Karlsruhe Institute of Technology

Re-Engineering Genomic Tree Sequence Inference Algorithms

Master's Thesis of

Johannes Hengstler

At the Department of Informatics
Institute of Theoretical Informatics

Reviewer:	Prof. Dr. Alexandros Stamatakis
Second reviewer:	Prof. Dr. Peter Sanders
Advisor:	M.Sc. Lukas Hübner

01.02.2024 – 01.08.2024

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I have followed the by-laws to implement scientific integrity at KIT.

Karlsruhe, 01.08.2024

. . . . . .

(Johannes Hengstler)

Abstract

In the field of population genetics, the driving forces of evolution within species can be studied with trees. Along a genome, each tree describes the local ancestries of a small genomic region. Together, those trees form a tree sequence that describes the ancestry of a population at every site of the sequence. Inferring tree sequences for whole genomes with many haplotype samples is a computationally expensive task, however.

The state-of-the-art tool to infer tree sequences is *tsinfer*, which infers ancestries for human chromosomes from 5000 samples within a few hours. The tool has the capability to parallelize the computation, but we identify a structure in the input data that limits its parallelizability. We propose a novel parallelization scheme aiming to improve scaling at high thread counts, independently of this structure. Furthermore, we propose several optimizations for the inference algorithm, improving cache efficiency and reducing the number of operations per iteration.

We provide a proof-of-concept implementation, and compare the computation speed of our implementation and *tsinfer*. When inferring ancestries for the *1000 Genomes Project*, our implementation is consistently faster by a factor of 1.9 to 2.4. Additionally, depending on the choice of parameters, our parallelization scheme scales better between 32 and 96 cores, improving its speed advantage, especially at higher core counts. In phases where our novel parallelization scheme does not apply, our optimizations still improve the runtime by a factor of 2.2. As available genomic data sets are growing rapidly in size, our contribution decreases the computation time and enables better parallelization, allowing the processing of larger data sets in reasonable time frames.

Zusammenfassung

Die Disziplin der Populationsgenetik untersucht die treibenden Kräfte der Evolution mit Bäumen. Entlang eines Genoms beschreibt jeder solche Baum die lokale Abstammung einer kleinen genomischen Region. Zusammen ergeben diese Bäume eine Baumsequenz, die den Stammbaum einer Population an jeder Site des Genoms enthält. Allerdings ist der Rechenaufwand für die Inferenz von Baumsequenzen für ganze Genome teuer, wenn viele Haplotypen auf einmal in den Stammbaum eingefügt werden sollen.

Das state-of-the-art Programm, um eine solche Baumsequenz zu berechnen, ist *tsinfer*, welches Stammbäume für menschliche Chromosomen mit 5000 Eingabesequenzen in wenigen Stunden berechnet. Das Programm kann die Berechnung parallelisieren, aber wir zeigen eine Struktur in den Eingabedaten auf, welche den Grad der Parallelisierung begrenzt. Wir schlagen eine alternative Methode der Parallelisierung vor, die das Skalierungsverhalten bei hohen Threadzahlen unabhängig von dieser Struktur verbessert. Darüber hinaus optimieren wir den Inferenzalgorithmus, indem wir die Cache-Effizienz einiger Datenstrukturen und die Anzahl an Operationen pro Iteration verringern.

Wir implementieren unseren veränderten Algorithmus und vergleichen diesen mit *tsinfer*. Unsere Implementation ist auf Daten des *1000 Genomes Projects* konsequent um einen Faktor von 1.9 bis 2.4 schneller. Je nach Wahl der Parameter skaliert unsere Parallelisierungsmethode außerdem zwischen 32 und 96 Prozessorkernen besser als *tsinfer*, und baut so den Geschwindigkeitsvorteil vor allem bei hohen Kernzahlen aus. In den Phasen der Inferenz, in denen unsere Methode der Parallelisierung nicht anwendbar ist, ist unser Algorithmus trotzdem um einen Faktor 2.2 schneller. Unser Beitrag verringert die Berechnungszeit und verbessert die Parallelisierung der Inferenz von Baumsequenzen. Diese Verbesserungen erlauben trotz des enormen Wachstums von Genomdatensätzen die Inferenz größerer Eingaben in angemessenen Zeiträumen.

Contents

1 Introduction

1.1 Motivation

Trees have long been one of the most fundamental data models in biology to describe evolutionary processes [Gor87]. For example, the discipline of phylogenetics studies the genesis of species, in which trees are used to model their evolutionary history. In 1866, Ernst Haeckel first sketched Darwin's ideas about trees to explain relationships between plants [HWL17]. Today, modern phylogenetic trees comprise hundreds to thousands of branches, computationally derived from hundreds of genomes [Sti+24]. With the advent of large-scale sequencing technologies, and the subsequent explosion in available genomic data, the interest in describing the genetic variation of different individuals within the same species (a *population*) also increased [JSSM21]. For sexually reproducing species, however, a single tree is insufficient to model the full ancestry of their genome [Wu11].

In the processes modeled by phylogenetics, genomes branch off each other through evolutionary processes, and then remain unaffected by other genomes. Therefore, each sequence has exactly one predecessor, which itself has no more than one predecessor. This relationship can naturally be modeled with a tree. In contrast, sexual reproduction is characterized by recombination of genetic information: This process breaks the ancestry of a sequence into intervals, divided by *recombination points*, where two analog sequences with different ancestries recombine. Consequently, accurately explaining the ancestry of multiple genomes within a population requires a network (commonly called *Ancestral Recombination Graph*) that allows modeling distinct ancestries for different intervals of its genomic sequences [Hud+90 | Wu11]. Any given site in the genome is still explained by a single *marginal tree*, but moving along the sequence changes the tree's topology at recombination points.

Unfortunately, contemporary algorithms to construct ancestral recombination graphs are unfit to process more than a few dozen genomes at once [RHGS14]. To process modern data sets with thousands of genomic sequences (like the *1000 Genomes Project* [Con+15]), Song and Hein proposed working with the sequence of marginal trees directly. They show that using trees to explain local ancestries yields more accurate results than previous methods [SH05]. Expanding on this idea, Kelleher et al. [Kel+19] published their tool *tsinfer* that infers a sequence of trees for thousands of sample sequences.

1.2 Contribution

In this work, we examine the *tsinfer* algorithm, more closely focusing on the algorithmic aspects than the authors did. The basis of *tsinfer's* inference algorithm is a modified Viterbi algorithm, which we describe in detail in Section 3.4.2. We reimplement this algorithm in our own tool named *airs*,[1] using cache-aware data structures to iterate faster over the tree

[1] GitHub repository: *https://github.com/Cydhra/libairs/tree/77d5220d848f6325148b48ee175c1528a957c531*

sequence during inference (Section 4.1). Furthermore, we reduce the number of operations on the marginal trees with an additional contracted tree data structure, and use a more efficient approach to its traceback phase (Section 4.3).

We show that the structure of the input data limits the amount of work that *tsinfer* can parallelize, and propose a novel parallelization scheme that improves scaling to high processor counts independently of the input data, almost doubling the speedup factor at 128 cores from 50 to 94 (Figure 5.5b). Our alternative approach utilizes the available cores more effectively, by allowing threads to perform some work redundantly (Section 3.6). We benchmark it against *tsinfer* as part of our own implementation. As a tradeoff, it does not necessarily compute the same tree sequence when run with different numbers of threads on the same input, while *tsinfer* provides this guarantee. Finally, we present ideas on how to improve both the implementation and our parallelization scheme further (Section 6.1).

1.3 Outline

In Chapter 2, we introduce the necessary terminology and notation to understand the biological and mathematical models this work is based on, as well as the data structures this algorithm employs. Additionally, we present the Viterbi algorithm as a fundamental building block of the *tsinfer* inference algorithm. Afterward, we describe *tsinfer*'s inference algorithm in detail in Chapter 3. At the end of Chapter 3, we also present our proposed parallelization scheme, comparing it with *tsinfer*'s approach. Next, we motivate and discuss our implementation choices, and compare them with *tsinfer* (Chapter 4, presenting the results of our runtime measurements in Chapter 5. Finally, in Chapter 6, we provide a discussion of open problems regarding the accuracy of the calculated tree sequence and possible further optimizations of our algorithm and parallelization scheme.

2 Background

In this work, we present an algorithm to infer ancestries between individuals of the same species based on their genomic sequences. In this chapter, we introduce definitions and terminology necessary to understand the algorithm in later chapters. In Section 2.1 we introduce Markov chains, which is the model underlying *tsinfer* and our adaption. Next, we introduce necessary terminology from biology in Section 2.2. Finally, in Section 2.3 we discuss some concepts unique to *tsinfer* and tree sequences.

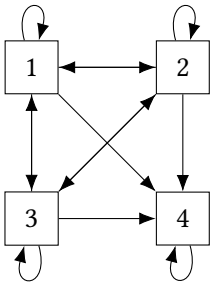
2.1 Markov Chains

A discrete Markov chain is a stochastic model describing random processes using a finite set of states with probabilistic state transitions.

Definition 2.1: *Let $\mathcal{S} \coloneqq \{x_0 \dots x_n\}$ be a set of n states and $A^{n \times n}$ a transition matrix where $A_{i,j}$ defines the probability of transitioning from state x_i to x_j. If $A^{n \times n}$ is invariant over time, we call the tuple $\mathcal{M} \coloneqq (\mathcal{S}, A)$ a* Markov chain.

We call the sequence of random variables $\mathcal{X} \coloneqq \{X_0 = x^1, X_1 = x^2, \dots, X_m = x^m\}$ with $x^i \in \mathcal{S}$ the state sequence. *Formally, the probability of a random variable X_i assuming the value $x_i \in \mathcal{S}$ depends only on the value of X_{i-1} and is defined by A as $\mathbb{P}[X_i = x_i \mid X_{i-1} = x_j] = A_{j,i}$. That is, the probability distribution for the next state depends only on the current state, but not on any previous states. We call this property the* Markov property.

Further, we associate each state transition with an output value o_i called observation. *This means that each sequence of states $\mathcal{X}$ results in a sequence of observations $\mathcal{O} = \{o_i \dots o_m\}$. Note that, observation o_0 depends only on the value of X_0, since no state transition leads to X_0.*



$$A \coloneqq \begin{bmatrix} .25 & .25 & .25 & .25 \\ .25 & .25 & .25 & .25 \\ .25 & .25 & .25 & .25 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 2.1: A Markov chain model with 4 states and uniform probability distribution for state transitions between states 1 to 3. State 4 never transitions into a different state. If run for n steps, the random variables X_0 to X_n will select states uniformly at random until arriving in state 4 for the first time. Afterward, every following variable will always assume state 4.

For example, suppose the Markov chain model displayed in Figure 2.1 outputs the letter of the alphabet of the new state in each step ($1 \to A, 2 \to B, \ldots$). If the Markov chain generates a state sequence $\mathcal{S} = \{X_0 = 2, X_1 = 1, X_2 = 2, X_3 = 3, X_4 = 4, X_5 = 4, X_6 = 4, X_7 = 4, , X_8 = 4, , X_9 = 4\}$, the output sequence after nine steps of this Markov process would be `BABCDDDDD`.

Notably, we define that outputs are generated during state transitions, meaning the output o_i can depend both on the previous state X_{i-1} and the new state X_i. This implies that distinct state sequences can yield identical sequences of observations.

2.1.1 Hidden Markov Models

Definition 2.2: *Let $\mathcal{M} \coloneqq (\mathcal{S}, A)$ be a Markov chain. Further, let $\mathcal{O} \coloneqq \{o_0 \ldots o_m\}$ be a set of m observations from running $\mathcal{M}$. We call the sequence of random variables $\mathcal{X} \coloneqq \{X_0, X_1, \ldots, X_m\}$ (with unknown values) leading to $\mathcal{O}$ a* hidden Markov model.

Given a sequence of observations $\mathcal{O}$ and the Markov chain $\mathcal{M} \coloneqq (\mathcal{S}, A)$, the *Viterbi algorithm* [Vit67] is a dynamic programming algorithm that calculates the most likely state sequence leading to $\mathcal{O}$.

The Viterbi algorithm consists of two phases, (1) the forwards phase, and (2) the traceback phase. In the *forwards* phase, the algorithm starts with a prior probability distribution determining the initial likelihood of each state of the Markov chain. The algorithm then updates the likelihoods assigned to each state iteratively for each observation. In iteration i, the new likelihood for state x_K with $0 \leq K < n$ is the maximum of all transitions into x_K that yield observation o_i:

$$L[X_i = x_K] \coloneqq \max_{0 \leq j < n} L[X_{i-1} = x_j] \cdot \mathbb{P}[X_i = x_K \mid X_{i-1} = x_j, o_i]$$

where $\mathbb{P}[X_i = x_K \mid X_{i-1} = x_j, o_i]$ is obtained from the marginal distribution of transitioning from x_j given observation o_i:

$$\mathbb{P}[X_i = x_K \mid X_{i-1} = x_j, o_i] = \frac{\mathbb{P}[X_i = x_K | X_{i-1} = x_j] \cdot \mathbb{P}[o_i \mid X_{i-1} = x_j, X_i = x_K]}{\sum_{0 \leq a < n} \mathbb{P}[X_i = x_a | X_{i-1} = x_j] \cdot \mathbb{P}[o_i \mid X_{i-1} = x_j, X_i = x_a]}$$

We care only for the state transition with the highest probability, because we are searching for the most likely path through the Markov process that yields $\mathcal{O}$. For each state x_K in each step i, we store the most likely state transition that leads into it in a table (Figure 2.2).

Because every state has exactly one incoming state transition, they form unambiguous paths from every state backwards through the Markov chain. Some states have no outgoing connections into future steps, if no state in the next step is most likely reached from it, but it is always possible to trace back to the first step of the Markov chain. We call paths in this table *Viterbi paths*.

After the algorithm iterated through all steps of the Markov chain, it enters the *traceback* phase. It selects the state of the Markov chain with the highest likelihood. Then, it traces back through the table to reconstruct a contiguous path through the Markov chain. Due to the Markov property (see Definition 2.1) and the fact that we only store the most likely transition into each state, this path is the most likely state sequence that leads to $\mathcal{O}$.

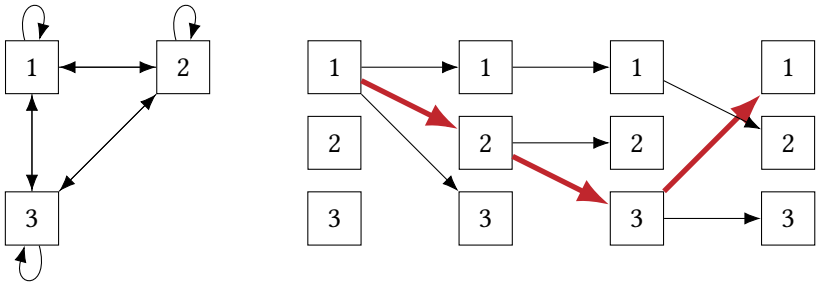


Figure 2.2: An examplary table (right) for a three-state hidden Markov chain (left) run for four steps (the transition probabilities and observations are not shown for simplicity, but they are required for the algorithm). The table has a row for each state of the Markov chain and a column for each step. Each entry of the table stores the most likely state transition that leads into that state in the respective time step. When traced back from right to left, the transitions form contiguous Viterbi paths through the Markov chain. Selecting state 1 in the last step as the most likely state leads to the red path as the most likely Viterbi path.

2.2 Genetics

In this section, we introduce terminology and necessary simplified concepts from genetics.

Species and Individuals A species is a classification for a group of organisms. While exact definitions of the term vary, we define a species as a group of organisms that can produce fertile offspring.[1] We call organisms within a species' *individuals*.

DNA Deoxyribonucleic acid (DNA) is a molecule encoding the genetic information of each known individual.[2] DNA is a long chain of nucleotides, which are molecules made from one of four different bases denoted by a letter (`G`, `C`, `A`, `T`). The sequence of all four possible bases encountered along a DNA molecule encodes the genetic information of an individual. The chemical properties of DNA, nucleotides, and the four bases are irrelevant for this work; we focus only on the genetic information encoded in the base sequence.

Most organisms have more than one DNA molecule per cell. Each DNA molecule is called a *chromosome*.[3] Within each chromosome, different regions (called *genes*) code for different biomolecules, while other regions have regulatory or chemical functions. Because the function of any genetic information is irrelevant for this work, though, we consider chromosomes and nucleotides, ignoring any further distinction. Species can differ widely, both in the amount of chromosomes and the sequence of bases within them, but within a species, only a small fraction of DNA differs between individuals.

Variant Sites Mutations in the DNA sequence lead to differences between species and individuals within a species. While there are different kinds of mutations, we consider only *Single-Nucleotide Polymorphisms (SNPs)*. SNPs are changes observed in a single base in one position of a DNA sequence, either by nucleotide conversion, or insertion of a base and

[1] Since this work is explicitly about sexually reproducing organisms (see paragraph about ancestry), we exclude asexual reproduction

[2] Though other genetic information exists, we only consider information encoded in the DNA molecule.

[3] There are DNA molecules that are not considered a chromosome, but this distinction is irrelevant for this work, so we simplify here.

subsequent deletion of an adjacent base. We call sequence positions that carry an SNP *variant sites*. We do not consider mutations that change the sequence length of DNA, delete or insert nucleotides, or change the amount or configuration of chromosomes (for example, by recombining ends of previously distinct chromosomes), so each mutation in the input data has a well-defined sequence position (the variant site).

Alleles Every chromosome that has a mutation compared to a reference chromosome introduces an *allele* for the affected variant site. We define an allele as a base that differs among distinct copies of a chromosome (*haplotypes*). If we consider only two chromosomes, at most two alleles can exist: the reference allele (*ancestral allele*) and the mutated allele (*derived allele*). We call mutations that only have two associated alleles across all considered DNA sequences *biallelic*. If more copies of the same chromosome carry SNPs with different bases in the same sequence position, the mutation becomes *multiallelic*.

Ploidity Different species have different numbers of copies of their chromosomes in their cells. Each copy is called a *haplotype* and has an individual ancestry. We call organisms that have only one copy per chromosome *haploid*, organisms with two chromosomes *diploid*, and organisms with two or more copies *polyploid*. For this work, a *polyploid* genome is essentially equivalent to multiple *haploid* organisms, since both sets of chromosomes have individual ancestries (see the next definition). For this reason, we use the terms *genome*, *genomic sequence*, and *haplotype* synonymously and ignore if two haplotypes originated from the same individual.

Ancestry, Copying, and Recombination When individuals of a species replicate sexually, they pass genetic information to their descendants. In this work, we focus on sexually reproducing species. For those, two specialized parent cells containing half of a complete genome merge, so the descendant cell inherits half of its genome from the mother organism, and half from the father organism. This requires at least a short diploid phase in an organism's lifecycle, meaning sexually reproducing organisms have at least two copies of their genome at one point after the parent cells merge. Both copies have individual ancestries, but then undergo *recombination*.

During recombination, matching chromosomes pair up, break at random sites,[4] and recombine with each other. This mixes up the ancestries, meaning both matching chromosomes now copy parts from both parent organisms.

2.3 Tree Sequences

The tree sequence that we build during this algorithm aims to explain the ancestries observed in recombining organisms. For this, we divide a set of DNA sequences (*sample sequences*) into consecutive intervals that are copied from different ancestors. Then, we construct trees using inferred parent-descendant relationships induced by these intervals. In this section, we define what a tree sequence is and how it models the ancestries between sample genomes.

[4] The breaking sites are not chosen uniformly at random, but we will assume this property as a simplification for this work (Section 3.1).

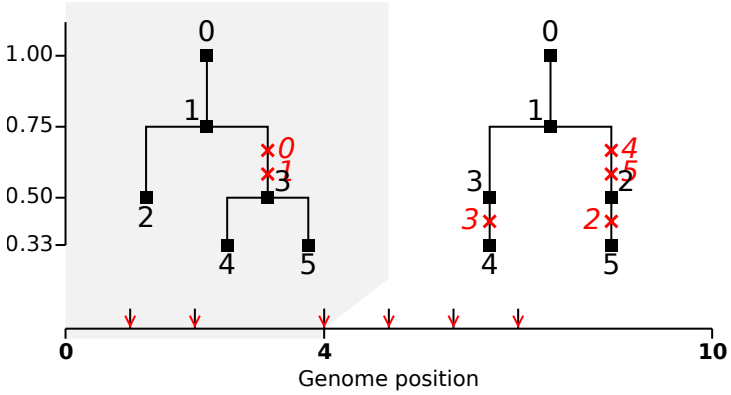


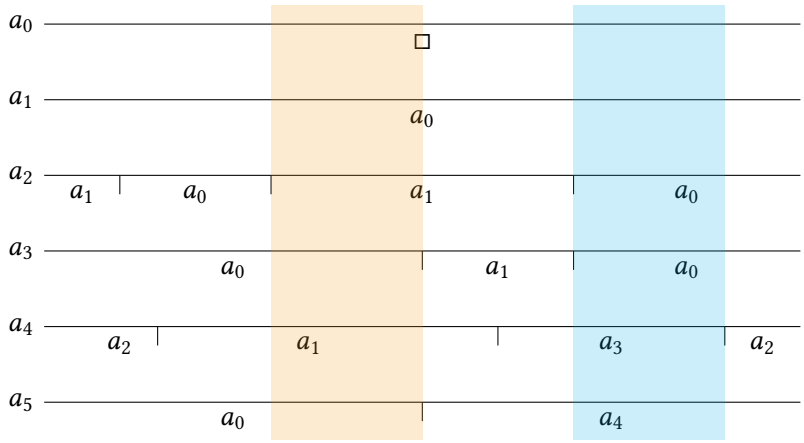
Figure 2.3: An exemplary tree sequence from synthetic data, rendered with the tool suite of *tsinfer*. The x-axis shows genomic positions, the y-axis shows the relative age of ancestors (Section 3.3.2), red crosses mark mutations. Two marginal trees cover the intervals $0 \ldots 4$ and $4 \ldots 10$. At genomic position 4, a recombination in node 5 changes the tree topology.

Marginal Tree The marginal tree describes the ancestry at a given site in the genome for all haplotypes that are present in the marginal tree. Because we assume no genealogical hierarchy among the input sequences, they are the leaves of the marginal tree. Inner nodes are hypothetical ancestor haplotypes, which the algorithm infers from available data (see Section 3.3). The algorithm then divides the input sample sequences and ancestor sequences into intervals and infers a parent-descendant relationship between each interval and an older ancestor sequence (Figure 2.4a). This relationship models copying the genomic interval from the older ancestor. Within each genomic interval of a sequence S, the marginal tree must contain an edge that models the inferred relationship between S and the parent ancestor. This means that each marginal tree at a site consists of all edges associated with the intervals that contain that site (Figure 2.4b and Figure 2.4c).

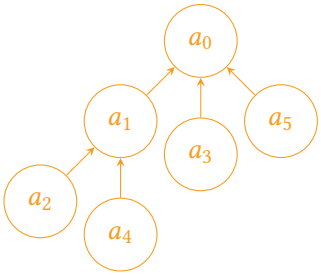
Moving along the genomic sequence, the marginal tree changes topology whenever an interval ends in any sequence. The edge with the ended interval *leaves* the marginal tree, and a new edge reconnects the now unconnected node with a new parent node.

Tree Sequence We call a sequence of trees ordered along the length of a genomic sequence a *tree sequence*. Consecutive trees in the tree sequence are defined for consecutive intervals in the genomic sequence. Neighboring variant sites may have the same marginal tree associated with them, if they are not divided by a recombination event in any ancestor genome. Thus, adjacent recombination events of any two ancestors are the boundaries of an interval in the tree sequence that has one marginal tree assigned (Figure 2.4a). Consequently, no ancestor has a recombination event within such an interval.

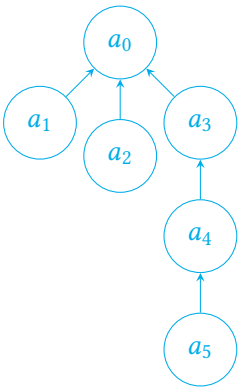
Each site in the genomic sequence is associated with exactly one tree, thus intervals do not overlap and there are no gaps. The tree defined at each site explains the ancestry at that site (Figure 2.4b and Figure 2.4c).



(a) Six ancestor genomes (a_0 through a_5) form a tree sequence with their inferred genomic intervals. Each interval's parent ancestor is denoted between the interval boundaries. The parents induce the edges in the marginal trees. Note that a_0 has only one interval because it has no older ancestors, and a_1 has only one interval because it can copy only from a_0. Two exemplary genomic intervals are marked in orange and cyan, inducing the two different marginal trees shown below.



(b) Marginal tree in the orange interval.



(c) Marginal tree in the cyan interval.

Figure 2.4: We query a tree sequence at two intervals (marked areas in Figure 2.4a). The marginal trees in those intervals are shown in Figure 2.4b and Figure 2.4c.

3 tsinfer

In this chapter, we outline the *tsinfer* algorithm. First, we discuss what input data *tsinfer* requires, and what additional data it can process. In Section 3.2, we provide a broad overview of its three phases, which we further discuss in Sections 3.3 to 3.5. Finally, we examine the parallelization approach of *tsinfer* in Section 3.6, and propose our modified parallelization scheme afterward. Using this overview, we motivate our implementation choices in the next chapter.

3.1 Input Data

tsinfer infers tree sequences from genomic variant data. More specifically, *tsinfer* handles only biallelic single nucleotide polymorphisms (see Section 2.2). We call the individual variants *variant sites* to refer to both the mutation and the site. Each variant site comprises a position in the genome, its ancestral, and derived allele (called *ancestral state* or *derived state*). Additionally, for each sample genome, we store whether it carries the ancestral or derived state. As *tsinfer*, we make the biologically reasonable assumption that each sample genome covers all sites. Neither *tsinfer* nor our implementation handle multiallelic sites. This means every variant site always has exactly two alleles, one of which is the variant's mutation, and every mutation happens exactly once (there cannot be multiple variant sites with the same position).

The *tsinfer* inference algorithm also accepts mutation and recombination rates for each variant site. If only one rate per parameter is given, *tsinfer* assumes constant mutation and recombination rates across the genome. Since this work focuses on the algorithmic aspects of the inference algorithm, we assume the mutation and recombination rate to be constant, and use *tsinfer*'s default values for them. Changing this behavior is a trivial modification of our implementation.

The *tsinfer* tool does accept information about which sequences originate from the same (polyploid) individual, but the inference algorithm exclusively works on haploid sequences. It stores information about diploid or polyploid genomes in the resulting tree sequence, but it does not influence inference. For this reason, we consider processing this data out of scope.

Moreover, *tsinfer* also accepts metadata about the samples, the sampled individuals, population data, and provenance data. Since the inference algorithm uses none of this data, we consider any metadata out of scope for this work.

Finally, *tsinfer* specially handles low-confidence data and missing data. We consider this out of scope as well, but the functionality can be trivially added to our algorithm.

3.2 Overview

The inference algorithm is split into three phases: (1) Ancestor Generation, (2) Ancestor Matching, and (3) Sample Matching. We provide a brief overview of each phase, and then explain them in depth in Sections 3.3 to 3.5.

During *Ancestor Generation*, the algorithm infers putative ancestral sequences from the input variant sites. Aside from some special cases explained later, it will infer one hypothetical sequence for each variant site, which we call the *ancestral sequence*. We call the mutation site that induces an ancestral sequence its *focal site*. Starting from the focal site, the generation algorithm extends the sequence in both directions by guessing the most likely state at each site from the available variant data using various heuristics (Section 3.3.4). Unlike sample sequences, an ancestor is not necessarily defined for the entire sequence length. The observed frequency of a focal mutation among the sample sequences determines the relative age of the induced ancestral sequence.

In the *Ancestor Matching* phase, *tsinfer* sorts the ancestral sequences by their relative age and then inserts them into a common tree sequence. To insert an ancestor, the algorithm has to find intervals within its haplotype that are copied from older ancestors. We call the ancestor that is copied from the *parent* of that interval. For each interval, the algorithm inserts one edge to its parent into the tree sequence. The matching phase sorts the ancestors, because they are allowed to copy only from older ancestors, and so it exploits their relative age to simplify the insertion process.

The algorithm inserts ancestors iteratively, beginning with the ancestral state A_0. The full tree sequence of ancestor A_0 to A_n is the input for iteration $n + 1$, which inserts ancestor A_{n+1} into the sequence. To match an ancestor, Kelleher et al. [Kel+19] use a modified Viterbi algorithm to find the most likely parent ancestor sequence from which A_{n+1} copies at each site.

The *Sample Matching* phase inserts the input samples into the tree sequence. This phase functionally performs the same operations as the second phase, but the sample sequences cannot copy from each other. This simplification allows for trivial parallelization and simplifies the Viterbi algorithm because the hidden Markov model does not change between instances.

3.3 Ancestor Generation

We want to generate a tree sequence in such a way that at each variant site, we have a *marginal tree* that explains the common ancestry between all input samples at that site. Since the input data only comprises the leaves of this tree, *tsinfer* has to infer common ancestors between the samples. It infers ancestral sequences based on the assumption that every mutation occurred only once in a species' history, and every individual that carries this mutation inherited it from this common ancestor. Thus, in Phase 1 of the algorithm, it generates about one ancestor sequence per mutation, which represents one node in the tree sequence. We discuss exceptions in Section 3.3.3. In Section 3.3.1 and Section 3.3.4, we explain how the algorithm generates the ancestor sequences.

3.3.1 Focal Sites

Naively, each mutation induces one ancestral sequence, and we call this mutation the *focal site* or *focal mutation* for this ancestor (see Figure 3.1). Further, Kelleher et al. [Kel+19] assume that because DNA is (imperfectly) copied in blocks from ancestors, the surrounding sequence around the focal mutation can be inferred by comparing the sequences of all samples that carry this mutation. Based on these two assumptions, *tsinfer* generates one ancestor per mutation and extends it in both directions by selecting either the ancestral state or the derived state based on majority consensus between all ancestors that carry the mutation (see Section 3.3.4).

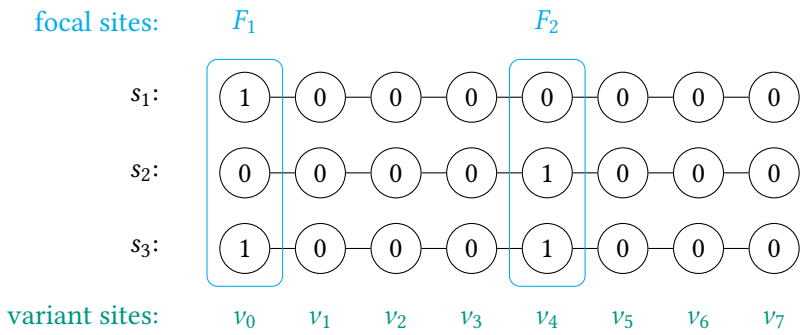


Figure 3.1: Three sample sequences s_1 to s_3 carry two mutations (ancestral state is denoted as 0, derived state as 1). We choose both mutations as focal sites for one ancestor each, so the algorithm infers two ancestors from this data.

3.3.2 Focal Site Age

We infer the relative age of a mutation by its observed frequency among the input sample sequences. It induces only an order among mutations, it does not provide meaningful information about the real chronology of a species' history. For example, the focal sites F_1 and F_2 from Figure 3.1 both have relative age $\frac{2}{3}$. Kelleher et al. [Kel+19] base this heuristic on two publications [KO73 | GT98] and claim that it yields "surprisingly accurate" results. We discuss this heuristic and its implications further in Section 3.6 and Section 6.5. The relative age is relevant both for the later Ancestor Matching step (as pointed out in Section 3.2), and for the Ancestor Generation. That is, when extending an ancestor of relative age x, we do not want to include mutations in its putative sequence that are younger than the ancestor itself.

3.3.3 Merging and Breaking Focal Sites

If a set of mutations are always observed together, *tsinfer* merges them and they induce only one ancestor. This ancestor, consequently, has multiple focal sites. For example, the derived state at variant sites v_0, v_1, and v_6 of Figure 3.2 is present in all of s_1, s_2, and s_4. No other sample sequence carries either of the three mutations. Those three variant sites can thus be merged into a set of three focal sites and induce a common ancestor. We call the samples that carry the derived state for the merged focal sites the *common sample set*.

Merging focal sites reduces the number of generated ancestors, which reduces the number of Viterbi algorithm instances in Phase 2. Moreover, it also reduces the tree size, since the merged mutations induce only one ancestor in the output tree sequence. Observe that if these focal sites were not merged, they would induce ancestors forming a simple path in the marginal trees at these focal sites. That is the case because all samples in the common sample set have the same induced ancestors as predecessors in the tree, since these samples inherit the same mutations from them. Analogously, none of the other samples carry the sample focal mutations, and thus none of them inherit from the same ancestors in the tree sequence. Merging the focal sites thus effectively compresses a simple path of ancestors in the marginal tree into a single ancestral node.

If there is no consensus about the state of older mutations in the sequence between merged focal sites, there cannot exist a common ancestor that explains all merged mutations. Therefore, if the common sample set disagrees on the state of a variant site that is older than and lies

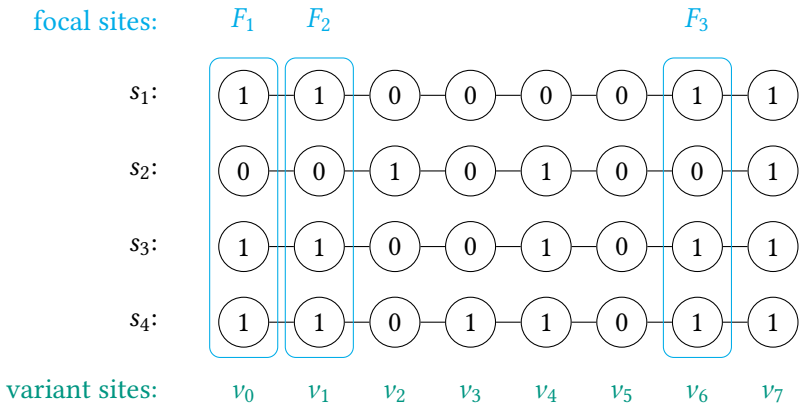


Figure 3.2: Variant sites v_0, v_1, and v_6 show mutations that are exclusively present in samples s_1, s_2, and s_4. We can therefore merge variant sites v_0 and v_1 into a single ancestor because there is no sequence in the algorithm copying only either one of them. Conversely, we cannot merge v_6 into the same ancestor because sample s_4 differs in the older state v_3 and we cannot be sure whether all sequences will copy from a merged ancestor containing all three variant sites.

between two focal sites, the focal sites cannot be merged.[1] For example, in Figure 3.2 variant site v_3 has relative age $\frac{1}{4}$ while F_1, F_2, and F_3 have relative age $\frac{3}{4}$. But since s_1, s_2, and s_4 disagree at site v_3, they cannot share a common ancestor that ranges from F_2 to F_3.

3.3.4 Extending from Focal Sites

Once *tsinfer* selected the focal sites, we extend each into an ancestral sequence. Each ancestral sequence induces one inner node in the tree sequence, which we call an *ancestor*. Since all ancestors are independent of each other, extending from focal sites is trivially parallelizable.

The extension algorithm examines the set of samples that carry the focal mutation (the *consensus set*) site by site. For each site, we decide the ancestral haplotype state by majority vote among the sample sequences in the consensus set (Figure 3.3). The vote does not consider sites belonging to a mutation younger than the focal mutation, since younger mutations cannot be present in older ancestors (see Site 2 in Figure 3.3). Instead, these sites default to the ancestral state.

As we are assuming that the degree of shared evolutionary history is directly dependent on the physical distance of two sites in the genome [Rei+02], the further away a site is from the current focal site, the weaker the signal it adds to the putative ancestor induced by the sequences. Consequently, once the signal gets too weak,[2] the extension algorithm often terminates before reaching the end of a sequence. To achieve this, the algorithm removes samples from the consensus set when they disagree with the consensus twice in a row (see samples s_5 and s_6 in Figure 3.3). Once the consensus set has reached half its original size, the extension algorithm terminates for the respective direction (see site 7 in Figure 3.3). Kelleher et al. [Kel+19] chose both of these heuristics empirically.

[1] While we can explain a disagreement with additional recombination events in descendant ancestors, the merging operation is a heuristic to simplify the tree, which is not useful when some of the descendants cannot copy from the common ancestor.

[2] The notion of weak signals is a simple heuristic here.

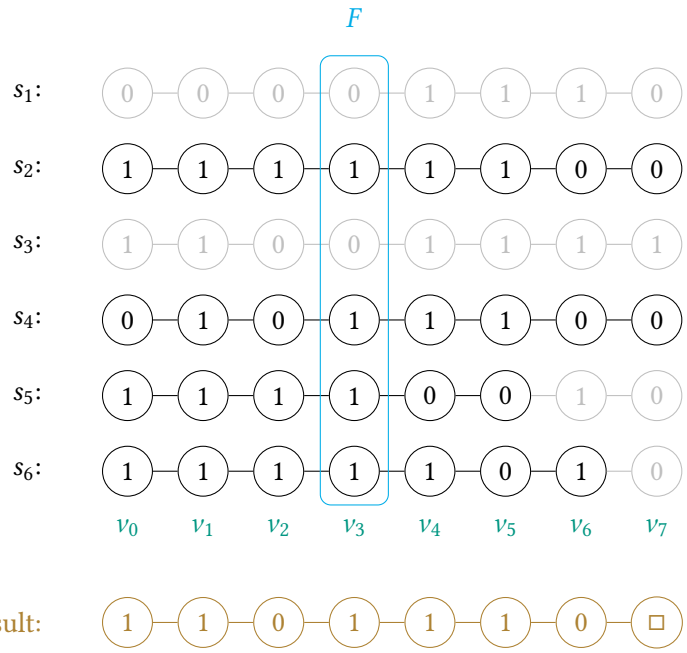


Figure 3.3: We extend the focal site F into an ancestor sequence by choosing its state using a majority vote of the consensus sample set (black). If a sample disagrees with the consensus at two consecutive positions, we remove it from the consensus set (gray). If the consensus sample set shrinks to half its original size, we terminate extension in that direction (site v_7), and subsequent result states remain undefined.

The extension algorithm extends singular focal sites in both directions along the genome until its heuristics cut it off. The ancestor is defined only between the two cut-off points; the sequence is undefined for all other sites. Thus, unlike sample sequences, ancestor sequences do not necessarily span the entire input sequence. If the ancestor has multiple focal sites, the algorithm extends the left-most site to the left, the right-most site to the right, and the space in between focal sites without its cut-off heuristics, so the ancestor does not have undefined holes.

The ancestor generation is trivially parallelizable and makes up only a negligible fraction of the overall runtime. We thus prioritize other sections of the algorithm and re-implement the ancestor generation without optimizations. Nonetheless, it does lend itself for future consideration, since it is based on many ad-hoc heuristics, rather than well-researched mathematical models. We discuss some caveats concerning these heuristics in Section 3.6 and Section 6.5.

3.4 Ancestor Matching

After generating the ancestral sequences, we continue with the construction of a tree sequence from them, thereby modeling their evolutionary history and allowing us to later place the sample sequences into the tree sequence. Kelleher et al. [Kel+19] call inserting the ancestor sequences into a tree sequence *ancestor matching*. For this, we need to infer genomic intervals in each ancestor sequence that are copied from older ancestors. For each genomic interval, we insert one edge into the tree sequence describing the parent-descendant relationship within this genomic interval. The transition of one such interval to the next along the genome

represents a recombination event. Once all ancestors are inserted into the tree sequence, the tree sequence defines a *marginal tree* at each site, comprising the ancestral sequences which are defined for this genomic site and the inferred edges between them. This *marginal tree* describes the inferred evolutionary history of all ancestors at this site.

In Section 3.4.1 we explain the Markov model which Kelleher et al. [Kel+19] base the inference of the tree sequence on. In Section 3.4.2 we explain how the modified Viterbi algorithm works that finds the most likely parent ancestors for each consecutive interval of a haplotype's genome. Finally, we explain the most crucial optimizations that *tsinfer* performs in Section 3.4.3.

3.4.1 Li-Stephens Model

Kelleher et al. [Kel+19] base the inference of the tree sequence on a Markov chain model developed by Li and Stephens [LS03]. This model simulates a new haplotype by copying blocks from k known haplotypes, using a Markov chain to select the parent haplotypes. The Markov chain is based on the same biallelic single-mutation model which we explained in Section 3.1. In short, it operates on observed variant sites, and each variant site has an ancestral and a derived allele. The model never generates more than two alleles per variant site, and mutations comprise only one nucleotide.

Li and Stephens [LS03] define a Markov chain (see Section 2.1) with $0 \leq i < k$ states x_i for each variant site (Figure 3.4). The Markov chain models the process of generating a new haplotype h_k from a set of existing haplotypes $h_0 \dots h_{k-1}$. Transitioning into state x_i at site j (Figure 3.4) models copying from ancestor h_i at site j. The transition matrix defines probabilities to transition from state x_i at site j to $x_{i'}$ at site $j+1$ according to the following equation:

$$\mathbb{P}\left[X_{j+1} = x_i \mid X_j = x_{i'}\right] := \begin{cases} \exp(-\rho_j \cdot d_j) + \left(1 - \exp(-\rho_j \cdot d_j)\right)\left(\frac{1}{k}\right) & \text{if } i = i' \\ \left(1 - \exp(-\rho_j \cdot d_j)\right)\left(\frac{1}{k}\right) & \text{else} \end{cases}$$

Here, ρ_j is the recombination rate per unit distance at site j, which is assumed to be non-constant due to various biological effects, and d_j is the distance between site j and site $j+1$.

The exponential term $\exp(-\rho_j \cdot d_j)$ models the expectation that sites that are physically close to each other (i.e., d_j is small), have a high probability of copying from the same ancestor. $\left(1 - \exp(-\rho_j \cdot d_j)\right)\left(\frac{1}{k}\right)$ distributes the inverse of this probability equally among all k ancestors, which is why the term also appears in the first case (modeling a recombination to itself).

A second transition function models random mutations by introducing another case distinction. It switches between the ancestral allele of the selected parent sequence and the derived allele, given

$$\mathbb{P}\left[h_{k_j} = a \mid X_j = x_i, h_0, \dots, h_{k-1}\right] := \begin{cases} \frac{k}{k+\theta} + \frac{1}{2} \cdot \frac{\theta}{k+\theta} & \text{if } a = h_{i_j} \\ \frac{1}{2} \cdot \frac{\theta}{k+\theta} & \text{if } a \neq h_{i_j} \end{cases}$$

Here, θ is the mutation rate. The copying process is exact (i.e., $h_{k_j} = h_{i_j}$) with probability $\frac{k}{k+\theta}$, and a mutation occurs with probability $\frac{\theta}{k+\theta}$. Due to the $\frac{1}{2} \cdot \frac{\theta}{k+\theta}$ term in the model, the probability to mutate a site approaches $\frac{1}{2}$ for $\theta \to \infty$.

While Li and Stephens [LS03] combine both transition functions into a single Markov chain model, the functions are unrelated, and we can therefore consider them separately (see Algorithm 3.1).

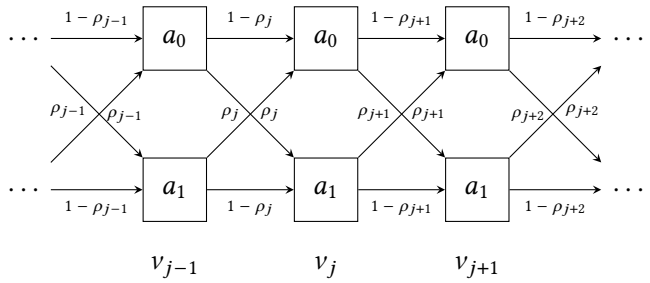


Figure 3.4: The Li-Stephens model Markov chain has one state per ancestor (a_0 and a_1) per variant site. This allows for varying recombination probabilities (abbreviated as ρ_j here) between sites.

This Markov chain models a process that generates a new haplotype h_k by imperfectly copying blocks from a set of ancestors $h_0 \dots h_{k-1}$. To use it for inference, Kelleher et al. use the Markov chain in a hidden Markov model (see Section 2.1). Now, a given haplotype is an observation, which we can use to find the most likely state sequence that produced the haplotype using the Viterbi algorithm (Section 2.1.1). Kelleher et al. [Kel+19] call this state sequence the *copying path*, and the states in the sequence define the parent haplotype for each variant site, with sites physically close to each other likely having the same parent. We combine adjacent sites with the same parent into genomic intervals (see Figure 2.4a). For each interval, we insert an edge between the corresponding parent node and the new haplotype node in the tree sequence. To find the copying path, Kelleher et al. [Kel+19] use a modification of the canonical Viterbi algorithm, which we explain next.

3.4.2 Modified Viterbi

Here, we describe an algorithm to find the *copying path* (Section 3.4.1) of a single ancestor sequence. Each ancestor sequence (and in Phase 3 each sample sequence) that we insert into the tree sequence, requires one instance of this algorithm. It finds the parent sequences from which the inserted sequence copies its genome.

The oldest ancestor has no (known) ancestor to copy its sequence from, and thus is the only ancestor not copying its states from any older ancestors. Therefore, the oldest ancestor serves as the root node of the tree sequence. Now, consider iteration $n + 1$: To insert the new *input haplotype* into the tree sequence, *tsinfer* treats it as the observation of the hidden Markov model discussed in Section 3.4.1. The haplotype we are inserting can copy from all older ancestors, which are already in the tree sequence. In order to find the copying path of the new haplotype, Kelleher et al. [Kel+19] use the Viterbi algorithm discussed in Section 2.1.1.

For the *forward* phase, the algorithm starts with a uniform prior distribution of all states the Markov chain can be in – one for each older ancestor. Henceforth, we refer to these states as *Markov states* to distinguish them from the allele state of a sequence. The algorithm then iterates over all variant sites of the input haplotype, which is our sequence of observations. For each site, the algorithm calculates the most likely state transition from each Markov state of the previous site into each k Markov state of the current site.

Canonically, the Viterbi algorithm reconstructs the most likely path through the Markov states. It therefore stores the most likely state transitions in a table (variant site $\times$ state). Note, however, that the probabilities for recombination events are symmetrical. That is, the probability of transitioning from state x to state $y \neq x$ via a recombination is $L[x] \cdot \rho_j \cdot d_j$,

Algorithm 3.1: The Viterbi algorithm forward phase. Recombination and mutation probability are abbreviated as ρ and θ.

Input: A set of ancestor haplotypes `ancestors`. The input haplotype `input_haplotype`.

forall s in 0..`len(input_haplotype)` **do**
  forall k in 0..`len(ancestors)` **do**
    `p_recomb` $\longleftarrow L_{M_{s-1},s-1} \cdot \rho$
    `p_no_recomb` $\longleftarrow L_{k,s-1} \cdot (1-\rho)$
    if `p_recomb` > `p_no_recomb` // if recombination is more likely
    then
      `recomb_flags[s]` $\longleftarrow$ `true`
      $r \longleftarrow$ `p_recomb`
    else
      $r \longleftarrow$ `p_no_recomb`
    if `input_haplotype[s]` $\neq$ `ancestors[k][s]` // if mutation required
    then
      `mutation_flags[s]` $\longleftarrow$ `true`
      $m \longleftarrow \theta$
    else
      $m \longleftarrow (1-\theta)$
    $L_{k,s} \longleftarrow r \cdot m$ // calculate new likelihood
  $M_s \longleftarrow \max_k$ `ancestors`$[L_{k,s}]$ // calculate site maximum

which depends only on $L[x]$ and the recombination rate $\rho_j \cdot d_j$ (which is constant for this variant site; see Section 3.4.1). It therefore suffices to store (1) a flag indicating whether a recombination occurring or not occurring is more likely, and (2) the most likely Markov state x of the previous site.

If an ancestor has a different ancestral state than the input haplotype, it additionally sets a flag indicating that a mutation happened and adjust the state's likelihood to account for this mutation (Algorithm 3.1). We use this mutation flag during *traceback* to account for mutations explaining incorrect copying.

After the forward phase completes, we want to reconstruct the most likely path through the Markov chain (i.e., the concatenation of the most likely transition at each variant site). Thus, the Viterbi algorithm picks the Markov state with the highest likelihood at the last variant site. We call the Markov state's associated ancestor the *current ancestor*. The algorithm then iterates backwards through all variant sites and records the intervals in which the current ancestor is the most likely state. Once it encounters a recombination flag for the current ancestor, the traceback algorithm switches to the most likely state of the previous site. Otherwise, if no flag is set, the most likely state transition is remaining in the current Markov state, and the algorithm moves to the previous site. For each mutation flag encountered for the current ancestor, we record a mutation in the tree sequence.

In subsequent iterations, the input haplotype is part of the set of ancestors that can be copied from, and so the Markov chain has one additional state per site (Section 3.4.1). A notable exception are ancestors with the same relative age, since they are not allowed to copy from each other. This is especially important when parallelizing the ancestor matching phase, as discussed in Section 3.6.

3.4.3 Likelihood Contraction

Only including older ancestors in the Markov chain and the Viterbi algorithm already reduces the computational effort substantially. Nevertheless, the asymptotic computational complexity is quadratic in the amount of mutations because the number of generated ancestors scales linearly with the number of mutations. As contemporary datasets like the *1000 Genomes Project* [Con+15] contain millions of variant sites per chromosome, a naive Viterbi implementation becomes computationally infeasible.

In this section, we discuss a major optimization of *tsinfer* and, by extension, our implementation. It directly affects the parallelizability of the ancestor matching phase, which we discuss in Section 3.6 after explaining this optimization.

Kelleher et al. [Kel+19] exploit the tree sequence data structure to further reduce the amount of work per Viterbi iteration. Note that many ancestors have the same likelihood during the Viterbi algorithm: Suppose two ancestors A, B have the same likelihood at variant site i. The likelihoods of A and B at site $i + 1$ can differ only if their ancestral states differ at site $i + 1$. If the ancestral states are equal, the algorithm calculates both new likelihoods with the same procedure. Otherwise, one gets multiplied by the mutation probability, and one by the inverse mutation probability (depending on which state matches the input haplotype; see Algorithm 3.1). The likelihood can remain different for subsequent events, but initially, only a difference in ancestral states can cause their likelihoods to diverge. Since sequences often vary only in a few sites, their likelihoods, especially of sequences that copy from each other, remain equal for large parts of the Viterbi iteration.

Moreover, we intuitively expect distant mutations to have less effect on the current likelihood than closer mutations. In fact, for ancestors with different likelihoods that have the same suffix of allele states (ancestral or derived) in subsequent sites, we often observe that the likelihoods converge.

The partially constructed tree sequence already contains information on when two ancestors differ, namely mutation events. The matching algorithm can use this knowledge to avoid performing redundant likelihood calculations. When the inference algorithm starts a new Viterbi instance, all ancestors have the same initial likelihood. Because all ancestors have a common root in the tree sequence (the ancestral sequence), the matching algorithm needs to calculate the next likelihood only for this common root. We maintain a set of *active ancestors*, which initially contains only the root. The algorithm performs likelihood calculations only for ancestors in this set. Whenever an ancestor has a mutation compared to its parent in the tree, the algorithm adds it to the set of active ancestors. After each step, the algorithm compares each active ancestor's likelihood with the likelihood of its next active parent. If they match, it removes the ancestor from the active set, and only the parent remains in the set.

Removing nodes from the active ancestors set parallels contraction of the edge between parent and descendant nodes in the marginal tree. Initially, the entire marginal tree at the first variant site is contracted into the root. All recombination events that affect an uncontracted node also affect all nodes that are merged into this node because their likelihoods are equal. Whenever a mutation event causes a difference in the parent's and the descendant's genomic

sequence, the likelihood of the descendant node diverges from its parent node, and so we reinsert the node and its edge into the marginal tree. Once the likelihoods of a descendant node and its parent converge, we contract their edge again. Thus, we call this optimization *likelihood contraction*.

This has one important caveat. If a recombination moves a node in the marginal tree (Figure 4.1b), all contracted ancestor nodes in the subtree under the moved node change their uncontracted parent node. This means that their common likelihood now potentially differs from their new parent's likelihood, and so the root of the moved subtree (i.e., the moved node) needs to be added to the set of active ancestors until its likelihood matches with its new uncontracted parent.

Preliminary measurements on small datasets show that, often, less than one percent of all ancestors are uncontracted at a time. We will discuss this observation further in Section 4.5. While the algorithm still scales quadratically in the amount of mutations, in practice only a fraction of the input must be considered at a time. For this reason, it is infeasible to run the matching algorithm without this optimization.

This optimization, however, severely limits how parallelizable the algorithm is, since the partially constructed tree sequence of all previous Viterbi instances is now an input for each new iteration. We discuss how *tsinfer* approaches parallelization with this optimization in mind in Section 3.6

There is another caveat to this optimization: Since so many ancestors have the same likelihood at a time, the Li-Stephens model leaves open which of all equally likely ancestors should be the origin of a recombination. Because only the common root of all ancestors with equal likelihood is uncontracted in the algorithm, *tsinfer* will always choose it for recombination events. That is, whenever multiple ancestors are equally likely, it chooses the oldest one. This does not affect the likelihood in the Li-Stephens model, but it can be argued that this is undesirable in a genealogy where ancestors are likely many generations apart. We consider this out of scope for this work, however, and mimic the behavior of *tsinfer* with some restrictions further discussed in Section 3.6.

3.5 Sample Matching

The third step to inference is adding the sample sequences into the tree sequence, called *matching samples*. This step uses the same Viterbi algorithm described in the previous section, albeit without some of the complications. Sample sequences are not allowed to copy from each other, so the set of Markov states is immutable. Therefore, sample sequences constitute the leaves of the marginal trees. This allows for trivial parallelization of the computation, since all instances of the Viterbi algorithm are independent of each other.

3.6 Parallelization

The first and third phases of *tsinfer* are trivially parallelizable because the individual work units do not depend on previous outputs. The ancestor matching step can be parallelized easily as well, because naively, running the modified Viterbi algorithm discussed in Section 3.4 does not require the results from previous ancestors. However, as outlined in Section 3.4.3, the amount of redundant calculations if we disregard the results of previous iterations is too large. In tests performed on small data sets (10 000 variant sites at 5000 samples), we were

able to reach the performance of *tsinfer* running in a single thread with likelihood contraction with our trivial implementation running in four threads without likelihood contraction. But since the amount of redundant work scales quadratically with the amount of variant sites, parallelizing without likelihood contraction does not scale well.

Matching two or more haplotypes in parallel, however, is not possible if the younger haplotype requires the tree edges of the older one for the likelihood contraction. Kelleher et al. [Kel+19] employ two techniques to parallelize as much work as possible, despite this limitation.

Firstly, ancestors that are of the same relative age are not allowed to copy from each other, but they are computed consecutively. For this reason, *tsinfer* schedules them in parallel, as soon as their older predecessors finished computing. Since the inference algorithm sorts ancestors by their relative age, it can find ancestors of the same age easily. The authors call each set of ancestors of the same relative age an *epoch*. However, this parallelization has severe limitations. We observed from Section 3.3.2 that with a higher amount of samples, the amount of same-age ancestors shrinks.

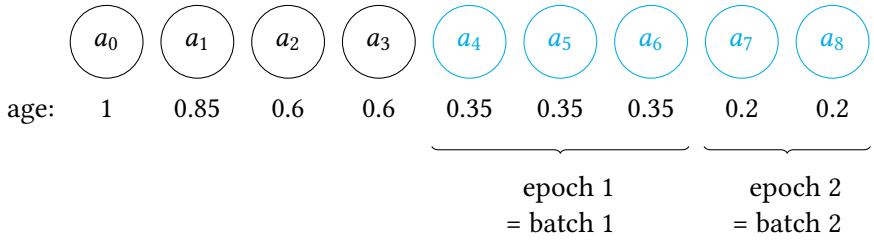
Secondly, Kelleher et al. [Kel+19] run a scanline algorithm to find incomplete ancestors that do not overlap, and are also not bridged by another ancestor that overlaps with both of them. This way, the scanline algorithm finds sequences that cannot possibly copy from each other and can thus be scheduled in parallel. This method is only feasible for old ancestors, though, because younger ancestors tend to cover most variant sites of the genomic sequence. For this reason, *tsinfer* aborts the scanline algorithm after a threshold age. In the following discussion, we will focus on the epoch-batching and ignore the additional batching achieved by the scanline algorithm.

As we saw in Section 3.4.1, the recombination probability depends on the number of states in the Markov chain, since the total recombination probability is distributed among all possible haplotypes. So if the amount of haplotypes from which the input haplotype can copy changes, the Viterbi algorithm will return a different most likely path. As we will discuss shortly, our parallelization approach does not depend on this scanline algorithm. We therefore chose to ignore it for our implementation, which slightly alters our results compared to *tsinfer*.

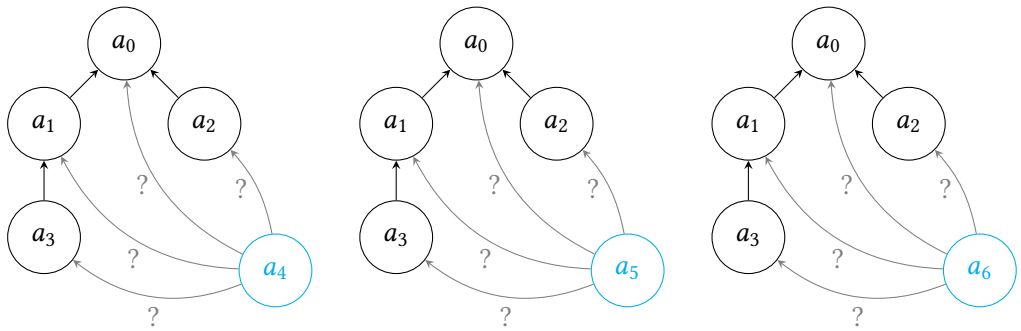
Our new approach to parallelization solves both of the aforementioned problems. We still implement the likelihood contraction discussed in Section 3.4.3, however, we do not require every haplotype in the Viterbi algorithm to already have edges in the tree sequence. Instead, we treat haplotypes that have not yet been matched into the tree sequence as uncontractible. They partake normally in the Viterbi algorithm, but never compare their likelihood to any other node and cannot leave the set of active ancestors.

This further changes the output tree sequence of the Viterbi algorithm. If two haplotypes have the same likelihood, we can just choose the older one for recombination, to mimic what *tsinfer* does implicitly. However, since we do not know the nodes' parents, it is possible that we choose an unrelated older ancestor, expecting it to become the parent. While the likelihood remains the same, the resulting tree sequence will differ from *tsinfer*. We further discuss this problem in Section 5.1.

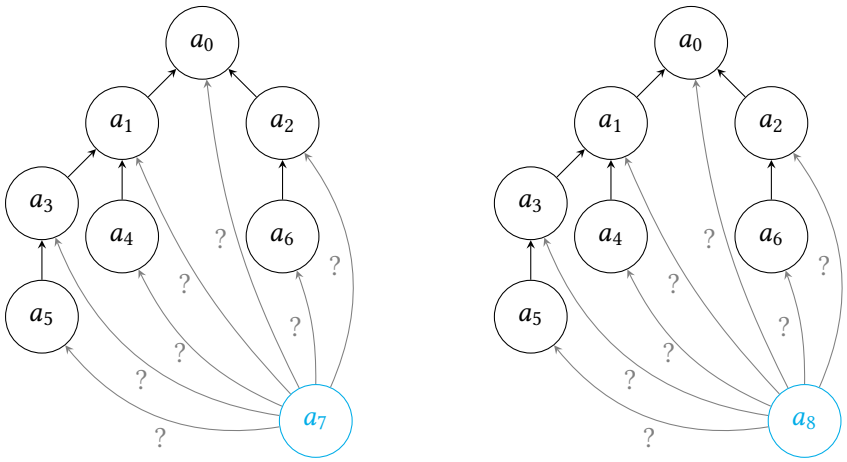
Moreover, it also means that the result can change depending on the amount of threads. Using more threads will result in larger parallel work units, which means there are more uncontractible ancestors per Viterbi instance.



(a) Of the nine ancestors, *tsinfer* already matched the oldest four (black) into a partial tree sequence. It schedules the next five (cyan) in two batches, according to their relative ages.

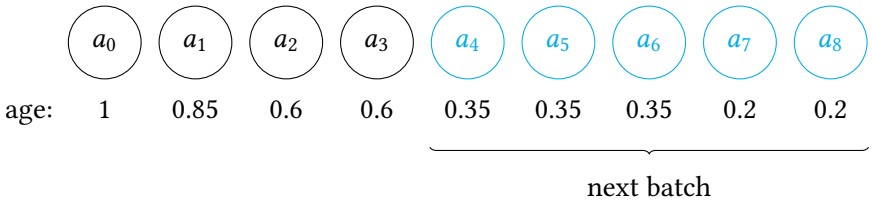


(b) The ancestors of the first batch all have the same relative age, and therefore cannot copy from each other. Thus, we can match them against older ancestors in parallel.

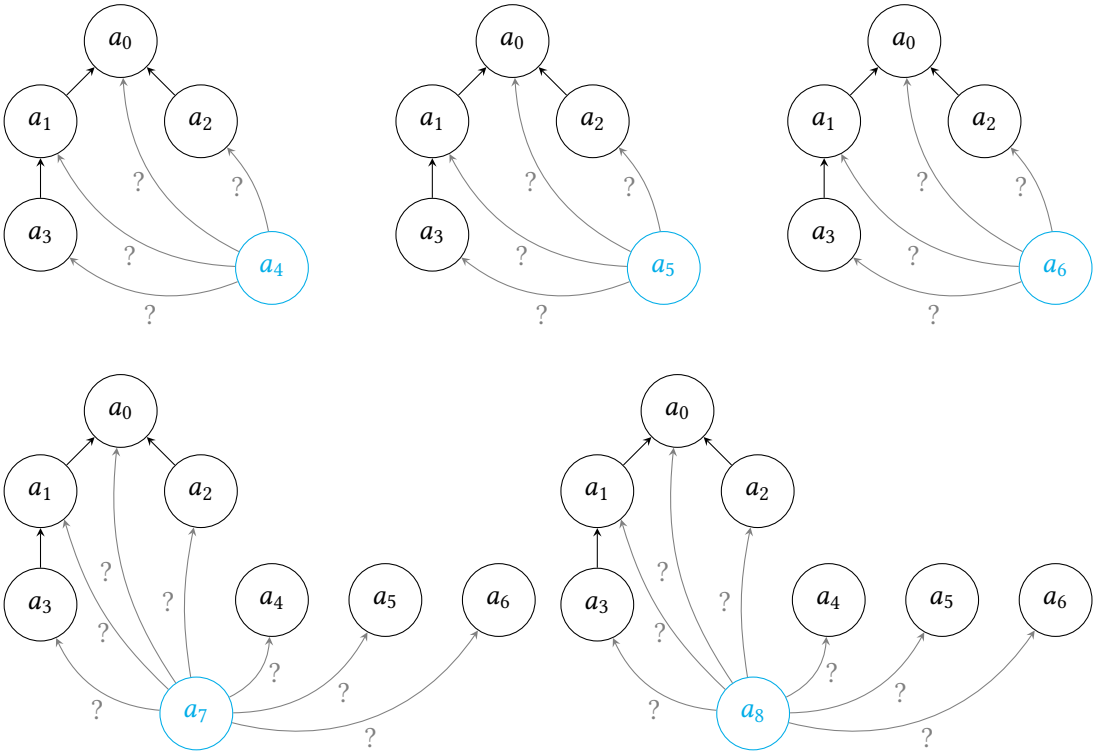


(c) The ancestors of the second batch all have the same relative age, and therefore cannot copy from each other. But they can copy from the first epoch, so the first epoch must already be present in all marginal trees.

Figure 3.5: Ancestors of equal relative age can be scheduled in parallel, because they cannot copy from each other. Because later epochs depend on the marginal trees comprised of earlier epochs for likelihood contraction, *tsinfer* can only parallelize within epochs.



(a) Of the nine ancestors, our proposed algorithm already matched the oldest four (black) into a partial tree sequence. It schedules the next five (cyan) in one batch, according to its batch size parameter.



(b) We match the first three ancestors of the batch (a_4 through a_6) analogously to *tsinfer*. Since a_7 and a_8 can copy from $a_4 \ldots a_6$, though, we add those three as additional nodes outside the marginal trees (because we have not inferred their parents yet). Within the Li-Stephens model, this yields equally likely results, but we cannot contract a_7 and a_8 during this batch.

Figure 3.6: With our approach, we can schedule ancestors independently of their relative age. As a result, however, some ancestors cannot be contracted during the Viterbi algorithm, increasing the number of redundant calculations we perform.

4 Implementation

With an understanding of the algorithm, we now present the most crucial implementation choices, comparing them against *tsinfer*. We implemented our proposed algorithm in the *Rust* programming language 2021 edition.

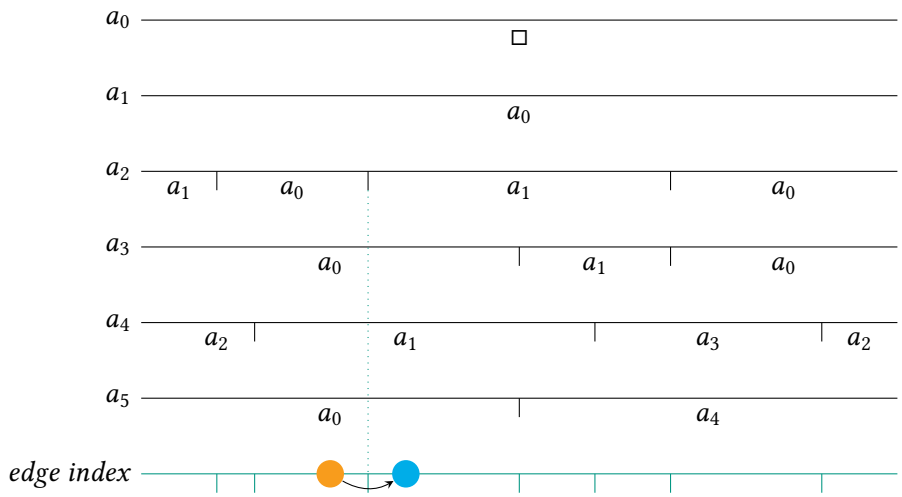
First, we discuss how Kelleher et al. [Kel+19] and we implemented the likelihood contraction introduced in Section 3.4.3 in Section 4.1. The table generated by the Viterbi algorithm (Section 2.1.1) is too large to be stored in memory, and thus we need to compress it (Section 4.2). Both likelihood contraction and the layout of the compressed table require a more complex Viterbi traceback phase, for which we present two approaches in Section 4.3. Further, in Section 4.4 we discuss *tsinfer's* and our approaches towards parallelizing the three algorithm phases. Finally, in Section 4.5, we discuss a potential tradeoff, reducing the number of likelihood contraction steps and, in return, performing redundant operations in the Viterbi algorithm.

4.1 The Tree Sequence Iterator

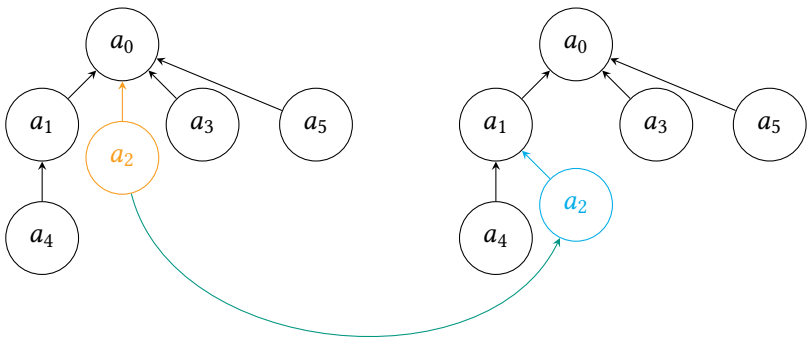
The likelihood contraction presented in Section 3.4.3 reduces the number of redundant floating point operations during the Viterbi algorithm, by contracting edges in the marginal tree (Section 2.3) with equal likelihoods. For this, the algorithm keeps track of the marginal trees at each site, which model the parent-descendant relationships between older ancestors (Section 3.4.2). Likelihoods of ancestors that are equal to their parent's likelihood are not explicitly calculated until the ancestor's genomic sequence differs from their parent. After each step in the Viterbi algorithm, the algorithm compares each ancestor's likelihood with their parent ancestor's likelihood, and if they match, it contracts the edge between both nodes.

Keeping track of parent-descendant relationships requires iterating over the marginal trees of the partial tree sequence during the Viterbi algorithm. Both implementations approach this by maintaining a mutable tree data structure representing the current marginal tree. The tree is modified by the iterator in between Viterbi iterations (i.e., between variant sites). As explained in Section 2.3, each edge of the marginal tree corresponds to an interval in an ancestor's genomic sequence copied from its parent ancestor. To iterate over all marginal trees thus requires iterating over the sorted intervals of all ancestors in the partial tree sequence. We call this sorted list the *edge index* (Figure 4.1a). Whenever an interval ends, the corresponding ancestor's node is moved from its current parent ancestor to the parent ancestor of the next interval (Figure 4.1b). If an ancestor ends before the genomic sequence ends (Section 3.3.4), the iterator removes it from the marginal tree.[1]

[1] It cannot have descendant ancestors at this point because it has likelihood zero in the Viterbi algorithm for sites where it is not defined, so removing a node does not split the tree.



(a) Six ancestor genomes (a_0 through a_5) form a tree sequence with their inferred genomic intervals. Two neighboring intervals induce a recombination event between them. All recombination events sorted by their genomic position form the edge index (green). We show the transition between marginal trees from the orange to the blue variant site in Figure 4.1b.



(b) The recombination event between the orange and blue variant site in Figure 4.1a moves ancestor node a_2 from a_0 to a_1 in the marginal tree.

Figure 4.1: An example of how the sorted sequence of all recombination events (*edge index*) can be used to update a tree data structure to represent the marginal tree at any genomic interval.

After an instance of the Viterbi algorithm completes, we insert the input haplotype's newly inferred intervals into the partial tree sequence, and into the edge index. So we need a data structure that supports fast iteration over its sorted elements and insertion of new elements. Insertion happens only after each *epoch* (Section 3.6), while iteration is performed during each Viterbi instance and is thus the most critical operation for performance.

Kelleher et al. [Kel+19] use an *AVL tree* [AL62] to maintain an ordered list of all edges, sorted by the start of their interval. Because *tsinfer* requires a list of edges that end at a site and a list of edges that start at a site, the algorithm maintains two AVL trees. We explain the reasoning in Section 4.3. Unfortunately, the AVL tree implementation contains only a single edge per node, impairing cache efficiency during iteration. To mitigate this, *tsinfer* linearizes both trees into arrays after each epoch, copying the data each time. The algorithm then iterates over the arrays during the Viterbi algorithm and maintains the AVL trees for fast insertion only.

Conversely, we use a combination of a *B-Tree* [BM70] and a *Fenwick Tree* [Fen94] implemented and published by Carneiro [Ruc23]. It achieves cache-efficient iteration by storing up to 1024 elements per leaf. Using Carneiro's tree data structure yielded speedups of up to 40 % over using the *B-Tree* implementation of *Rust's* standard library in preliminary experiments on human genomic data.

After each Viterbi iteration, the algorithm compares the likelihoods of all ancestor nodes and their next uncontracted ancestor node in the tree (Section 3.4.3). To find a node's next uncontracted ancestor, *tsinfer* walks up the marginal tree until it reaches an uncontracted node.

Intuitively, we assume that many unique paths in the tree are scanned during this operation. That is because dissimilar ancestors (that are less likely to be contracted, and therefore both contribute to the number of operations performed) are less likely to be in parent-descendant relationships, and the scans therefore cover more edges on their paths to the tree root. Consequently, we assume that each iteration of the modified Viterbi algorithm scans a substantial number of edges in the tree. On the other hand, we expect the marginal tree to only differ slightly between nearby variant sites, and thus we require few operations to update its data structure. To avoid frequent scans in the tree, we instead maintain a *contracted tree* data structure, which contains only the uncontracted nodes and edges of the marginal tree (Figure 4.2). Two nodes in the contracted tree are connected if one is the next uncontracted ancestor of the other. Whenever we update the marginal tree between variant sites, we also update the contracted tree.

4.2 The Viterbi Table

The Viterbi algorithm stores the most likely state transition for each Markov state in the Markov chain for each step in the Markov chain (Section 2.1.1). Since Kelleher et al. [Kel+19] exploit the symmetry of recombination probabilities in their modified algorithm (Section 3.4.2), it suffices to store the most likely Markov state per site in addition to the recombination and mutation flags. Unfortunately, with millions of mutations and thus millions of putative ancestors, the resulting table is still too large to store in main memory. However, due to likelihood contraction (Section 3.4.3), most entries in the table are empty because the respective ancestor is contracted. This allows for storing the table in a sparse format, which we explain in this section.

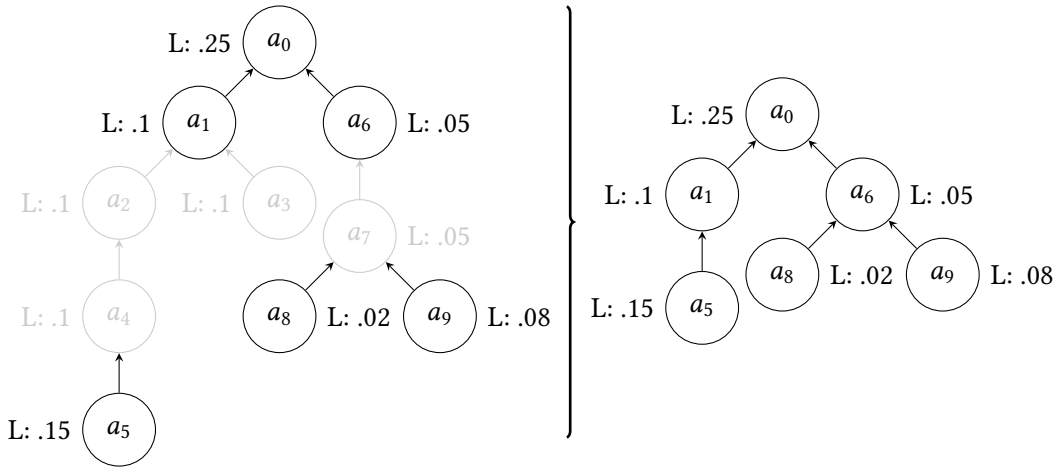
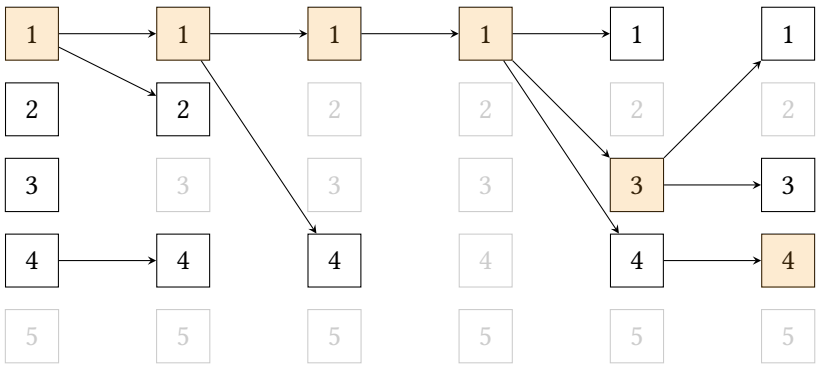


Figure 4.2: The contraction of edges between nodes with equal likelihood induces a *contracted tree*.

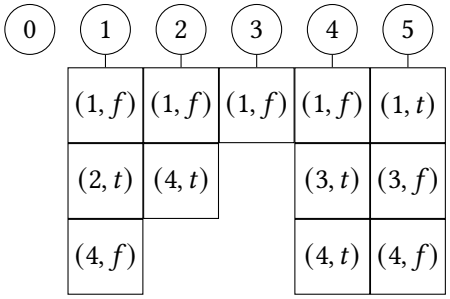
Kelleher et al. [Kel+19] store a list of tuples for each variant site. The list contains a tuple for each ancestor that is not contracted at this site, comprising an identifier for the ancestor and its recombination flag. During the traceback phase, *tsinfer* scans the marginal tree for the first uncontracted ancestor of the traced ancestor to reconstruct the flags. Although *tsinfer* could also store mutation flags this way, it does not need to because the traceback phase scans all variant sites, allowing reconstruction of the flags.

In contrast, our proposed algorithm avoids scanning all variant sites during the traceback phase (Section 4.3), necessitating a transposition of the memory layout. Instead of storing lists of flags per site, we store lists of flags per ancestor. Specifically, we store a linked list of recombination events, mutation events, and contraction events (i.e., an interval in which the ancestor is contracted) for each ancestor node. Most of these lists are short, because the ancestors are contracted for most of their genomic sequence (Section 3.4.3), only being active in short intervals where they carry new mutations compared to their parent ancestor). Every element in the list comprises a discriminator defining the event type, an interval length in case of contraction events, and a pointer to the previous element. Since we access the elements of a list only in reverse order (from the last site to the first), we require only one pointer per list element.

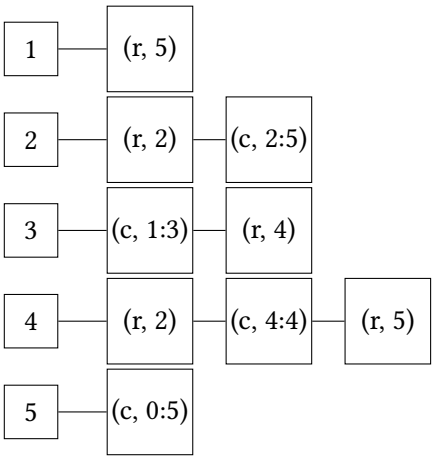
We use linked lists instead of an array of lists because we cannot easily predict the length of the lists. During the ancestor matching phase (Section 3.4, most ancestors in the Viterbi algorithm are contracted, and thus the algorithm adds no events to their list. However, we cannot easily predict which ancestors are contracted for which parts of their respective sequence. This problem is amplified when matching multiple incomplete ancestor genomes consecutively, since they are potentially defined for distinct intervals of the genome, with different contracted ancestors. Moreover, our approach to parallelization causes most ancestors to be uncontractible for at least a few instances of the Viterbi algorithm until they are matched into the tree sequence (Section 3.6). All these factors require that we re-allocate the lists with a high frequency, or accept large overheads for the reallocation of vectors. Instead, we found that an array of linked lists managed in a single memory allocation greatly benefits both performance and memory footprint. A future optimization may increase the count of events per linked list node to increase cache efficiency during traceback.



(a) The canonical table contains one cell per ancestor per variant site, storing the most likely transition that leads into that state. The most likely state at each site is marked orange, and needs to be stored separately, for both *tsinfer* and our algorithm.



(b) The table of *tsinfer* is an array of arrays containing a boolean recombination flag (t, f) for each uncontracted ancestor per variant site. A missing entry signals that the ancestor is contracted.



(c) Our table is an array of linked lists for each ancestor, containing recombination events, mutation events (omitted here), and contraction events.

Figure 4.3: The layouts of the sparse Viterbi table of *tsinfer* and our proposed algorithm. While *tsinfer* stores an array per variant site, we store data for sites only if there exists an event for this site.

4.3 The Viterbi Traceback Phase

Both, the contraction of nodes in the marginal tree (Section 3.4.3), and the contraction of the Viterbi table (Section 4.2) require special handling during the Viterbi traceback phase. Our approach differs from that of *tsinfer*, both of which we discuss in the following.

Because of *tsinfer*'s encoding of likelihoods into the marginal tree (Section 4.2), it requires the marginal tree data structure at each site to search for recombination and mutation events. Thus, the traceback phase of the Viterbi algorithm requires iterating backwards through the tree sequence. This is why *tsinfer* maintains two AVL trees for its edge list (Section 4.1): To iterate backwards through the edges, *tsinfer* needs access to the edges sorted by the end of their genomic interval. While tracing an ancestor through the Viterbi table, *tsinfer* requires access to its next uncontracted parent ancestor if the traced ancestor is contracted, since there is no data stored in the compressed table for contracted ancestors (Figure 4.3b).

To find the next uncontracted parent ancestor, *tsinfer* scatters the dense array stored in its table (Figure 4.3b) into a sparse array, where the array index of an entry corresponds to the ancestor identifier. With the marginal tree, *tsinfer* scans for the next uncontracted ancestor at every site by traversing towards the root of the tree, and testing if the corresponding array entry contains data. Once it finds an entry with data, *tsinfer* uses this data to decide whether a recombination is required. This scan is mandatory, even if the next uncontracted ancestor has not changed since the last computed variant site, because *tsinfer* stores no information about contraction (i.e., it cannot know if the next uncontracted ancestor has changed).

Conversely, our implementation of the traceback phase skips sites without mutation or recombination events. Instead, the linked list (*event list*) of the currently traced ancestor contains all mutation, recombination and contraction events in left-to-right order (Figure 4.3c). Thus, as we store all necessary information in these event lists, our proposed algorithm does not need to iterate through the tree sequence in the *traceback* phase, substantially reducing the overhead for maintaining the marginal tree. When we encounter a contraction interval in the currently traced ancestor, we instead look up the parent ancestor in the interval list (a_0 through a_5 in Figure 4.1a) of the ancestor. For all sites in the contraction interval, we now consider the event list of the parent ancestor instead. If the parent ancestor is contracted as well, we repeat the parent lookup in its own interval list. If we encounter a recombination event, our algorithm switches to the event list of the ancestor we recombined from, and continues iterating backward from the recombination site.

Overall, our implementation saves time by not updating the entire marginal tree at each site (Figure 4.1), but instead just search the interval lists (a_0 through a_5 in Figure 4.1a) when needed. Moreover, we scan upwards in the marginal tree only if the contracted tree layout actually changes (i.e., when we encounter a contraction event), instead of scanning at each site.

4.4 Multithreading

We discussed how *tsinfer* and our implementation approach parallelization in Section 3.6. Here, we give a quick overview of the implementation details.

For the *Ancestor Generation* phase, Kelleher et al. [Kel+19] push focal sites onto a queue, which a pool of worker threads is consuming. Workers push their generated ancestors into a common data structure guarded by a mutex. Our approach uses the *rayon*[2] thread

[2] crates.io: *https://crates.io/crates/rayon*, version: 1.10.0

pool implementation with work-stealing to consume the list of focal sites in parallel. When the algorithm finishes generating all ancestors, we sort the resulting list using an unstable parallel *quicksort* implementation of the *rayon* library. Notably, both implementations do not parallelize the merging and breaking of focal sites discussed in Section 3.3.3.

During ancestor matching (Section 3.4) and sample matching (Section 3.5), *tsinfer* generates batches, which are pushed into a queue. For the batches generated during the ancestor matching phase, Kelleher et al. [Kel+19] are using the epochs provided by the algorithms described in Section 3.6. A pool of workers drains the queue greedily, and the results are pushed into another priority queue, which keeps track of the order in which tasks are scheduled. When *tsinfer* finishes computing an epoch, it adds it to the edge index data structure (Section 4.1).

Our approach to the second phase is similar. We push batches into an inter-thread-communication channel,[3] and workers from a *rayon* thread pool greedily consume tasks from the queue. When the queue is empty, we join the threads and push the results into the edge index data structure. Because the core utilization depends on how often threads are joined, and how uniformly work is distributed across threads, we introduced a parameter to the algorithm to adjust the queue size. This is possible because the queue size is independent of epochs. A larger queue size leads to better core utilization and less frequent thread synchronization, however it increases the amount of redundant work because more ancestors in the queue are no longer contractible (Section 3.6). We discuss this tradeoff in detail in Section 5.3.2.

We handle the third phase analogously to the first phase with a work-stealing thread pool.

4.5 A Contraction Trade-Off

While testing with small data sets, we were able to speed up computation by performing the likelihood contraction (Section 3.4.3) only selectively. We observed the time spent scanning and contracting the marginal tree to be a non-trivial fraction of the time spent calculating the Viterbi paths. As already noted in Section 3.4.3, the number of uncontracted ancestors at any time is often an order of magnitude smaller than the total number of ancestors. Consequently, we tried reducing the amount of time spent contracting the marginal tree by decreasing the frequency with which we scan for contractible edges in the marginal tree. Specifically, we tested contracting every k'th step, where k is a parameter of the algorithm, as well as contracting only when the number of uncontracted ancestors reached a fraction of $\frac{1}{k}$ of the total amount of ancestors in the tree sequence. For small data sets, the latter approach worked best, gaining substantial speedups on select instances. We were, however, unable to reproduce the speed gains on larger instances without changing the parameters. Unfortunately, due to time constraints, we were not able to test whether the trade-off is viable with different parameters, or if it does not scale to larger instances at all.

It remains an open problem whether this tradeoff scales to real-world datasets, or whether the resulting time spent on redundant calculations eventually out-scales the time saved in the contraction step. We discuss this tradeoff and its synergy with another optimization again in Section 6.3.

[3] crates.io: *https://crates.io/crates/flume*, version: 0.11.0

5 Evaluation

We propose a novel parallelization scheme in Section 3.6. Then, in Chapter 4, we discuss several algorithmic improvements to the inference algorithm and compare them with *tsinfer's* implementation choices. In this chapter, we evaluate our implementation. First, we discuss how we determine the correctness of our implementation and point out its limitations (Section 5.1). Then, we present our experimental setup, with which we compare both implementations in Section 5.2. Finally, in Sections 5.3.1 to 5.3.3, we examine individual results and evaluate both the parallelization scheme and the algorithmic improvements.

5.1 Correctness

Our proposed algorithm generates different tree sequences than *tsinfer*. Our approach to parallelization causes our Viterbi implementation to choose different ancestors for recombination events than *tsinfer* (Section 3.6). Additionally, we did not implement the scanline batching algorithm that *tsinfer* uses in addition to epoch batching (Section 3.6). This choice potentially changes the parameter k in the Markov chain transition probabilities (Section 3.4.1) for ancestors that would be in different batches in *tsinfer*. Moreover, canonically, *tsinfer* chooses the most likely ancestor for recombination. If multiple ancestors are equally likely, and these ancestors form a subtree in the marginal tree, *tsinfer* chooses the oldest of these ancestors (Section 3.4.3). However, if two ancestors are equally likely but do not form a subtree in the marginal tree, an arbitrary, implementation-specific choice is made. Because this choice happens to depend on the order of matched ancestors, the inference algorithm generates different results if their order is changed.[1] Since both implementations make different arbitrary choices during ancestor generation, they do not choose the same ancestors for recombination.

For these reasons, our algorithm does not produce identical outputs to *tsinfer*, neither in single-threaded nor in multithreaded applications. It is therefore not trivial to prove the implementation's correctness. We compare generated tree sequences on small data sets generated with *msprime*[2] [Bau+21] and *stdpopsim*[3] [Adr+20]. For this, we disable *tsinfer's* path compression optimization (because we do not implement it), disable rounding (setting the precision parameter higher than 23), and manually disable the scanline batching (Section 3.6) in the Python module. We infer a full tree sequence from different synthetic datasets. In small datasets (a few hundred mutations and under 30 sample genomes), we achieve identical outputs to *tsinfer*.

To verify the correctness of larger results, we compare the input sample sequences and the haplotypes obtained from the mutations encoded in the tree sequence topology. Measuring the edit distance, we note on average a 3 % discrepancy between the base pairs of the input sample sequences and the sample sequences obtained from the tree sequence. We believe this to be the result of an implementation error in the handling of mutations. More importantly, we believe the runtime measurements to be unaffected by this error.

[1] We sort ancestors by their (discrete) age, but ancestors with the same age are arbitrarily ordered

[2] Version: 1.3.0

[3] Version: 0.2.0

To verify that our generated tree sequences are reasonable, we compare the parsimony scores of our computed tree sequences and *tsinfer's*, considering the number of mutations and the number of recombination events required to explain the observed sample sequences. Our results have on average a discrepancy of 10 % in the number of mutations and below 2 % in the number of recombination events to explain the observed sequences. Whether the resulting difference in recombination events is a limitation of our implementation or an inherent effect of our proposed changes remains an open problem.

While a more exhaustive evaluation of our approach is necessary, we nonetheless are of high confidence that our implementation is in principle correct and remaining differences are caused by minor bugs.

To verify this in the future, we want to repeat the measurements that were used to evaluate the results of *tsinfer*: Kelleher et al. [Kel+19] simulated topologies and generated haplotype information from those. Then, they added simulated measurement errors to the input data, and *tsinfer* and three other tools to infer ancestral recombination graphs inferred ancestries from the data. Afterward, the authors compared the resulting data to the simulated topology using tree distance measures. We need to compare the results of *airs* and *tsinfer* to simulated topologies to verify that our tool generates trees with comparable similarity scores. We do, however, leave these evaluations for future work.

5.2 Methods

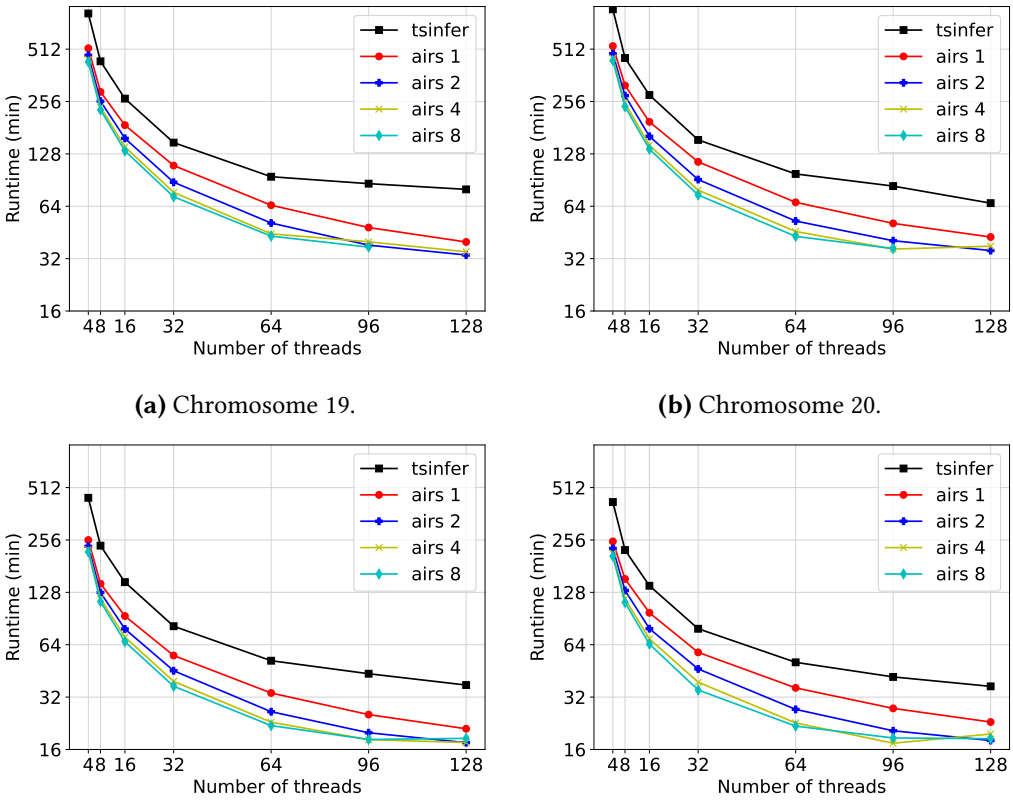
We evaluate our proposed algorithm (*airs*[4]) by measuring its runtime inferring tree sequences on human genome data from the *1000 Genomes Project* [Con+15], and comparing it to *tsinfer* processing the same input data. We measure the four smallest chromosomes (Chromosome 19 through Chromosome 22), and the largest Chromosome. We do not expect the behavior of the algorithm to differ substantially when processing the other genomes. Furthermore, each measurement requires multiple days of computation time, so to save on computational resources and corresponding CO_2 emissions, we did not measure chromosomes two through eighteen. The data comprises 5008 sample genomes, and between approximately 600 000 and 1 000 000 biallelic single nucleotide polymorphisms.

We also measure speedup by dividing the sequential runtime of *tsinfer* by the parallel runtime of *airs* and *tsinfer*. We expect our implementation to be faster sequentially (Section 5.3.3), but we do not have the necessary data available yet.

We extract biallelic SNP mutations from the VCF files and then convert them into the data formats expected by the two algorithms. We run the experiments on a double-socket AMD EPYC 7713 system with 128 logical cores per socket. The system has a NUMA architecture with 16 nodes comprising 2048 GB main memory. We measure the wall-clock times for all three phases of the inference algorithm individually, utilizing between 4 and 128 threads. Since our implementation has an additional parameter to control how big the work units during parallelization are, we evaluate parallelization using four values for the `per-thread` parameter: 1, 2, 4, and 8.

We configure *tsinfer* to use comparable parameters: We disable path compression, increase precision to 23 (disabling rounding), and set mutation and recombination probability to constant values. By default, *tsinfer* is compiled with the `-ffast-math` compiler flag, which

[4]GitHub repository: *https://github.com/Cydhra/libairs/tree/77d5220d848f6325148b48ee175c1528a957c531*



(a) Chromosome 19.

(b) Chromosome 20.

(c) Chromosome 21.

(d) Chromosome 22.

Figure 5.1: Runtime plots of the full inference benchmark on the four smallest chromosomes of the *1000 Genomes Project*. Our implementation is *airs*, and the number denotes the `per-thread` parameter. We consistently outperform *tsinfer*, especially at high thread counts. Higher `per-thread` values eventually stop scaling well due to redundant calculations, and some measurements with work unit sizes exceeding 256 ancestors are missing due to memory constraints. Lines for visual guidance only.

Rust does not stably support yet. Nonetheless, we left the option enabled for the test cases. The mutation rate in both algorithms is fixed at 1×10^{-20} and the recombination rate at 1×10^{-2}.

5.3 Results

In this section, we discuss the results of the measurements outlined in Section 5.2. We plot the measurements in Figures 5.1 to 5.6. In Section 5.3.1 we discuss the overall runtime of both implementations on the five data sets. Afterward, we examine the ancestor matching phase more closely (Section 5.3.2), and in particular the effects of our novel parallelization scheme. Finally, we discuss the measurements of the sample matching phase (Section 5.3.3).

5.3.1 Overall Runtime

We benchmark the runtime when inferring tree sequences for Chromosome 19 through Chromosome 22 of the *1000 Genomes Project* [Con+15] using between 4 and 128 threads. Our implementation outperforms *tsinfer* by a factor of 1.8 to 2.4 (Figure 5.1). Even at 128 threads and two ancestors scheduled per thread, *airs* is between 1.9 (Chromosome 20) and 2.4 (Chromosome 19) times faster. At a fixed `per-thread` parameter, this factor remains around 2 for lower thread counts. As expected, the runtime decreases with a higher `per-thread` parameter, due to higher core utilization. The speed gain decreases with higher parameter values, though, as the number of redundant calculations resulting from the uncontractible nodes increases (Section 3.6). We observe that the speed gain between four nodes per thread and eight nodes per thread becomes negligible: The speedup factor of using eight ancestors per thread does not exceed 2.2 at 32 threads. At higher thread counts, the speedup gets smaller for large `per-thread` parameters. This is most likely due to the high number of redundant calculations, but for the larger chromosomes, another explanation might be memory layout problems due to non-uniform memory access. We elaborate on this when we discuss the largest chromosome below.

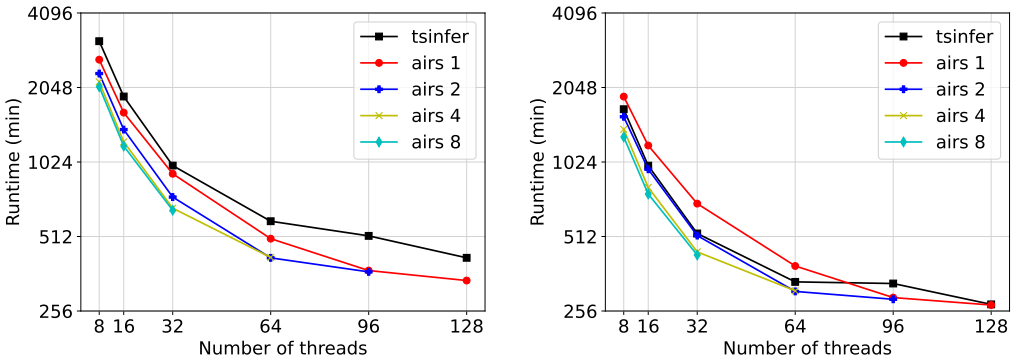
We run the same experiment on the largest human chromosome (Figure 5.2a). Here, we can observe that our implementation, while still faster than *tsinfer*, does perform worse in comparison to smaller chromosomes. We believe that this results from the high memory usage of our implementation. While inferring tree sequences for the two smallest chromosomes, the algorithm allocates around 1 TB (around 500 GB for low thread counts). Processing Chromosome 1, however, requires closer to 1.5 TB (more than 2 TB for large work units, which is why measurements for higher `per-thread` parameters are missing). We encounter similar problems with Chromosomes 19 and 20, which is why some of the measurements in Figures 5.1 to 5.5 are missing. Since the machine has non-uniform memory access, this high memory usage undoubtedly leads to adverse memory access times for the threads. Currently, sequences generated by our algorithm use one byte per site, even though the ancestral sequences can be represented using a single bit per site. Furthermore, the linked list (Section 4.2) wastes memory by storing a pointer per element instead of storing elements in linked blocks. We believe optimizing the memory usage will lead to reduced runtimes on larger inputs due to advantageous memory access patterns.

5.3.2 Speedup and Scaling in Phase 2

The second phase of the inference algorithm matches the inferred ancestors of Phase 1 into a tree sequence (Section 3.4). We discussed a novel parallelization scheme in Section 3.6). In this section, we examine the results of the ancestor matching phase of *tsinfer* and *airs* and discuss how our parallelization scheme affects the runtime.

In Figure 5.3, we observe similar runtime speedup patterns in Phase 2, as we do for the full algorithm (Figure 5.1). This is expected, since the parallelization of the sample matching is embarrassingly easy (Section 5.3.3; Figure 5.6), and so fundamental changes in the speedup of our algorithm compared to the speedup of *tsinfer* depend on the ancestor matching phase. In contrast, we expect the speedup of our implementation in the sample matching phase to depend only on the sequential runtime of the inference algorithm.

To compare the relative scaling better, we divide the runtime of *tsinfer* by the runtime of our implementation (Figure 5.4). We observe that the speedup of *airs* compared to *tsinfer* initially gets smaller, as the epoch parallelization approach of *tsinfer* initially scales better



(a) Runtime plot of the full inference for Chromosome 1.

(b) Runtime plot of the ancestor matching phase exclusively. Here, the high synchronization overhead of our implementation is especially evident at a `per-thread` parameter of 1 (red).

Figure 5.2: Tree sequence inference for Chromosome 1. The runtime improvements are smaller compared to smaller datasets, and lines for four and eight ancestors per thread end prematurely, because the machine does not have enough RAM. We believe the worse performance compared to smaller chromosomes to be an artifact of unfavorable non-uniform memory accesses. Lines for visual guidance only.

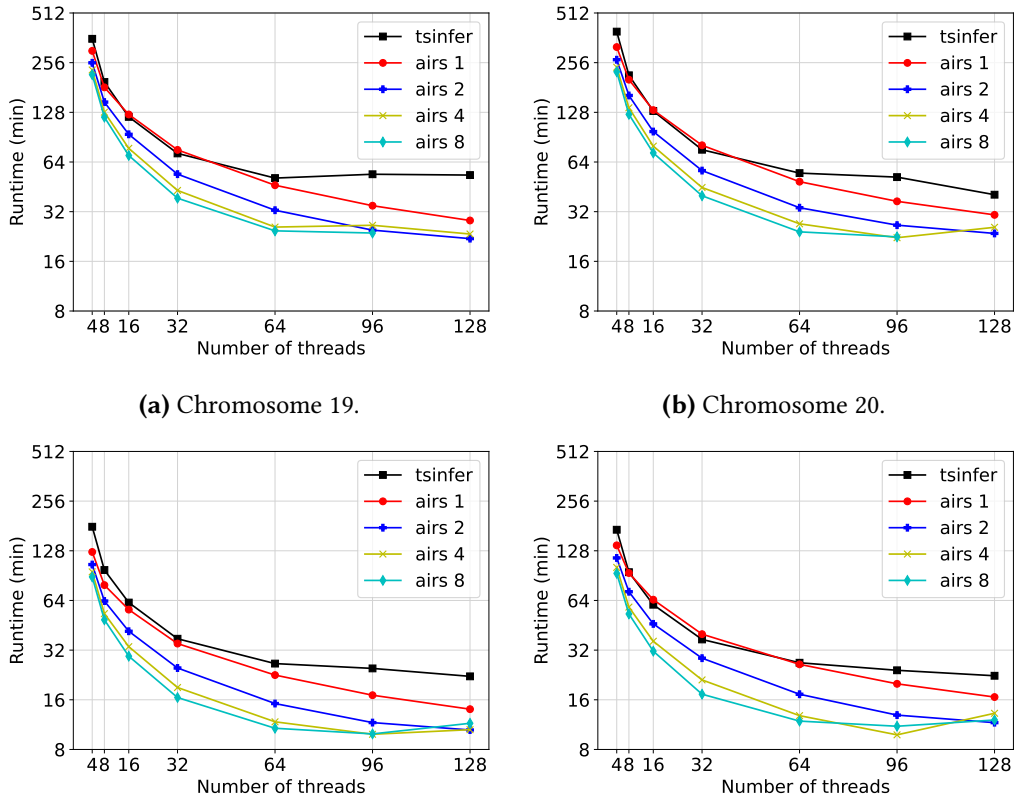
than our approach. Nevertheless, as thread counts exceed 16 (32 for low work unit sizes), the runtime of *airs* decreases faster than *tsinfer's* runtime, showing the better scaling we achieve by parallelizing independently of epochs. Eventually, the speedup begins to stagnate when the additional work our implementation has to perform outweighs the better core utilization. For small `per-thread` parameters, this happens around 96 to 128 threads. Inference instances with large values for the parameter actually lose their speed advantage at high thread counts, and are slower than instances with smaller work unit sizes.

We also compared the speedups of parallel *airs* and *tsinfer* against the sequential runtime of *tsinfer* (Figure 5.5). We can observe the same effects here: The speedup of *tsinfer* stagnates at high thread counts, while our implementation continues scaling even at 128 threads. Nonetheless, the speedup of 128 threads with four or eight scheduled ancestors per thread is lower than the speedup at 96 threads.

Ideally, we optimize the core utilization with smaller work units, so as to profit from the better scaling behavior with fewer redundant calculations. We discuss ideas to achieve that in Section 6.1 and Section 6.2.

5.3.3 Speedup in Phase 3

Phase 3 of the inference algorithm inserts the sample genomes into the tree sequence (Section 3.5). Because these sequences are not allowed to copy from each other, we can insert them in parallel without considering any data dependencies. Both *tsinfer* and our approach are based on greedy workers consuming a queue of all sequences and computing their parent sequences (Section 4.4). Thus, we do not expect the scaling of both algorithms to differ. Instead, any observed speedup should be the result of faster sequential computation.



(a) Chromosome 19.

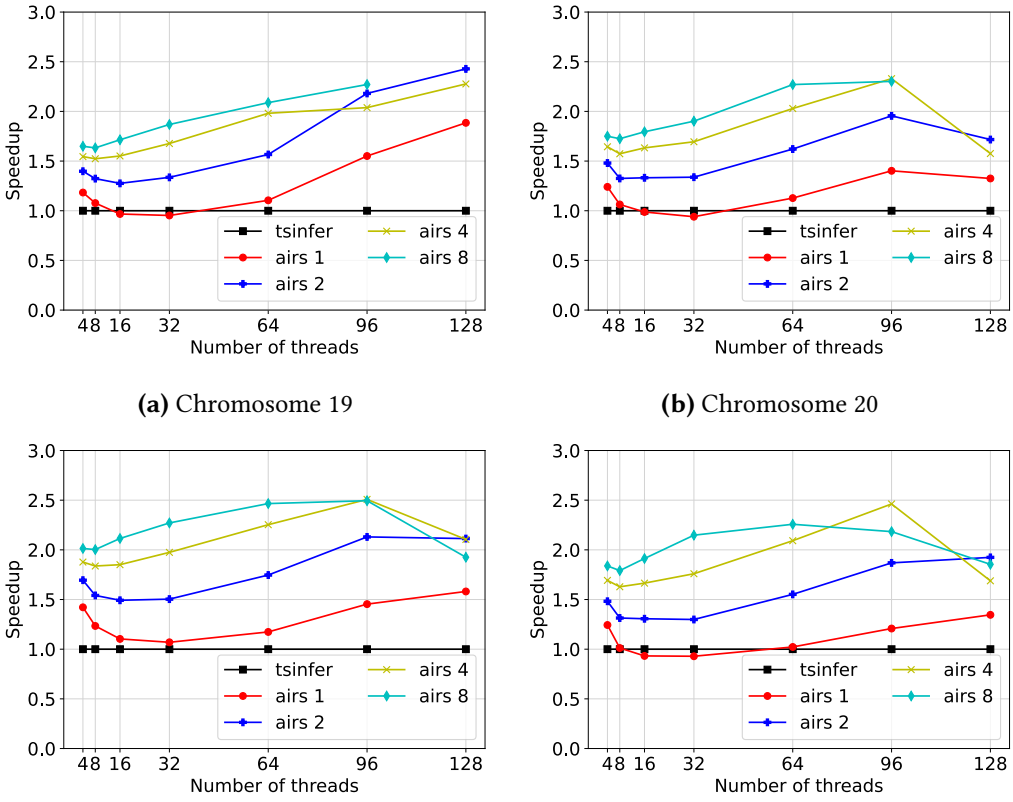
(b) Chromosome 20.

(c) Chromosome 21.

(d) Chromosome 22.

Figure 5.3: Runtime plots of the inference benchmark on the four smallest chromosomes of the *1000 Genomes Project*, considering only the ancestor matching phase. The improved scaling behavior of our parallelization approach is visible in the measurements at 64 to 128 threads. High work unit sizes (for example, eight ancestors per thread at 128 threads) perform worse than smaller sizes, because of the high number of redundant calculations. Lines for visual guidance only.

We measure the relative speedup of our implementation by dividing the runtime of *tsinfer* by our algorithm's runtime (Figure 5.6). As expected, the speedup remains nearly constant across all thread counts, because there is almost no linear component and no synchronization required. We therefore assume that this speedup is due to the algorithmic improvements discussed in Sections 4.1 to 4.3. The speedup averages around 2.2, and we assume that the single-threaded speedup is comparable because we do not expect the behavior to change with fewer threads. Unfortunately, our implementation currently cannot run single-threaded due to an implementation error, which we aim to fix in the future.



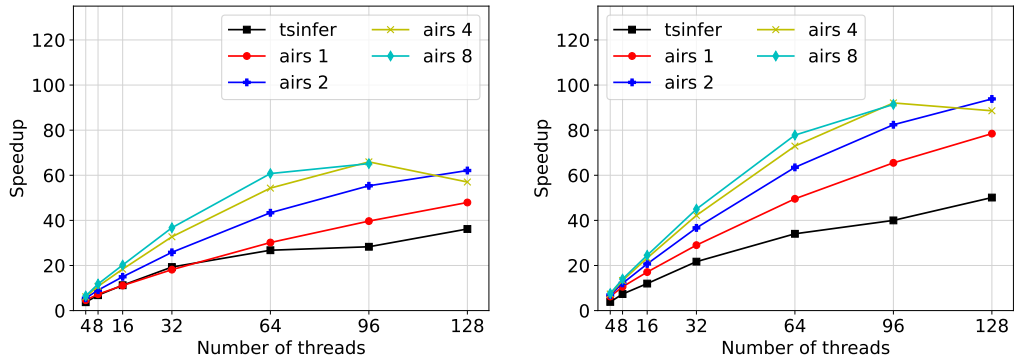
(a) Chromosome 19

(b) Chromosome 20

(c) Chromosome 21

(d) Chromosome 22

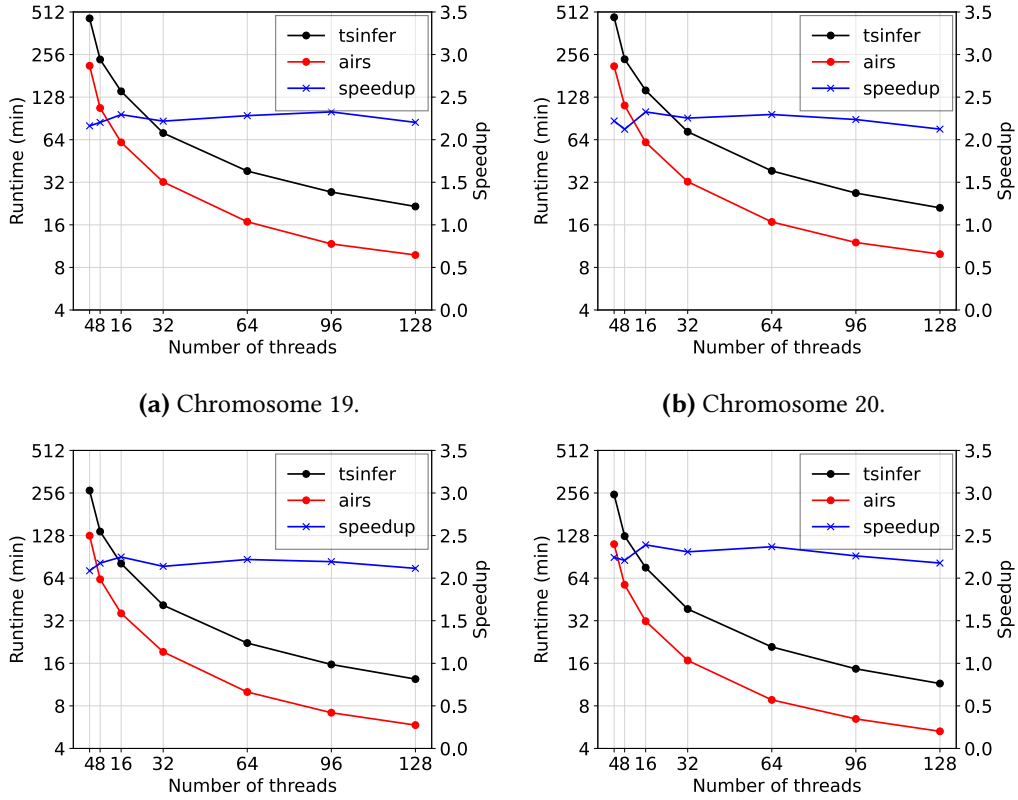
Figure 5.4: We divide the runtime of *tsinfer's* by the runtime of our algorithm in Phase 2 at equal thread counts to examine the scaling behavior; the y-axis shows the relative speedup of our implementation compared to *tsinfer*. We observe an initial advantage in scaling for *tsinfer*, while its epoch parallelization approach can effectively utilize the worker threads while avoiding redundant calculations. Afterward, our implementation scales better than *tsinfer* until the amount of redundant work outweighs the better thread utilization between 64 and 96 threads. We note, that higher `per-thread` parameters stop scaling earlier as the amount of redundant work grows faster with larger work units.



(a) Speedup of Phase 2

(b) Speedup of full inference algorithm

Figure 5.5: We divide the runtime of each measurement by the sequential runtime of *tsinfer* to obtain speedup plots of tree sequence inference on Chromosome 20. Because our implementation is faster sequentially, some measurements appear to have super-linear speedup. Lines for visual guidance only.



(a) Chromosome 19.

(b) Chromosome 20.

(c) Chromosome 21.

(d) Chromosome 22.

Figure 5.6: We calculated the speedup of our implementation compared to *tsinfer* in Phase 3 by dividing the runtime of *tsinfer* through the runtime of our algorithm. Because this phase is easily parallelized and requires no synchronization, we observe on average a near-constant speedup of 2.2 across all chromosomes, likely resulting from the sequential performance improvements. Lines for visual guidance only.

6 Future Work

We identify a range of open problems and ideas for improvement in our implementation in Sections 6.1 to 6.4. Finally, we comment on some of the heuristics in *tsinfer* and how they may impact the results in Section 6.5.

6.1 Offline Scheduling

Our results (Section 5.3.1) show that our parallelization scheme scales worse when we schedule work in smaller units, because the threads spend more time waiting at a synchronization barrier. We currently mitigate this problem by increasing the work unit size with the `per-thread` parameter, which, however, yields diminishing returns, because more incompressible ancestors (Section 3.4.3) induce more redundant calculations during the Viterbi algorithm. W note, however, that we can estimate the work of a single step in the *Ancestor Matching* phase by the length of the ancestor haplotype, and the amount of incompressible ancestors in the Markov chain (Section 3.4). Therefore, we hope to decrease the time threads spend idling by implementing an offline scheduler that assigns work to threads utilizing estimates about the required work.

6.2 Splitting Ancestor Genomes

Additionally, another problem we encountered when working with our greedy approach to assigning work to threads (Section 4.4) is ancestors differing in length. Intuitively, older ancestor tend to be shorter than younger ancestors, since we have less information (Section 3.3.4) about their genomic sequence. However, we cannot easily find limits on how imbalanced the lengths of ancestors can become. Scheduling c ancestors onto d threads may cause large imbalances if a scheduler encounters a bad distribution of ancestor lengths that it cannot evenly distribute among threads.

We considered splitting ancestor haplotypes into slices and assigning those to different threads. Naively, this is impossible because the new likelihoods of all Markov states depend on their previous likelihoods (Section 3.4.2). Nonetheless, by the intuition that distant mutations have little to no effect on the current likelihood (Section 3.4.1), we assume that only a prefix of each slice yields wrong results (i.e., the wrong most likely state and the wrong recombination flags). That is because we start with the wrong likelihood distribution at every but the first slice, as if we had exchanged everything before the slice with another genomic sequence. But over time, the likelihoods should approach the likelihoods we would see if we had not divided the sequence into slices because we expect the influence of the wrong initial values to decline. At some point, the likelihoods are presumably so close to expected values that subsequent results are correct. If we recompute the wrong prefix as a suffix of the previous

slice, we can reconstruct the Viterbi paths (Section 2.1.1) of the entire genome. Because our current approach seemed promising enough, and we did not find an easy model to estimate the length of wrong prefixes, we did not explore this idea further.[1]

6.3 Tradeoff between Likelihood Contraction and Redundant Work

In Section 4.5, we explored a tradeoff between the contraction of ancestor nodes in small data sets. Canonically, *tsinfer* attempts to contract edges in the marginal tree, if both ancestor nodes have the same likelihood (Section 3.4.3). The Viterbi algorithm checks for potential contractions after each of its iterations (Algorithm 3.1) by comparing all uncontracted ancestors in the marginal tree to their next uncontracted parent. Because *tsinfer* maintains no additional data structure for the contracted marginal tree, this requires walking up the marginal tree for each uncontracted node. Consequently, we were able to gain speedups by performing the likelihood contraction step for only a subset of the Viterbi iterations (Section 4.5). In preliminary experiments, however, the runtime gains we archived on smaller datasets using this technique did not carry over to real-world datasets. Evaluating if a different choice of hyperparameters leads to speedups even on these large real-world datasets is an open challenge.

6.4 Vectorization

Our approach to parallelization parallelizes the Viterbi algorithm instances (Section 3.6). The Viterbi algorithm forward phase consists of two nested loops, looping over variant sites and ancestors, respectively (Section 3.4.2). The innermost loop performs the likelihood calculations of Algorithm 3.1, which can be vectorized using the AVX-512 *gather* instruction to load the likelihoods of uncontracted nodes into SIMD registers. Then, likelihood calculations can be performed in parallel in those vector registers. Unfortunately, modern x86 processors usually reduce their clock frequency on certain AVX instructions, potentially slowing surrounding instructions [GKB20]. Because likelihood contraction (Section 3.4.3) happens between iterations of the Viterbi algorithm, vectorization of the loop can reduce the speed of likelihood contraction. Therefore, this optimization is especially interesting when combined with the tradeoff discussed in Section 6.3.

6.5 Ad Hoc Heuristics

While Kelleher et al. base the inference algorithm on a sound mathematical model (Section 3.4.1), they solve numerous problems with ad hoc heuristics. This includes the ancestor generation (Figure 3.3), the method of determining the relative age of mutations (Section 3.3.2), and the selection of equally likely ancestors during inference (Section 3.4.2).

During ancestor generation, mutations are extended into putative ancestor sequences by a majority vote among sample genomes that carry this mutation. During the extension, the algorithm removes sample genomes from this voting algorithm using empirically chosen

[1] If the Markov chain from Section 3.4.1 was ergodic, we could estimate the prefix length using its convergence speed to its stationary distribution. But because Li and Stephens [LS03] modelled non-constant recombination probabilities, the Markov chain has a state per variant site per ancestor, and each state can only transition into states of the subsequent variant site, sacrificing its ergodicity.

heuristics (Figure 3.3). Similarly, it halts extension using empirically chosen threshold values. Since the inference uses the ancestor sequences as its input, the quality of its inferred tree sequences hinges on the quality of the generated ancestor sequences. Therefore, an alternative algorithm for ancestor generation based on a well-researched biological model might yield ancestries closer to the ground truth.

The relative age of ancestors is determined with a heuristic based on biological rationale (Section 3.3.2). The approximation, however, allows only for a coarse discrete distribution. Specifically, the frequency of a mutation among all sample genomes serves as a proxy for its relative age. As a result, given n sample genomes, only n values for an ancestor's relative age are possible ($\frac{1}{n}, \frac{2}{n}, \ldots, \frac{n}{n}$). If the number of mutations is much higher than n, many mutations have the same age, which excludes them from being descendants of each other in any inferred tree. This approach severely restricts the resulting tree topology, without being rooted in a biological rationale. A better model might, again, increase output quality.

Lastly, in the *Li-Stephens* model discussed in Section 3.4.1, the algorithm often has to choose between multiple equally likely ancestors for recombination. The current heuristic is to always select the oldest ancestor. This decision has several advantages for the algorithm implementation: (1) The resulting trees are flatter, shortening time spent scanning up the tree during likelihood contraction (Section 3.4.3) and the Viterbi algorithm (Section 4.1 and Section 4.3). (2) Likelihood contraction (Section 3.4.3) can contract all equally likely nodes into their lowest common ancestor, which requires comparisons only with a node's parent. Other methods also require comparisons with sibling nodes. (3) The Viterbi traceback phase (Section 3.4.2), requires no access to contracted nodes, since the least common ancestor is the oldest ancestor. So no information about contracted nodes has to be stored. However, there is no biological reason to select the oldest ancestor, especially since there are likely many generations between ancestors. For example, an alternative algorithm might choose uniformly at random between equally likely ancestors. More research is needed to ascertain if alternative methods of selecting between those ancestors yield better results.

7 Conclusion

Tree sequences are a data structure in the field of population genetics used to describe and analyze the evolutionary history of sexually reproducing populations. With *tsinfer*, Kelleher et al. [Kel+19] provide a tool that infers tree sequences for orders of magnitude more genomes than previous tools that infer ancestral recombination graphs. For example, *tsinfer* is able to infer a tree sequence for a human chromosome with 5000 sample sequences in two hours [Kel+19].

In this work, we analyze the architectural choices of *tsinfer* and identify a range of potential optimizations: The tree sequence iterator *tsinfer* utilizes during the Viterbi algorithm uses two AVL trees, which regularly copy their nodes into an array. We use a cache-aware B-tree implementation instead, allowing for fast insertion and iteration without unnecessarily copying the data. Furthermore, we transpose the data layout for the sparse table of the Viterbi algorithm, allowing for a more efficient traceback phase where we save tree traversal operations at genomic sites that require no recombination events. Additionally, by maintaining a contracted tree data structure during the Viterbi algorithm, we further reduced the amount of tree traversal operations during the Viterbi forward phase.

Moreover, we identified a weakness in *tsinfer's* parallelization of the ancestor matching phase, which limits the core utilization by the number of same-age ancestors. We proposed a novel parallelization scheme, decoupling its degree of parallelism from the number of same-age ancestors. As a tradeoff, we allow threads to perform some redundant computations while increasing the core utilization.

We implemented our proposed changes in our own experimental tool, named *airs*, and measured its runtime for inferring tree sequences on human genomic data from the *1000 Genomes Project*, comparing it with the runtime of *tsinfer*. Our implementation is consistently faster by a factor of up to 2.4 with total runtimes at 128 threads ranging from 18 minutes for the smallest chromosome to five and a half hours for the largest. Furthermore, our parallelization scheme improves the runtime of *airs* compared to *tsinfer* at equal thread counts between 32 and 96 threads.

Additionally, we identify potential further improvements to our implementation. Specifically, we plan on scheduling ancestors based on estimates about their required work to reduce the overhead of thread synchronization. This way, we expect to reduce the size of work units and thereby the amount of redundant calculations, further improving scaling to high core counts. Moreover, we propose a potential tradeoff between likelihood contraction and additional redundant calculations, which may reduce the overhead of the likelihood contraction optimization. This tradeoff may also reduce the negative impact of SIMD operations in the Viterbi loop, allowing vectorization of its likelihood calculations.

With the rapidly decreasing costs of whole genome sequencing, the amount of genetic data available to conduct population-scale studies increases fast. With our contribution, we decrease the computational costs to infer ancestral data over large datasets, and more efficient utilization of available hardware.

Bibliography

[Adr+20] Jeffrey R Adrion, Christopher B Cole, Noah Dukler, Jared G Galloway, Ariella L Gladstein, Graham Gower, Christopher C Kyriazis, Aaron P Ragsdale, Georgia Tsambos, Franz Baumdicker, Jedidiah Carlson, Reed A Cartwright, Arun Durvasula, Ilan Gronau, Bernard Y Kim, Patrick McKenzie, Philipp W Messer, Ekaterina Noskova, Diego Ortega-Del Vecchyo, Fernando Racimo, Travis J Struck, Simon Gravel, Ryan N Gutenkunst, Kirk E Lohmueller, Peter L Ralph, Daniel R Schrider, Adam Siepel, Jerome Kelleher, and Andrew D Kern. "A community-maintained standard library of population genetic models". In: *eLife* Volume 9 (2020). Edited by Graham Coop, Patricia J Wittkopp, John Novembre, Arun Sethuraman, and Sara Mathieson, e54967. ISSN: 2050-084X. DOI: *https://doi.org/10.7554/eLife.54967*.

[AL62] Georgii M Adelson-Velskii and Evgenii Mikhailovich Landis. "An algorithm for the organization of information". Russian and English. In: *Proceedings of the USSR Academy of Sciences*. Trans. by F M Goodspeed. Vol. 3. 1962, p. 4.

[Bau+21] Franz Baumdicker, Gertjan Bisschop, Daniel Goldstein, Graham Gower, Aaron P Ragsdale, Georgia Tsambos, Sha Zhu, Bjarki Eldon, E Castedo Ellerman, Jared G Galloway, Ariella L Gladstein, Gregor Gorjanc, Bing Guo, Ben Jeffery, Warren W Kretzschumar, Konrad Lohse, Michael Matschiner, Dominic Nelson, Nathaniel S Pope, Consuelo D Quinto-Cortés, Murillo F Rodrigues, Kumar Saunack, Thibaut Sellinger, Kevin Thornton, Hugo van Kemenade, Anthony W Wohns, Yan Wong, Simon Gravel, Andrew D Kern, Jere Koskela, Peter L Ralph, and Jerome Kelleher. "Efficient ancestry and mutation simulation with msprime 1.0". In: *Genetics* Volume 220 (Dec. 2021), iyab229. ISSN: 1943-2631. eprint: *https://academic.oup.com/genetics/article-pdf/220/3/iyab229/43780247/iyab229.pdf*.

[BM70] Rudolf Bayer and Edward M. McCreight. "Organization and maintenance of large ordered indices". In: *Acta Informatica* Volume 1 (1970), pp. 173–189. DOI: *https://doi.org/10.1145/1734663.1734671*.

[Con+15] 1000 Genomes Project Consortium et al. "A global reference for human genetic variation". In: *Nature* Volume 526 (2015), p. 68. DOI: *https://doi.org/10.1038/nature15393*.

[Fen94] Peter M Fenwick. "A new data structure for cumulative frequency tables". In: *Software: Practice and experience* Volume 24 (1994), pp. 327–336. DOI: *https://doi.org/10.1002/spe.4380240306*.

[GKB20] Mathias Gottschlag, Yussuf Khalil, and Frank Bellosa. "Dim Silicon and the Case for Improved DVFS Policies". In: *ArXiv* Volume abs/2005.01498 (2020). DOI: *https://doi.org/10.48550/arXiv.2005.01498*.

[Gor87] A. D. Gordon. "A Review of Hierarchical Classification". In: *Journal of the Royal Statistical Society. Series A (General)* Volume 150 (1987), pp. 119–137. ISSN: 0035-9238. DOI: *https://doi.org/10.2307/2981629*.

[GT98] R.C. Griffiths and Simon Tavaré. "The age of a mutation in a general coalescent tree". In: *Communications in Statistics. Stochastic Models* Volume 14 (1998), pp. 273–295. eprint: *https://doi.org/10.1080/15326349808807471*.

[Hud+90] Richard R Hudson et al. "Gene genealogies and the coalescent process". In: *Oxford surveys in evolutionary biology* Volume 7 (1990), p. 44. eprint: *https://api.semanticscholar.org/CorpusID:82106985*.

[HWL17] Uwe Hossfeld, Elizabeth Watts, and Georgy Semeonovich Levit. "The First Darwinian Phylogenetic Tree of Plants." In: *Trends in plant science* Volume 22 (2017), pp. 99–102. DOI: *https://doi.org/10.1016/j.tplants.2016.12.002*.

[JSSM21] Murukarthick Jayakodi, Mona Schreiber, Nils Stein, and Martin Mascher. "Building pan-genome infrastructures for crop plants and their use in association genetics". In: *DNA Research: An International Journal for Rapid Publication of Reports on Genes and Genomes* Volume 28 (2021). DOI: *https://doi.org/10.1093/dnares/dsaa030*.

[Kel+19] Jerome Kelleher, Yan Wong, Anthony W Wohns, Chaimaa Fadil, Patrick K Albers, and Gil McVean. "Inferring whole-genome histories in large population datasets". In: *Nature genetics* Volume 51 (2019), pp. 1330–1338. DOI: *https://doi.org/10.1038/s41588-019-0483-y*.

[KO73] Motoo Kimura and Tomoko Ohta. "The Age of a Neutral Mutant Persisting in a Finite Population". In: *Genetics* Volume 75 (Sept. 1973), pp. 199–212. ISSN: 1943-2631. eprint: *https://academic.oup.com/genetics/article-pdf/75/1/199/34393389/genetics0199.pdf*.

[LS03] Na Li and Matthew Stephens. "Modeling Linkage Disequilibrium and Identifying Recombination Hotspots Using Single-Nucleotide Polymorphism Data". In: *Genetics* Volume 165 (Dec. 2003), pp. 2213–2233. ISSN: 1943-2631. eprint: *https://academic.oup.com/genetics/article-pdf/165/4/2213/42054068/genetics2213.pdf*.

[Rei+02] David Reich, Stephen Schaffner, Mark Daly, Gil McVean, James Mullikin, John Higgins, Daniel Richter, Eric Lander, and David Altshuler. "Human genome sequence variation and the influence of gene history, mutation and recombination". In: *Nature genetics* Volume 32 (Oct. 2002), pp. 135–42. DOI: *10.1038/ng947*.

[RHGS14] Matthew D Rasmussen, Melissa J Hubisz, Ilan Gronau, and Adam Siepel. "Genome-wide inference of ancestral recombination graphs". In: *PLoS genetics* Volume 10 (2014), e1004342. DOI: *https://doi.org/10.1371/journal.pgen.1004342*.

[Ruc23] B C D L Rucy. "indexset". GitHub. 2023. URL: *https://github.com/brurucy/indexset*.

[SH05] Yun S. Song and Jotun Hein. "Constructing Minimal Ancestral Recombination Graphs". In: *Journal of computational biology : a journal of computational molecular cell biology* Volume 12 (2005), pp. 147–169. DOI: *https://doi.org/10.1089/cmb.2005.12.147*.

[Sti+24] Josefin Stiller, Shaohong Feng, Al-Aabid Chowdhury, Iker Rivas-González, David A Duchêne, Qi Fang, Yuan Deng, Alexey Kozlov, Alexandros Stamatakis, Santiago Claramunt, et al. "Complexity of avian evolution revealed by family-level genomes". In: *Nature* Volume 629 (2024), pp. 851–860. DOI: *https://doi.org/10.1038/s41586-024-07323-1*.

[Vit67] A. Viterbi. “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm”. In: *IEEE Transactions on Information Theory* Volume 13 (1967), pp. 260–269. DOI: *https://doi.org/10.1109/TIT.1967.1054010*.

[Wu11] Yufeng Wu. “New Methods for Inference of Local Tree Topologies with Recombinant SNP Sequences in Populations”. In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* Volume 8 (2011), pp. 182–193. ISSN: 1545-5963. DOI: *https://doi.org/10.1109/TCBB.2009.27*.