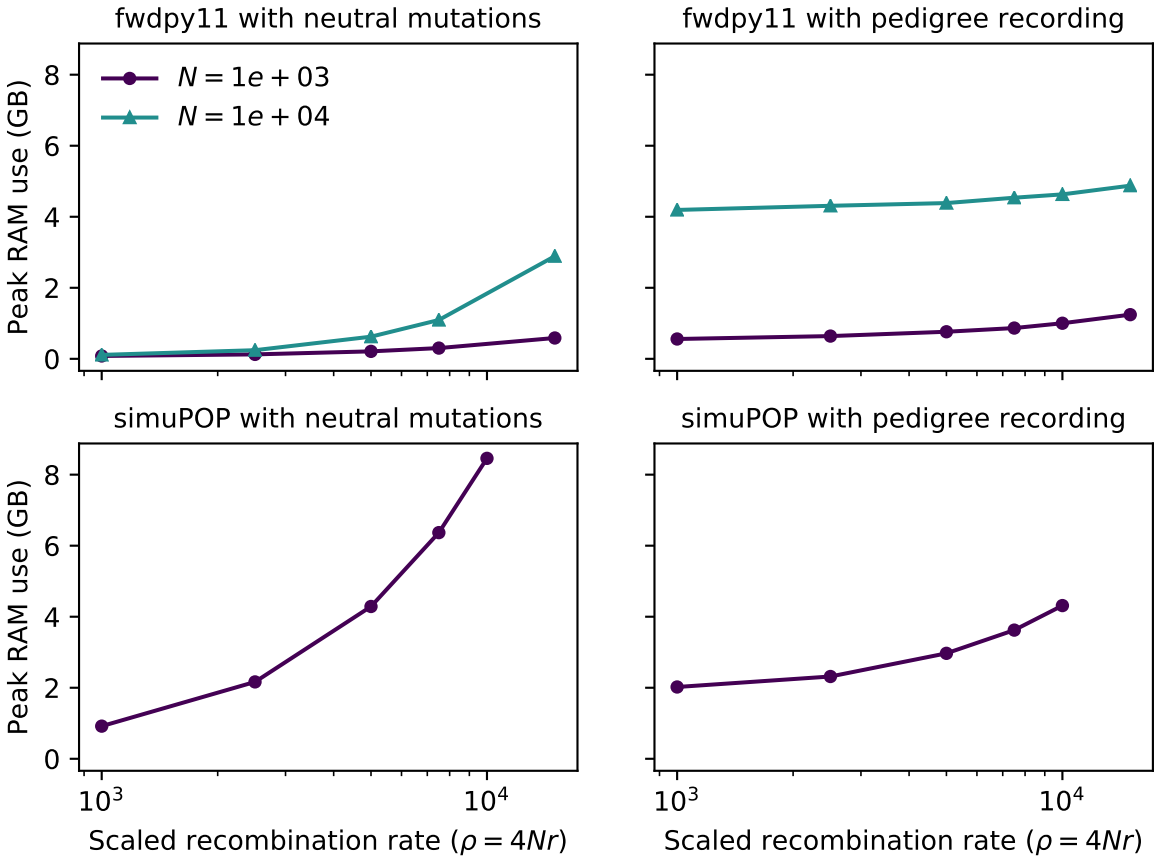


Figure D3: Peak RAM use in gigabytes (GB) for simulations without selection. The plots are from the same data as in Figure D1.

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

Python implementation of simplify (supplementary information)

```
"""
Python implementation of Algorithm S.

The example works by generating an initial TreeSequence
for a sample of 10 haplotypes using msprime.  We then
simplify the node/edge table in that TreeSequence with
respect to the first three samples.
"""
import heapq
import numpy as np

import msprime


class Segment(object):
    """
    An ancestral segment mapping a given node ID to a half-open genomic
    interval [left, right).
    """
    def __init__(self, left, right, node):
        assert left < right
        self.left = left
        self.right = right
        self.node = node

    def __lt__(self, other):
        # We implement the < operator so that we can use the heapq directly.
        # We are only interested in the left coordinate.
        return self.left < other.left

    def __repr__(self):
        return repr((self.left, self.right, self.node))


def simplify(S, Ni, Ei, L):
    """
    This is an implementation of the simplify algorithm described in Appendix A
    of the paper.
    """
    No = msprime.NodeTable()
    Eo = msprime.EdgeTable()
    A = [[] for _ in range(len(Ni))]
    Q = []

    for u in S:
        v = No.add_row(time=Ni.time[u], flags=1)
        A[u] = [Segment(0, L, v)]

    for u in range(len(Ni)):
```

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

```
        for e in [e for e in Ei if e.parent == u]:
            for x in A[e.child]:
                if x.right > e.left and e.right > x.left:
                    y = Segment(max(x.left, e.left), min(x.right, e.right), x.node)
                    heapq.heappush(Q, y)
        v = -1
        while len(Q) > 0:
            l = Q[0].left
            r = L
            X = []
            while len(Q) > 0 and Q[0].left == l:
                x = heapq.heappop(Q)
                X.append(x)
                r = min(r, x.right)
            if len(Q) > 0:
                r = min(r, Q[0].left)

            if len(X) == 1:
                x = X[0]
                alpha = x
                if len(Q) > 0 and Q[0].left < x.right:
                    alpha = Segment(x.left, Q[0].left, x.node)
                    x.left = Q[0].left
                    heapq.heappush(Q, x)
            else:
                if v == -1:
                    v = No.add_row(time=Ni.time[u])
                alpha = Segment(l, r, v)
                for x in X:
                    Eo.add_row(l, r, v, x.node)
                    if x.right > r:
                        x.left = r
                        heapq.heappush(Q, x)

            A[u].append(alpha)

    # Sort the output edges and compact them as much as possible into
    # the output table. We skip this for the algorithm listing as it's pretty mundane.
    # TODO replace this with a calls to squash_edges() and sort_tables()
    E = list(Eo)
    Eo.clear()
    E.sort(key=lambda e: (e.parent, e.child, e.right, e.left))
    start = 0
    for j in range(1, len(E)):
        condition = (
            E[j - 1].right != E[j].left or
            E[j - 1].parent != E[j].parent or
            E[j - 1].child != E[j].child)
        if condition:
            Eo.add_row(E[start].left, E[j - 1].right, E[j - 1].parent, E[j - 1].child)
            start = j
```

bioRxiv preprint doi: https://doi.org/10.1101/248500; this version posted January 26, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

```
    j = len(E)
    Eo.add_row(E[start].left, E[j - 1].right, E[j - 1].parent, E[j - 1].child)

    return msprime.load_tables(nodes=No, edges=Eo)


def verify():
    """
    Checks that simplify() does the right thing, by comparing to the implementation
    in msprime.
    """
    for n in [10, 100, 1000]:
        ts = msprime.simulate(n, recombination_rate=1, random_seed=1)
        nodes = ts.tables.nodes
        edges = ts.tables.edges
        print("simulated for ", n)

        for N in range(2, 10):
            sample = list(range(N))
            ts1 = simplify(sample, nodes, edges, ts.sequence_length)
            ts2 = ts.simplify(sample)

            n1 = ts1.tables.nodes
            n2 = ts2.tables.nodes
            assert np.array_equal(n1.time, n2.time)
            assert np.array_equal(n1.flags, n2.flags)
            e1 = ts1.tables.edges
            e2 = ts2.tables.edges
            assert np.array_equal(e1.left, e2.left)
            assert np.array_equal(e1.right, e2.right)
            assert np.array_equal(e1.parent, e1.parent)
            assert np.array_equal(e1.child, e1.child)


if __name__ == "__main__":
    verify()

    # Generate initial TreeSequence
    ts = msprime.simulate(10, recombination_rate=2, random_seed=1)
    nodes = ts.tables.nodes
    edges = ts.tables.edges

    # Simplify nodes and edges with respect to the following samples:
    sample = [0, 1, 2]
    ts1 = simplify(sample, nodes, edges, ts.sequence_length)
    print(ts1.tables)
```