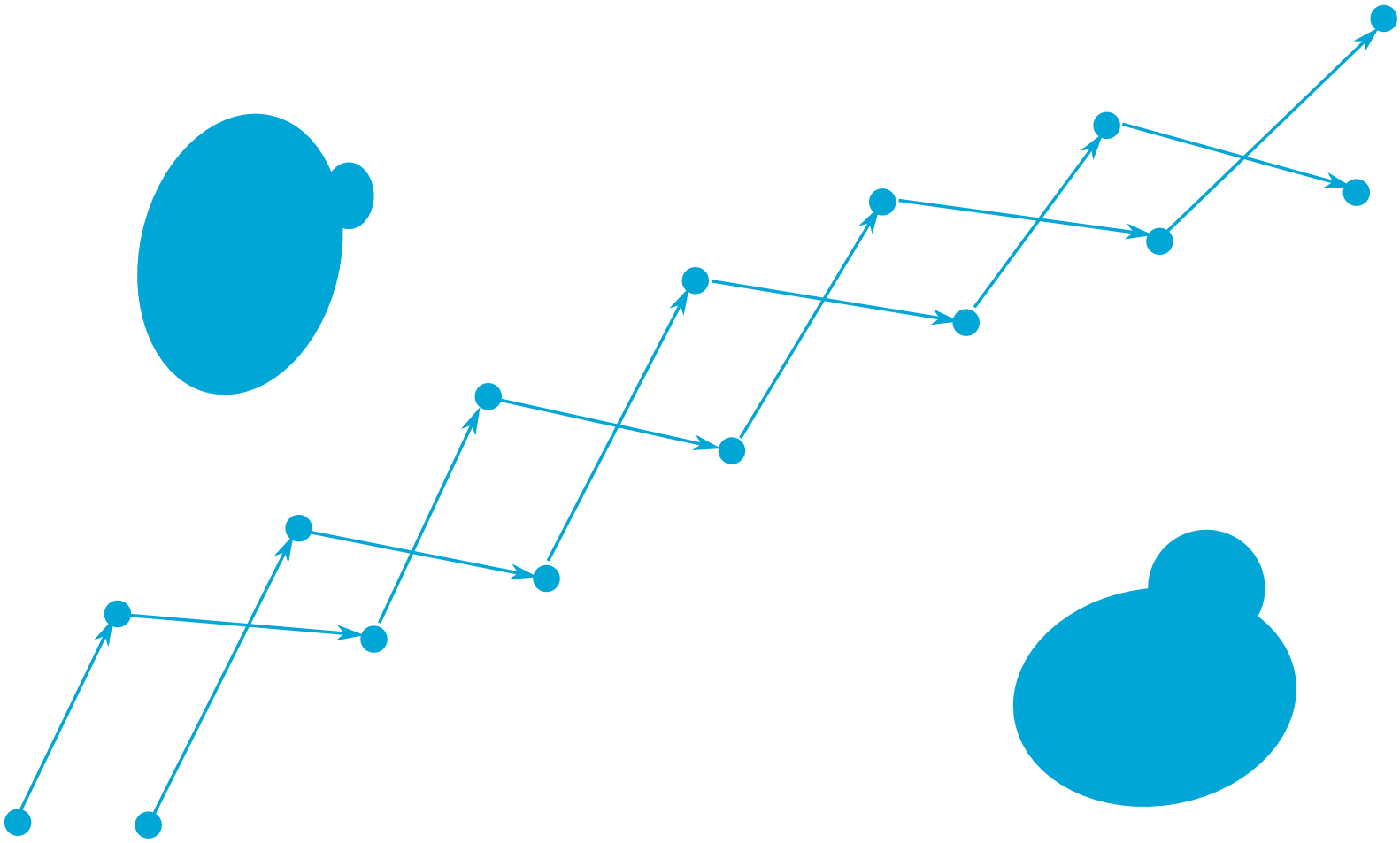
A Bayesian approach to haplotype-aware de novo genome assembly for polyploid organisms

L.R. van Dijk



TU Delft

A Bayesian approach to haplotype-aware *de novo* genome assembly for polyploid organisms

Lucas Roeland van Dijk
Delft University of Technology

Thesis Committee
prof. dr. ir. Marcel Reinders
dr. Thomas Abeel (supervisor)
dr. ir. Leo van Iersel

ABSTRACT Capturing all genetic variation within a polyploid organism is a challenge. Most current *de novo* assemblers have no notion of the concept “ploidy”. Consequently, when assembling the genome of diploid or higher ploidy organisms, the assembler mixes reads coming from either chromosome copy and builds a single DNA sequence representing an arbitrary composition of a set of homologous chromosomes. This hampers any downstream analysis, such as allele specific expression analysis, gene association studies or population genetics. Current haplotype assembly methods focus on phasing single nucleotide variants (SNVs), and are limited in their ability to deal with larger and structural variants.

The third generation sequencing platforms, like the PacBio RS II or the Oxford Nanopore, make long read sequencing affordable. In this work we explore a novel method using long reads and an assembly graph to phase haplotypes. We introduce PHASM: a prototype *de novo* genome assembler that outputs separate DNA sequences for each haplotype. We show that phasing using an assembly graph has potential, but that approximate alignments between reads introduces caveats that make this problem difficult.

KEYWORDS Genome Assembly; Haplotype Phasing

Table of Contents

1. Introduction

1.1. Context

A challenge of the 21st century is to find an alternative to fossil fuels, and biotechnology can aid in the search to an alternative. For example, there are several industrial plants using engineered strains of the yeast *Saccharomyces cerevisiae* to convert biomass to the biofuel bioethanol [1]. But, the production and transport of bioethanol is limited because of ethanol toxicity in *Saccharomyces cerevisiae* [2].

A solution to increase ethanol resilience may be hidden in the genome of the related yeast *Saccharomyces pastorianus*. This hybrid between *Saccharomyces cerevisiae* and *Saccharomyces eubayanus* is used for brewing lager-type beers, and has evolved over the past 500 years with human selection to withstand high concentrations of ethanol [3–5].

The hybridisation event between the two species resulted in a highly variable genome, and large variation is observed between the several hundred known strains of *S. Pastorianus* [6]. There are differences in chromosome and gene copy number; duplicate genes are allowed to specialise and become paralogs with their own function and expression profiles; and this is in addition to any variation on the single nucleotide level, affecting alleles of gene copies within a given strain. All this variation results in a diverse set of phenotypic traits among known strains [3, 6, 7]. We are interested in the genetic variation that drives ethanol resilience.

Capturing the genetic variation within a single strain, or between multiple strains, is a challenge: most current *de novo* genome assemblers have no notion of the concept ploidy. Consequently, when assembling the genome of diploid or higher ploidy organisms, the assembler mixes reads coming from either chromosome copy and builds a single DNA sequence representing an arbitrary composition of a set of homologous chromosomes [8, 9]. Any variations between these chromosome homologues, e.g. single nucleotide variants (SNVs), gene copy number variations or other variants, are either mixed with variants from other homologous chromosomes or not included at all. This affects any downstream analysis, for example allele specific expression analysis, gene association studies, or population genetics [10, 11]. This challenge will become more important with the increasing demand in non-model diploid and polyploid reference genomes, assemblies for highly rearranged disease samples, and pan-genome analyses [9, 12, 13].

The third generation sequencing technology, like the PacBio RS II or the Oxford Nanopore, make long read sequencing affordable. These long reads solve the traditional issue in *de novo* genome assembly, resolving repetitive regions in the genome, which results in highly contiguous and complete assembled genomes [14, 15]. Furthermore, they are useful for resolving haplotypes: if a set of consecutive alleles are observed in a single read, it is possible to infer that these alleles are on the same chromosome. In this work we explore a novel method to use long read data for phasing variants in a polyploid organism.

1.2. Related work

1.2.1. Algorithms for SNV based haplotype assembly Even though most *de novo* genome assemblers do not output haplotype resolved sequences, it is possible to reconstruct each haplotype using a post processing step. This process, called *haplotype assembly*, usually involves mapping your reads back to a reduced "haploid" reference genome, either an existing one or one created *de novo* just before, calling observed variants (SNVs); and then assign each allele to a chromosome by looking at the read data.

Haplotype assembly is computational hard problem: given m heterozygous SNVs, the number of possible haplotypes for a diploid organism is then 2^m. One of the most common formulations for haplotype assembly is the "Minimum Error Correction" (MEC) problem, and among other similar formulations is proven to be NP-hard [16, 17].

The accuracy of haplotype assembly methods are hampered by the short read lengths of the second generation sequencing platforms. To connect two SNV sites it is required to observe them together in a single read, and with read lengths of several hundred bases, this probability is quite low [11]. Therefore these short reads do not provide enough information to accurately resolve the haplotype. Here, the long reads produced by the third generation sequencing platforms help solve this issue, as the probability to observe multiple SNVs in a single read is much higher if the average read length is around ten thousand bases long. With Oxford Nanopore, reads up to one million bases have been observed. They do bring their own challenges, especially because of their high error rate of 6–15%.

Because the MEC problem is NP-hard, it will come as no surprise that all algorithms have exponential runtime or resort to heuristics. There is a heuristic approach based on calculating Max-Cuts [18]; an attempt to solve the problem using integer linear programming [19], but this latter approach has some troubles with "difficult blocks" and resorts to heuristics to solve these blocks. Some earlier "fixed-parameter tractable" (FPT) algorithms have a runtime that is exponential in the number of variants per read [20, 21], but this makes it impractical for long read data.

A few FPT algorithms designed with long read data in mind are *ProbHap* [22], *WhatsHap* [23] and *HapCol* [24]. These algorithms are more suitable for long read data because they are only exponential in the coverage of a variant rather than the number of variants per read. *ProbHap* tries to optimise a likelihood function, which is a more generalised version of the objective function of MEC, using a dynamic programming approach [22]. *WhatsHap* introduces an exact dynamic programming algorithm that solves the weighted version of MEC [23]. The weights represent the quality of the measured base in a read, according to the error model of the sequencing platform. *HapCol* is another exact dynamic programming approach that solves a "k-constrained" variant of the MEC problem, trying to exploit the fact that current long read technologies have a relative uniform distribution of errors [24].

The algorithms discussed so far all solve the haplotype assembly problem in the diploid case. But a lot of plants, fish or yeasts have polyploid or aneuploid genomes, with three, four or even ten copies per chromosome. The phasing problem becomes much harder in the polyploid case. If k is the ploidy of your organism and there are m heterozygous SNV sites, then the number of possible configurations is of the order of k^m. Additionally, in the diploid case knowing one haplotype immediately implies the other, but this does not hold for higher ploidy organisms [25].

There are no exact algorithms for the polyploid phasing problem, all resort to heuristics. Aguiar *et al.* proposed one of the first methods to resolve haplotypes for polyploid organisms, with an algorithm called *HapCompass* that is based on minimum spanning trees [26, 27]. *SDhaP* is an algorithm using semi-definite programming to minimise the MEC-score [28], and *HapTree* uses a probabilistic model to build the most likely haplotype configuration with a branch-and-bound like algorithm [29]. As pointed out by Berger *et al.*, the MEC-score makes less sense in polyploid case because different configurations can have the same MEC-score due to local similarities among chromosomes [29]. In a recent review by Motazedi *et al.*, *HapTree* is considered the most accurate method for ploidy ≤ 6, while also the most demanding in terms of computing resources. *HapCompass* is more accurate for higher ploidy, but none of the methods are accurate in the absolute sense for ploidy > 6 [25].

1.2.2. Haplotype-resolved genome assembly A disadvantage of SNV based phasing is that these methods are unable to deal with larger or structural variants [30]. Furthermore, most current *de novo* genome assemblers include a "bubble popping" step, in which bubbles in the assembly graph are flattened [31–34]. For haplotype assembly, this process is backwards: popping bubbles removes any variation among homologous chromosomes, which is later recollected by calling SNVs.

The problem of haplotype-resolved diploid or polyploid *de novo* genome assembly is not new, but until recently no universal solution for this problem existed [9, 30]. *Falcon-Unzip* is the first *de novo* genome assembler that uses long reads produced by the third generation sequencing technology to resolve haplotypes for diploid organisms [9]. For polyploid organisms no solution exists yet.

1.3. Research Question

In this work we aim to answer the following question:

How well can we do haplotype-resolved de novo genome assembly for polyploid organisms and output separate DNA sequences for each haplotype?

We divide our main question in the following subquestions:

- *How to capture variation among homologous chromosomes in an assembly graph?*
- *How to find the best set of paths through an assembly graph such that each path represents a haplotype?*
- *How to evaluate the output?*

To answer these questions we introduce PHASM (PHasing ASseMbler): a prototype long read *de novo* genome assembler for polyploid organisms that outputs separate DNA sequences for each chromosome homologue.

Early in the project, we tried to formulate the problem of finding paths through an assembly graph as a network flow problem. We did not continue down this path because the information you can put on the nodes and edges is static, which makes it difficult to change the likelihood of certain configurations based on decisions made earlier. Furthermore, a network flow solution results in flow values for every edge. This is not yet a set of paths through the assembly graph. To obtain a set of paths, a path decomposition step is required, which is a computational hard problem for a general directed graph. This is further explained in the supplemental material.

Instead, we have chosen a more pragmatic approach. An overview of the complete assembly pipeline can be seen in Figure 1. PHASM expects long reads from a whole genome sequencing experiment as input data (Figure 1a); it identifies pairwise local alignments between these reads (Figure 1b, Section 2.1); we build an assembly string graph using these pairwise local alignments (Figure 1c, Section 2.2); we identify superbubbles in the assembly graph and build bubble chains (Figure 1d, Section 2.3); which act as input for the phasing algorithm that searches for the best set of paths through each bubble chain such that each path represents a haplotype (Figure 1e, Section 2.4). The DNA sequence corresponding to these paths are then written to a FASTA file.

Phasing using an assembly graph has several advantages over SNV based phasing: it is possible to phase larger blocks at once because paths in an assembly graph can span multiple variants; and in an assembly graph it is easier to detect larger structural variants — for example, the Y-shaped fork in Figure 1c is a possible translocation.

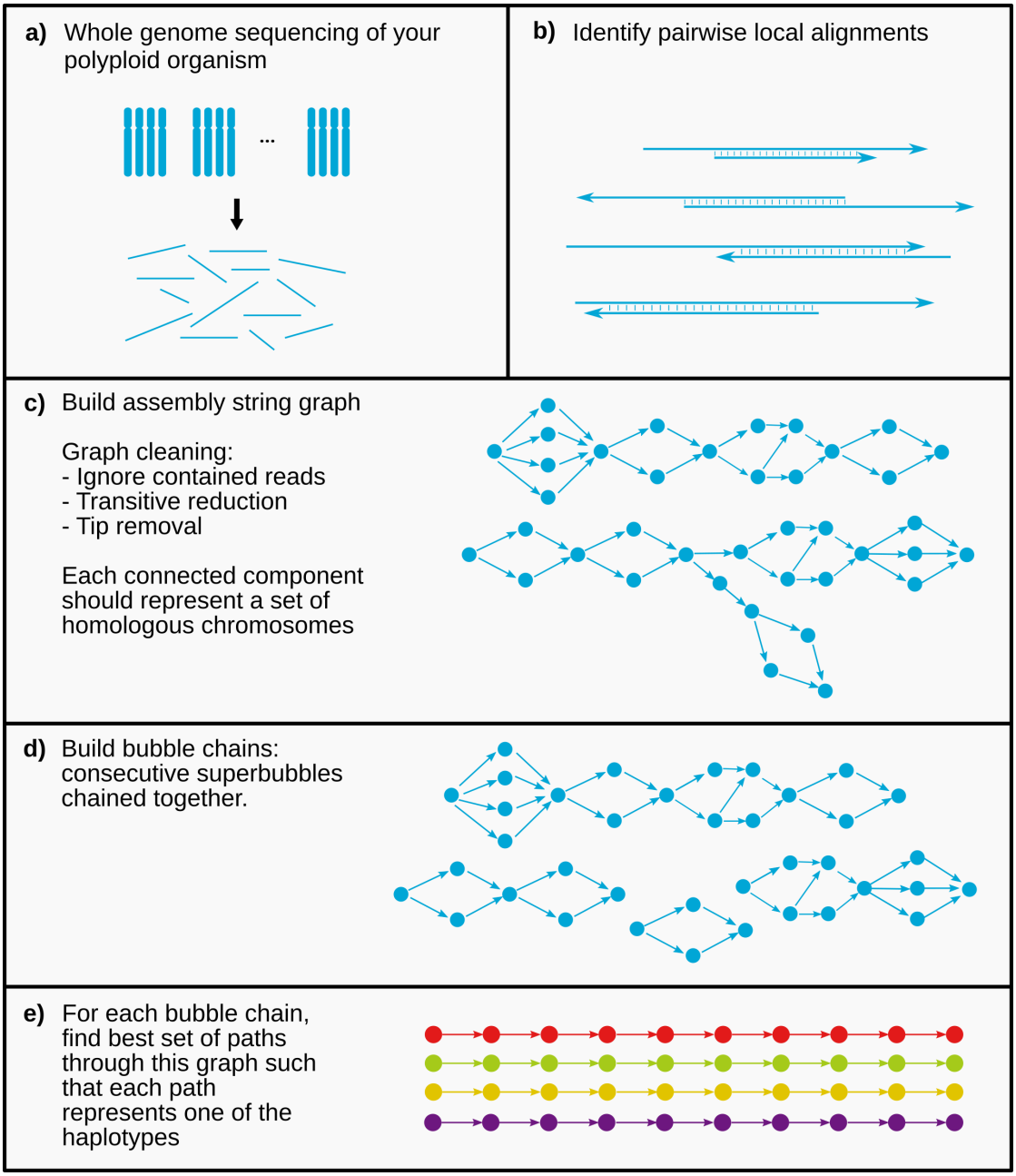


Figure 1 An overview of the complete PHASM assembly pipeline. **(a)** Perform whole genome sequencing on your polyploid organism and obtain a read dataset. **(b)** Find pairwise local alignments between reads. **(c)** Using pairwise local alignments, PHASM builds an assembly string graph, in similar fashion to miniasm [34]. Several methods are applied to clean the graph, but note that no bubbles or superbubbles are popped. The Y-shaped forks in an assembly graph can represent a translocation. **(d)** The next step consists of identifying superbubbles and building *bubble chains*: consecutive superbubbles chained together. These subgraphs are the input for the phasing algorithm. **(e)** This is the phasing step. For each bubble chain PHASM searches for the best set of paths through this subgraph such that each path represents a haplotype. We use read data to check which paths should be connected.

This work focusses on read data without sequencing error. This allows us to build a theoretical framework for phasing using an assembly graph, and later adapt the algorithms to deal with real data.

In Section 3 we present our results, which are discussed in Section 4. We reflect on the project and propose several ideas for future research efforts in Section 5.

2. Materials and Methods

2.1. General Definitions and Notation

In this section we introduce the definitions and notation of several general concepts. The notation is mostly borrowed from miniasm [34].

Definition 2.1. Let $\Sigma = \{A, C, G, T\}$ be the alphabet of DNA nucleotides. A DNA sequence is a sequence of characters $S = a_1a_2a_3, \ldots, a_n$ where $a_i \in \Sigma$. Its length is denoted as $|S| = n$. The Watson-Crick complement of a character a is denoted as $\overline{a}$. In a similar way the reverse complement of a sequence can be defined as $\overline{S} = \overline{a_1a_2a_3, \ldots, a_n} = \overline{a}_n\overline{a}_{n-1}, \ldots, \overline{a}_1$.

It is possible that two reads locally align. We define a local alignment between two reads with additional metadata as follows:

Definition 2.2. A local alignment between two DNA sequences S_a, S_b is denoted as the tuple $(S_a, S_b, as, ae, bs, be)$, where:

- S_a: the query read, use $\overline{S}_a$ to denote the reverse complement;
- S_b: the target read, use $\overline{S}_b$ to denote the reverse complement;
- as: starting position of the alignment on the query read;
- ab: end position of the alignment on the query read;
- bs: starting position of the alignment on the target read;
- be: end position of the alignment on the target read;

In some contexts we are only interested in the reads involved in a local alignment, in which we will refer to this local alignment as $(S_a, S_b, \ldots)$.

What these variables represent is shown more clearly in Figure 2. In this figure two reads with a local alignment are shown, and the shaded gray area marks the aligning region. This local alignment involves the reverse complement of the target read S_b, which is denoted with $\overline{S}_b$. Note that the aligning region does not necessarily involve the whole suffix of S_a or the whole prefix of S_b. If $ae \neq |S_a|$ or $bs \neq 0$, then the local alignment induces "overhang regions". If the overlap between the two reads would be perfect, then these overhang regions would not exist.

A read that can be fully aligned within an other read (apart from the optional overhang region) is said to be contained.

2.2. Assembly Graph Construction

No existing tool is available to find pairwise exact overlaps. Therefore we use DALIGNER to find pairwise local alignments between reads [35]. DALIGNER is used for identifying approximate local alignments, but our data is error free. By using very strict settings, we expect that the

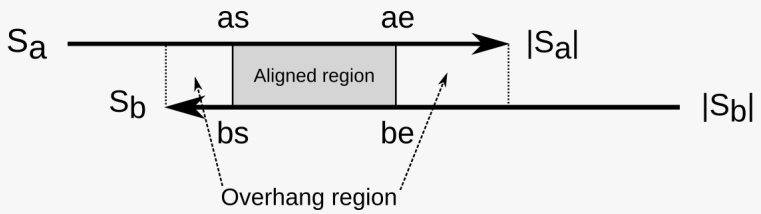


Figure 2 A local alignment between two reads S_a and S_b. In this figure read S_a has a local alignment with the reverse complement of a read S_b. This configuration would result in an edge $S_a \rightarrow \overline{S_b}$ with $l(S_a \rightarrow \overline{S_b}) = as - bs$, and an edge for its reverse $S_b \rightarrow \overline{S_a}$, with $l(S_b \rightarrow \overline{S_b}) = (|S_b| - be) - (|S_a| - ae)$.

approximate alignments have little influence and that mostly real overlaps are reported. We report the settings used with DALIGNER in the supplemental material. We will come back to this point in Section 3, when discussing our results.

Using this database of local alignments we construct an assembly graph similar to miniasm [34]. In summary, an assembly graph is a general directed graph $G = (V, E)$, without any multi-edges, where vertices represent DNA sequences and the edges denote overlaps. For each read, we add two vertices to the assembly graph: one for the forward strand, and one for its reverse complement. If a suffix of read S_a matches a prefix of S_b (or any of the reverse complements), we create an edge between the two corresponding vertices. If there is a local alignment between two reads S_a and S_b, the double stranded nature of the DNA molecule implies another local alignment between the reverse complements. Therefore, when creating an edge $S_a \rightarrow S_b$ between two reads S_a and S_b, another edge $\overline{S_b} \rightarrow \overline{S_a}$ is created.

Each edge is labelled with a length, which is defined as the length of *unmatched prefix* of S_a. The length of an edge will be denoted as the function $l(u \rightarrow v)$, $l : E \rightarrow \mathbb{N}$. An example is shown in Figure 2.

Contained reads are ignored, we perform transitive reduction [36], and remove short incoming and outgoing tips. A graphical overview of the graph cleaning steps is shown in Figure 3. As a final step, any linear unambiguous (non-branching) path is merged to a single vertex.

The sequence corresponding to a path in the assembly graph can be obtained by concatenating the unmatched prefixes at each edge in the path.

2.3. Build bubble chains instead of long linear unambiguous paths

Sequencing error or heterozygousity between chromosome homologues can result in branches in the assembly graph. As a result, a common motif in an assembly graph is a bubble or superbubble (Figure 4) [9, 31, 34]. To obtain linear unambiguous paths as long as possible, most current *de novo* genome assemblers pop these bubbles by only keeping the path best supported by read data [31–34].

Popping bubbles, however, removes information on variation between chromosome homologues from your assembly graph. We therefore propose a novel strategy: instead of building long linear unambiguous paths we construct acyclic subgraphs where superbubbles are included as a whole. By chaining consecutive superbubbles we construct an acyclic subgraph called a *bubble chain*.

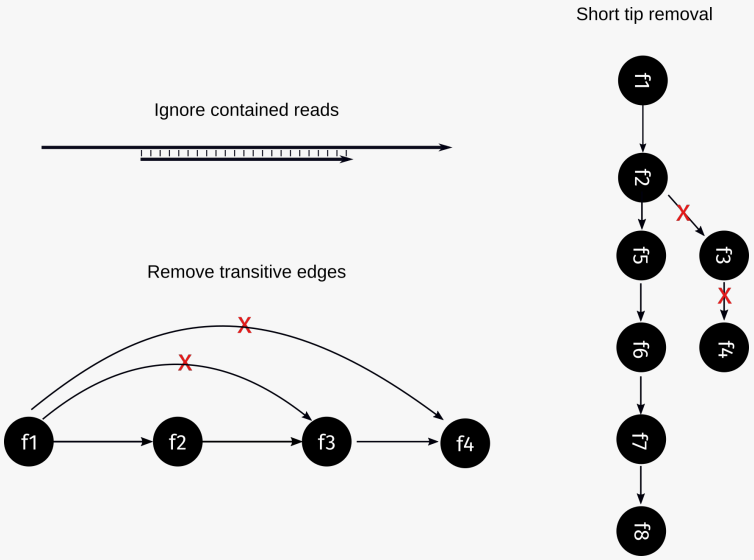


Figure 3 Several graph cleaning methods graphically explained. Reads that fully align within another read are ignored, and therefore not added to the assembly graph. Overlaps between two reads that can be inferred through other reads result in transitive edges, which can be removed without affecting the connectivity of the graph. Any short tips, which are often induced by alignment artifacts or missing alignments, are cut from the graph.

2.3.1. Identifying superbubbles We formally define the concept of a superbubble using the original definition from Onodera *et al.* [37]:

Definition 2.3 (Onodera *et al.* [37])**.** Let $G = (V, E)$ be a directed graph, and let $s, t \in V$ be two distinct vertices. Then, s and t are the entrance and exit of a superbubble $\langle s, t \rangle$ if the following conditions are met:

- **reachability:** t is reachable from s;
- **matching:** the set of vertices reachable from s without passing through t is equal to the set of vertices that can reach t without passing through s;
- **acyclicity:** if U is the set of vertices satisfying the matching criterion, then the subgraph induced by U is acyclic;
- **minimality:** no vertex in U other than t forms a pair with s that satisfies the conditions above.

The set of vertices $I = U \setminus \{s, t\}$ is called the interior of the superbubble $\langle s, t \rangle$. As can be seen in Figure 4, superbubbles can also be nested.

Superbubble identification in a general directed graph can be done in $O(|V| + |E|)$ time by combining the graph partitioning algorithm from Wing-Kin Sung *et al.* [38] with the superbubble identification algorithm in a directed acyclic graph by Brankovic *et al.* [39]. Although a *superbubble* is strictly speaking a generalisation of a *bubble*, we will use these two terms interchangeably throughout this work.

2.3.2. Building bubble chains Our strategy is to build bubble chains: consecutive superbubbles chained together. Consecutive superbubbles are defined as follows:

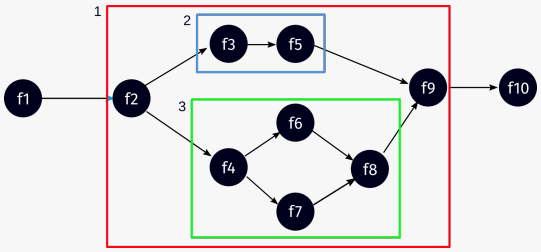


Figure 4 A few example superbubbles. $\langle f2, f9\rangle$ is a superbubble and it contains a few nested super bubbles: $\langle f3, f5\rangle$, $\langle f4, f8\rangle$. Traditionally *de novo* genome assemblers pop these kind of structures and only keep the path that is best supported by read data.

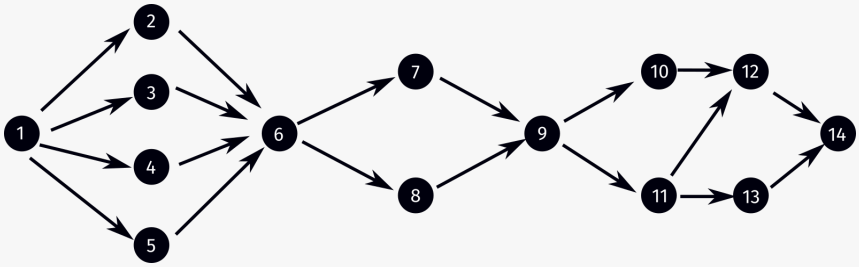


Figure 5 A bubble chain. This example contains three consecutive superbubbles: $\langle 1, 6\rangle, \langle 6, 9\rangle, \langle 9, 14\rangle$.

Definition 2.4. Let $G = (V, E)$ be a directed graph, and let $\langle s_1, t_1\rangle$, $\langle s_2, t_2\rangle$ be two distinct non-nested superbubbles of G. $\langle s_1, t_1\rangle$ and $\langle s_2, t_2\rangle$ are consecutive if $t_1 = s_2$ or there exists a single linear non-branching path between t_1 and s_2 in G.

Recall that unambiguous linear paths in our assembly graph are merged to a single vertex, which gives rise to the following lemma:

Lemma 2.1. *Let $G = (V, E)$ be a directed graph with all linear unambiguous paths merged to a single vertex, and let $\langle s_1, t_1\rangle$, $\langle s_2, t_2\rangle$ be any two non-nested consecutive superbubbles of G. Then $t_1 = s_2$.*

Proof by contradiction. Assume $t_1 \neq s_2$. Then by definition there should be a linear unambiguous path between t_1 and s_2, which should have been merged. A contradiction. □

We formally define a bubble chain as follows:

Definition 2.5. Let $G = (V, E)$ be a weakly connected directed *acyclic* graph and B its set of non-nested superbubbles in topological order. G is called a *bubble chain* if it satisfies the following condition:

$$b_i, b_{i+1} \text{ are consecutive} \qquad b_i, b_{i+1} \in B, \quad 1 \leq i < |B| \tag{1}$$

An example of a bubble chain can be seen in Figure 5. Lemma 2.1 also holds for a bubble chain.

The algorithm to build bubble chains is shown in Algorithm 1. It expects that all linear unambiguous paths in the given assembly graph are merged to a single vertex. The algorithm starts by identifying all non-nested superbubbles in the assembly graph. We build a list with potential start points, in which we give priority to superbubble entrances with no incoming edges. Other potential start points are superbubble entrances that are not an exit of an other superbubble.

Bubble chains are subgraphs of the original assembly graph, so for each start point we initialise the set of vertices V' for this bubble chain to the empty set. For each additional consecutive superbubble we add its vertices $U_{\langle s,t \rangle}$ to V'. Recall that $U_{\langle s,t \rangle}$ is the set of vertices that satisfy the matching criterion (see Definition 2.3). Moving to the next superbubble is as easy as setting the current bubble exit as next superbubble entrance, due to Lemma 2.1. We stop if we encounter a vertex that is not a superbubble entrance, or if we encounter a bubble that we already have visited. If V' is not the empty set, we report the subgraph for this bubble chain.

Algorithm 1 Reports bubble chains in an assembly graph with any linear unambiguous paths merged to a single vertex.

function BUILDBUBBLECHAINS($G = (V, E)$)
  $B \leftarrow$ FINDSUPERBUBBLES(G, nested=FALSE)
  $S \leftarrow \{s : \langle s, t \rangle \in B\}$ ▷ Bubble entrances
  $T \leftarrow \{t : \langle s, t \rangle \in B\}$ ▷ Bubble exits
  $visited \leftarrow \varnothing$
  ▷ Build list of potential start points, priority to vertices without incoming edges
  $start\_points \leftarrow [s \in S : \delta^+(s) = 0]$ ▷ Bubble entrances with no incoming edges
  $start\_points \leftarrow start\_points + [s \in S : s \notin T]$ ▷ Bubble entrances that are not the exit of an other bubble
  for all $s \in start\_points$ **do**
    $V' \leftarrow \varnothing$
    while $s \in S$ **do**
      $t \leftarrow \text{exit}(s)$
      if $s \in visited \wedge t \in visited$ **then**
        break
      $V' \leftarrow V' \cup U_{\langle s,t \rangle}$
      $visited \leftarrow visited \cup U_{\langle s,t \rangle}$
      $s \leftarrow t$ ▷ Move to the next bubble
    if $V \neq \varnothing$ **then**
      $E' \leftarrow \{(u, v) : (u, v) \in E, u \in V', v \in V'\}$
      report $G' = (V', E')$

2.4. Build haplotypes bubble by bubble

A bubble chain is a representation for the DNA sequence of a set of homologous chromosomes, with any variation among these chromosomes included. But, the exact DNA sequence of each chromosome, its haplotype, is still unknown. The problem is now to find the best set of paths through a bubble chain where each path represents a haplotype. In other words, we need to connect paths in one superbubble to paths in another superbubble. A graphical overview of the algorithm is shown in Figure 6. We build a tree of possible solutions (Figure 6a), by building each candidate incrementally. We use read data to assess the likelihood of each candidate (Figure 6b), and prune unlikely configurations early.

Currently, the algorithm assumes that the number of homologous chromosomes, or the ploidy level k, is known beforehand. Ploidy level can be estimated by analysing coverage [40, 41], but this has not been integrated in the genome assembly pipeline yet. Furthermore, the algorithm assumes that for each haplotype, at least one path through each superbubble spells a sequence

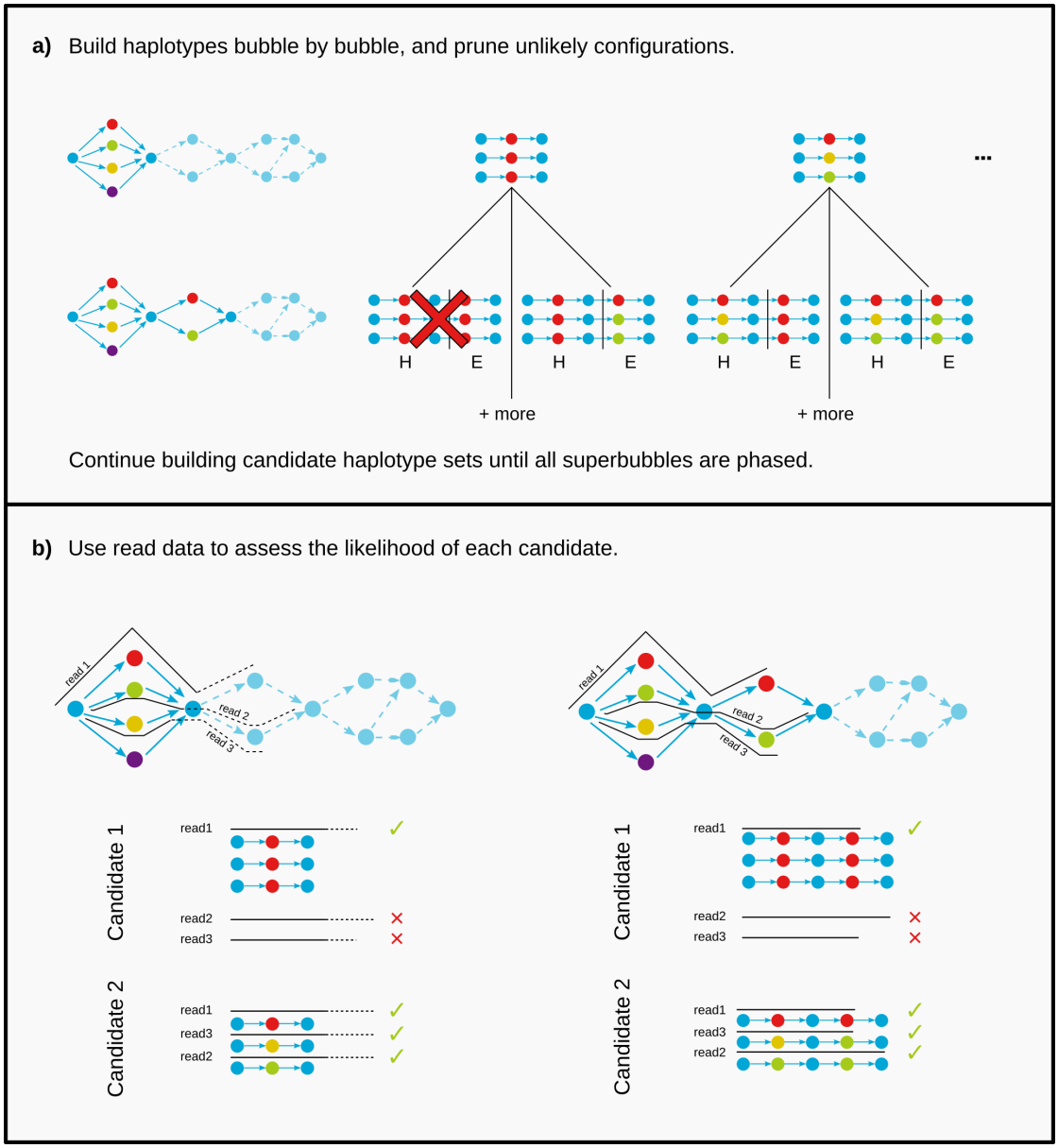


Figure 6 Overview of the phasing algorithm. The examples in this figure assume a ploidy $k = 3$. The coloured vertices denote variation observed at a single bubble, not its original chromosome. **(a)** The haplotypes are built bubble by bubble. At each bubble, we enumerate all possible k-tuples of paths through this bubble, and build a tree of possible configurations. We prune unlikely configurations early, to keep the problem tractable. **(b)** An illustration how to assess the likelihood of a candidate haplotype set. We look at how well other reads align to each candidate, and look for a haplotype set that best explain the read data. At each bubble, we ignore any read information downstream of this bubble.

that is consistent with this haplotype. The goal is to link paths in consecutive superbubbles.

The algorithm builds a k-ploidy phase, which is defined as follows:

Definition 2.6. A k-ploidy phase of a bubble chain $G = (V, E)$ is a tuple of k (not necessarily distinct) paths through G. The paths through G are represented by the vectors of vertices $(h_1, \ldots, h_k)$ which satisfy the following properties:

$$\begin{aligned} h_i[j] &\in V \qquad 1 \leq i \leq k, \quad 1 \leq j \leq |h_i| \\ (h_i[j], h_i[j+1]) &\in E \qquad 1 \leq i \leq k, \quad 1 \leq j < |h_i| \end{aligned}$$

Here $|h_i|$ denotes the number of elements (vertices) in a vector h_i. Note that the vectors do not necessarily have to be of equal length. Another term used in this document for any such tuple of k paths is a haplotype set H_{set}.

Finding the best set of k paths through a bubble chain is a hard problem because the number of paths through the graph grows exponentially with each added vertex, plus there are an exponential number of different k-tuples of these paths. Although our problem is more constrained, we are looking for k paths from the beginning of the bubble chain to the end of the bubble chain, we still deal with an exponential number of configurations. If we assume that the number of paths through a single superbubble (from entrance to exit) is of the order $O(k)$ (the number of haplotypes), then the number of different k-tuples of paths through this single superbubble is $O(k^k)$. If there are n non-nested superbubbles in the bubble chain, then the total number of phases is $O((k^k)^n) = O(k^{kn})$. As a final note, the order of each haplotype within a haplotype set does not matter, so a final estimate for the number of distinct phases is $O(\frac{1}{k!}k^{kn})$.

In the rest of the section we introduce a heuristic algorithm that builds the most likely configuration bubble by bubble. For each likely configuration built using the first m bubbles, we search for the most likely extension at bubble $m+1$. The algorithm and likelihood model are inspired by HapTree [29], but adapted to work on bubble chains instead of a list of SNV positions. First, we will explain the likelihood model on a global level, then we adapt this likelihood model to the local situation of a single superbubble, and then describe the algorithm to maximise the global likelihood.

2.4.1. Likelihood of a k-ploidy phase The basis of *de novo* genome assembly is the set of sequenced reads. The reads included as vertex in an assembly graph are a subset of the complete read set. Nevertheless, reads not included as vertex in the assembly graph can provide information for phasing. These reads possibly align to other reads which are included as vertex in the assembly graph, and therefore possibly connect two paths in different superbubbles. Throughout the rest of this work we will use the term *graph reads* to denote reads included as vertex in the assembly graph.

A bubble chain is a subgraph of the complete assembly graph, and only a subset of reads $r \in \mathcal{R}$ are relevant when phasing this bubble chain. We therefore collect the set of relevant reads R for a bubble chain as follows:

Definition 2.7. Let $G = (V, E)$ be a bubble chain, $\mathcal{R}$ be the set of *all* reads from the sequencing experiment, and let $\mathcal{A}$ be the set of all pairwise local alignments between all reads. The set of relevant reads $R \subseteq \mathcal{R}$ for this bubble chain is then:

$$R = \{r \in \mathcal{R} \mid \exists S_b \in V, (r, S_b, \ldots) \in \mathcal{A}\}$$

In other words, the set of relevant reads for this bubble chain contains all reads included as vertex in the bubble chain and all reads that align to these graph reads.

The goal is to find the most likely haplotype set given our relevant reads and corresponding alignments. The haplotype reconstruction problem for a bubble chain can be formulated as finding the most likely set of paths through the bubble chain, given the relevant reads. Using Bayes' theorem we formulate the likelihood of a haplotype set as follows:

$$P[H_{set}|R] = \frac{P[R|H_{set}]P[H_{set}]}{P[R]} \tag{2}$$

We are comparing haplotype sets built using the same relevant read set R so $P[R]$ will be the same for all haplotype sets. We therefore define the *relative likelihood* of a haplotype set as follows, similar to HapTree [29]:

$$RL[H_{set}|R] = P[R|H_{set}]P[H_{set}] \tag{3}$$

The prior $P[H_{set}]$ is assumed equal for all haplotype sets, except for haplotype sets with duplicate paths. This is to take into account that two different haplotype sets with the same paths but in a different order can generate the same relevant read set. Let $M = \{m_1, m_2, \ldots\}$ be the set of multiplicities for each path in a given haplotype set, then the number of distinct orderings of this haplotype set is the multinomial coefficient $\binom{k}{m_1,m_2,\ldots} = \frac{k!}{m_1!m_2!\ldots}$. The total number of possible haplotype sets for a bubble chain is $\prod_{b\in B} p(b)^k$, where B is the set of non-nested superbubbles in the bubble chain, and $p(b)$ denotes the number of different paths from the entrance to exit for a given superbubble b. Combining the two concepts above we define the prior $P[H_{set}]$ as follows:

$$P[H_{set}] = \frac{\binom{k}{m_1,m_2,\ldots}}{\prod_{b\in B} p(b)^k} \tag{4}$$

Consider the example below which shows two haplotype sets with paths through the bubble chain shown in Figure 5, with $k = 3$. Furthermore, there are $4^3 \cdot 2^3 \cdot 3^3 = 13824$ possible haplotype sets when $k = 3$ for the bubble chain in Figure 5.

$$\begin{aligned} H_{set,1} = [&[1,2,6,7,9,10,12,14],\\ &[1,2,6,7,9,10,12,14],\\ &[1,3,6,8,9,11,13,14]]\\ H_{set,2} = [&[1,2,6,7,9,10,12,14],\\ &[1,3,6,7,9,11,12,14],\\ &[1,5,6,8,9,11,13,14]] \end{aligned}$$

Haplotype set one has one duplicate path, so its set of multiplicities is $M = \{2,1\}$. Therefore the prior for haplotype set one $P[H_{set,1}] = \frac{\binom{3}{2,1}}{13824} = \frac{3}{13824}$. Similarly, haplotype set two has three distinct paths so its set of multiplicities is $M = \{1,1,1\}$. Its prior then becomes $P[H_{set,2}] = \frac{\binom{3}{1,1,1}}{13824} = \frac{6}{13824}$.

Next, we turn to the probability $P[R|H_{set}]$. We assume that each read is sequenced independently from the genome, so we define the probability of the total relevant read set as the sum of each individual probability of a relevant read, divided by the total number of relevant reads.

$$P[R|H_{set}] = \frac{\sum_{r \in R} P[r|H_{set}]}{|R|} \tag{5}$$

To calculate the probability of a single read given a haplotype set, we look at its alignments to reads part of the paths representing each haplotype. Recall that a local alignment between two reads is denoted by the tuple $(S_a, S_b, as, ae, bs, be)$ (see Definition 2.2 and Figure 2). We can now determine the total overlap length of a read r with a single haplotype h, by looking at the minimum value of as of all alignments of r with reads part of h, and the maximum value of ae of all alignments of r with reads part of h. This is shown more graphically in Figure 7. Mathematically put:

$$\begin{aligned} A(r,h) &= \{(S_a, S_b, as, ae, bs, be) \in \mathcal{A} \mid S_a = r, S_b \in h\} \\ O_{start}(r,h) &= \min_{(S_a,S_b,as,ae,\ldots) \in A(r,h)} as \\ O_{end}(r,h) &= \max_{(S_a,S_b,as,ae,\ldots) \in A(r,h)} ae \\ O(r,h) &= O_{end}(r,h) - O_{start}(r,h) \end{aligned} \tag{6}$$

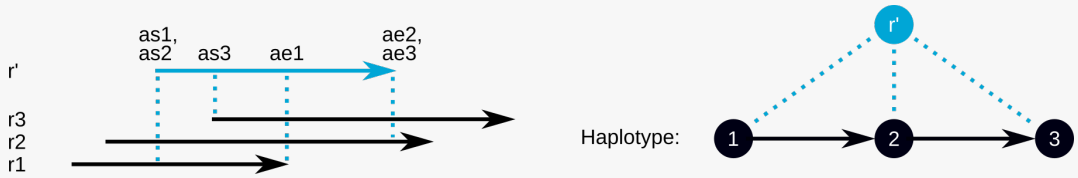
In the above equations $\mathcal{A}$ is the set of all pairwise local alignments between all reads, $A(r,h)$ is a subset of these pairwise local alignments involving only our target read r and a haplotype path h, and $O(r,h)$ denotes the total overlap length of a read r with a single haplotype h. If we divide this total overlap length by the length of the read r we obtain a probability that this read could come from this haplotype.

A haplotype set contains multiple haplotypes, and a read could have come from any of these haplotypes. We assume that each haplotype is equally likely, and end up with the following formulation for $P[r|H_{set}]$:

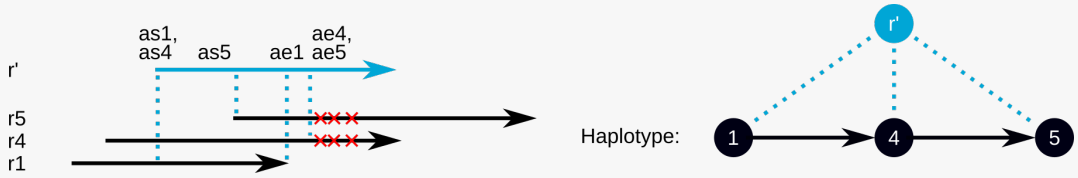
$$P[r|H_{set}] = \frac{1}{k} \sum_{h \in H_{set}} \frac{O(r,h)}{|r|} \tag{7}$$

Where $|r|$ means the length of the read. The goal is now to find a haplotype set that maximises the product $P[R|H_{set}]P[H_{set}]$, or the relative likelihood $RL[H_{set}|R]$.

2.4.2. Likelihood of an extension Reads are in general much shorter than the full chromosome, and therefore are only informative in regions where this read aligns well. This insight provides the foundation of our algorithm: we build a haplotype set bubble by bubble. Let H be the (partial) "true" haplotype set built using only the first m superbubbles, then at superbubble $m+1$ we generate all candidate extensions E: a tuple of k paths from the entrance to the exit of superbubble $m+1$. The goal is now to find the most likely extension E given our "true" haplotype set H. To determine the likelihood of an extension we only have to look at reads that both align to any of the graph reads of the first m superbubbles and to any of the graph reads of superbubble $m+1$.



(a) A read that fully overlaps with a given haplotype



(b) A read that partially overlaps with a given haplotype. Note that the red crosses mark mismatches with the read r', not in the algignment between $r4$ and $r5$.

Figure 7 A graphical explanation on how to determine the total overlap length of a read r' with a certain haplotype, built from other reads. Note that in both cases, read r' has three alignments with the haplotype, but the overlap with the haplotype in case **(a)** is larger than the overlap in case **(b)**.

Let $b_i = \langle s, t \rangle$ be the current superbubble for which we will generate new extensions E. To decide which extension is the most likely, we adapt our global likelihood model to the local situation of a single superbubble. We define the probability of an extension E given a (partial) "true" haplotype set H, built using superbubbles prior to b_i ($b_1, \ldots, b_{i-1}$), and relevant reads R as follows, applying the rule for conditional probability:

$$\begin{aligned} P[E|H,R] &= \frac{P[EHR]}{P[HR]} \\ &= \frac{P[R|H,E] \cdot P[HE]}{P[R|H] \cdot P[H]} \\ &= \frac{P[R|H,E] \cdot P[H|E] \cdot P[E]}{P[R|H] \cdot P[H]} \end{aligned} \tag{8}$$

When comparing different extensions, the "true" haplotype set H is the same for all extensions and each extension is built using the same relevant read set R. Thus, the factors $P[R|H]$ and $P[H]$ will be the same for all extensions. Furthermore, we assume that the probability of observing H is independent of the extension E, such that the probability $P[H|E]$ is also the same for each extension. Removing these factors from the equation, we define the relative likelihood for an extension as follows:

$$RL[E|H,R] = P[R|H,E]P[E] \tag{9}$$

As discussed earlier, only a subset of reads provide information on what paths through superbubbles $b_1, \ldots, b_{i-1}$ should be connected with other paths in superbubble b_i. We only need to look at reads that align to both b_i and any of the earlier superbubbles. Let us introduce the concept of *spanning reads* more formally:

Definition 2.8. Let H be the "true" (partial) haplotype set built using the first $i-1$ superbubbles, and $b_i = \langle s, t \rangle$ be the superbubble from which we will generate an extension E. Then, the we define the *spanning reads* $SR(b_i)$ of a superbubble b_i as the set of reads that have a local alignment with one of the graph reads in one of the superbubbles prior to b_i (i.e. any superbubble $b_j,\ j < i$), and have a local alignment with one of the graph reads in superbubble b_i. Mathematically put:

$$
\begin{aligned}
A(r, b) &= \{(S_a, S_t, \ldots) \in \mathcal{A} \mid S_a = r, S_t \in U_b\} \\
SR(b_i) &= \left\{ r \in R \mid \exists S_b : (r, S_b, \ldots) \in \bigcup_{j=1}^{i-1} A(r, b_j) \wedge \exists S_c : (r, S_c, \ldots) \in A(r, b_i) \right\}
\end{aligned} \tag{10}
$$

Here $A(r, b)$ represents the subset of all pairwise local alignments that involve a read r and one of the graph reads from superbubble b. The graph reads of superbubble b are denoted as U_b (the set of vertices that satisfy the matching criterion, see Definition 2.3). Although not explicitly stated, we ignore any graph read that is a superbubble entrance or exit, as they are shared across all haplotypes and therefore provide no information on which haplotype set is correct.

The relative likelihood for an extension E at current superbubble b_i then becomes:

$$
RL[E|H, SR(b_i)] = P[SR(b_i)|H, E]P[E] \tag{11}
$$

The prior $P[E]$ is defined in a similar way to the prior in the global model (Equation 4). Let $M = \{m_1, m_2, \ldots\}$ be the multiplicities of each path in the extension. Then $P[E]$ can be calculated using the multinomial coefficients:

$$
P[E] = \frac{\binom{k}{m_1, m_2, \ldots}}{p^k} \tag{12}
$$

Where p is the total number of simple paths through superbubble b, and k is the ploidy.

To calculate $P[SR(b_i)|H, E]$ we build the extended haplotype set $H' = H + E$, and check how well each read $r \in SR(b_i)$ overlaps with each of the haplotypes (see Figure 7). There is, however, an important difference compared to the global model. To ensure fair comparisons we exclude any information downstream of the current superbubble b_i. This also ensures that our assumption of observing H is independent of an extension E holds.

By only looking at the spanning reads, we already ignore reads that solely align downstream of the current superbubble. But, some spanning reads can be long enough to extend beyond the current superbubble. We therefore search for an index $o_{r,max} \leq |r|$ such that the substring $r[1, o_{r,max}]$ does not play a role in the next superbubble. This index can be found by looking at local alignments of a read r with any of the predecessors of the current superbubble exit t. A graphical explanation can be seen in Figure 8. In this figure an example bubble chain is shown, and we are currently generating extensions at the superbubble $\langle 6, 9 \rangle$. Therefore any information beyond vertex 9 should be ignored. In other words, we do not know what comes after superbubble exit 9, we do not know how large any following overlaps with vertex 9 are, so to be sure we do not include any information beyond the current superbubble we search for an $o_{r,max}$ such that our read r does not overlap with the superbubble exit 9.

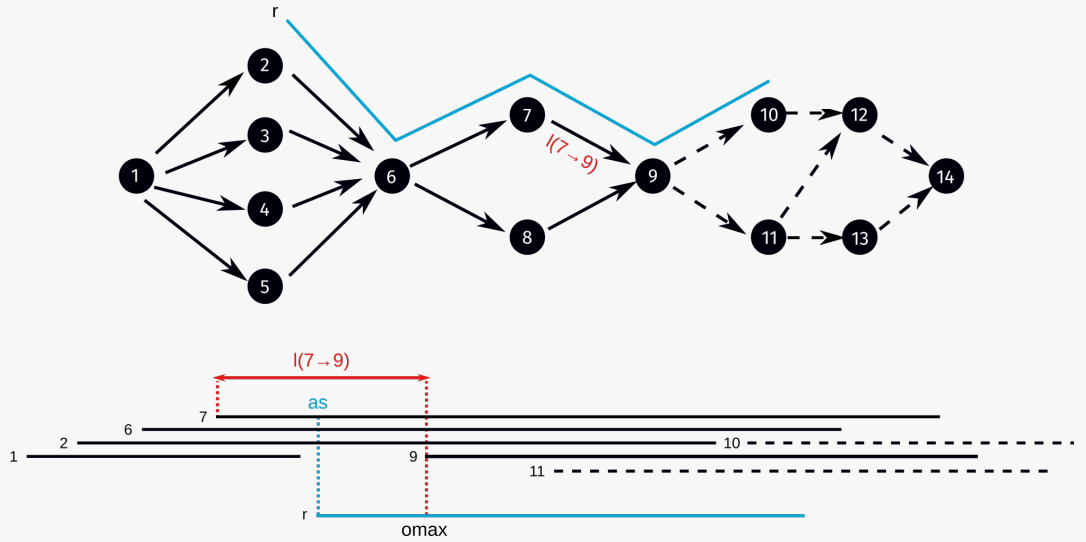


Figure 8 An example setting where some information of read r should be ignored because it extends beyond the current superbubble $\langle 6, 9\rangle$. To ensure we do not use any information from the next superbubble $\langle 9, 14\rangle$ (shown with dashed lines), we determine an index $o_{r,max}$ such that the substring $r[1, o_{r,max}]$ does not overlap with the bubble exit 9. Recall that the length of each edge is the length of the *unmatched prefix* of the corresponding local alignment, so we can quickly calculate $o_{r,max}$ by looking at the local alignment of r with any of the predecessors of superbubble exit 9.

Figure 8 also shows how to calculate $o_{r,max}$: we check for local alignments between read r and any of the predecessors of the superbubble exit vertex. Using the edge length and the start of the aligning region, we can calculate the index $o_{r,max}$. To conclude, at each superbubble $b_i = \langle s, t\rangle$ we calculate each $o_{r,max}$ as follows:

$$o_{r,max} = \begin{cases} \max\limits_{p \in \text{pred}(t) | (r,p,as,ae,\ldots) \in \mathcal{A}} l(p \to t) - as & \text{if } r \text{ has a local alignment with at least one of the predecessors of } t \\ |r| & \text{otherwise} \end{cases} \tag{13}$$

In the above equation $\text{pred}(t)$ is used to denote the set of predecessor vertices of t, $\mathcal{A}$ is the set of all pairwise local alignments, and $l(p \to t)$ is the edge length function. If a read has no local alignment with any of the predecessor vertices, we use its read length as $o_{r,max}$.

We now arrive at the formulation for $P[SR(b_i)|H, E]$:

$$P[SR(b_i)|H, E] = \frac{\sum_{r \in SR(b_i)} P[r|H, E]}{|SR(b_i)|} \tag{14}$$

$$P[r|H, E] = \frac{1}{k} \sum_{i=1}^{k} \frac{\min(O(r, h_i + e_i), o_{r,max})}{o_{r,max}} \tag{15}$$

The goal is now to find an extension E at the current superbubble b_i that maximises the relative

likelihood $RL[E|H, SR(b_i)] = P[SR(b_i)|H, E]P[E]$.

Before describing the full algorithm, we introduce one additional concept termed subreads:

Definition 2.9. Let $b_i = \langle s, t \rangle$ be a superbubble, and R be the set of relevant reads for the current bubble chain. We define the **subreads** $R'(b_i)$ as the set of reads relevant up to the current superbubble and not extending beyond the current superbubble b_i.

$$V' = \bigcup_{j=1}^{i} U_{b_j}$$
$$R'(b_i) = \{r[1, o_{r,max}] : r \in R \mid \exists S_b \in V', (r, S_b, \ldots) \in \mathcal{A}\} \tag{16}$$

Here V' is the set of vertices (reads) of all superbubbles up to the current superbubble b_i. $R'(b_i)$ then contains all these graph reads, plus any other read that aligns to one of the reads part of the graph, while making sure it does not extend beyond the current superbubble b_i.

2.4.3. Algorithm description In this section we present a dynamic programming-like algorithm, with some heuristic methods applied to prune the solution space. It is inspired by the algorithm used in HapTree [29], but adapted to work on graphs.

The general flow of the algorithm is as follows: (1) for each candidate "true" haplotype set H built using superbubbles $b_1, \ldots b_{i-1}$, generate all possible extensions at superbubble b_i, consisting of different k-tuples of paths through superbubble b_i; (2) calculate the relative likelihood $RL[E|H, R]$ for each generated extension, and discard any extension with a relative likelihood lower than an absolute threshold τ; (3) prune the solution tree even more by discarding any leaf that has a likelihood sufficiently lower than the most likely candidate; (4) move to the next superbubble and continue building solution tree. The algorithm will be explained in more detail below.

Extension. Given a candidate haplotype set H and the current superbubble b_i, the EXTEND algorithm generates all possible extensions E. First, all simple paths from superbubble entrance s to superbubble exit t are identified, from which all possible k-tuples are enumerated. For each extension the relative likelihood $RL[E|H, R]$ is calculated. If the relative likelihood is above a threshold τ, the extended haplotype $H' = H + E$ is added to a list $\mathcal{E}$, which is returned at the end of the algorithm.

The first superbubble is a special case: the order of haplotypes within a haplotype set does not matter, so instead of all k-tuples we generate all combinations with replacement of paths through the superbubble. Furthermore, the first bubble has no spanning reads by definition. Therefore we check all reads that align to this superbubble, while still making sure that we do not use information beyond the current superbubble.

It can happen that the distance between two superbubbles (in bases) is large, with the consequence that no reads are long enough to span between paths in the two consecutive superbubbles. When this happens, we have no information available for phasing. Therefore we start a new "haploblock" and act like the current superbubble is the first superbubble in a bubble chain, and then proceed as usual. This is not explicitly stated in the algorithm descriptions below.

The full algorithm can be seen in Algorithm 2.

Branching. At every superbubble b_i, we have a set of candidate "true" haplotype sets $\mathcal{H}$ built using the previous superbubbles $b_1, \ldots b_{i-1}$. The BRANCH algorithm extends every candidate

Algorithm 2 Algorithm to generate all possible extensions for a given (partial) haplotype set H at superbubble b_i.

function EXTEND($H, b_i = \langle s, t \rangle, \tau$)
  $\mathcal{E} \leftarrow [\,]$
  $\mathcal{P} \leftarrow$ ALLSIMPLEPATHS(s, t)
  if $i > 1$ **then**
    $ES \leftarrow$ ALLKTUPLES$(k, \mathcal{P})$
  else
    $ES \leftarrow$ COMBINATIONSWITHREPLACEMENT$(k, \mathcal{P})$
  for all $E \in ES$ **do**
    if $RL[E|H, SR(b_i)] > \tau$ **then**
      $H' = H + E$
      $\mathcal{E} \leftarrow \mathcal{E} + H'$
  return $\mathcal{E}$

haplotype set $H \in \mathcal{H}$, and returns a new set $\mathcal{H}'$ of candidate haplotype sets extended with paths through superbubble b_i and to be used at the next superbubble. This algorithm can be seen in Algorithm 3.

Algorithm 3 Generate extensions for all current candidate haplotype sets.

function BRANCH($\mathcal{H}, b_i = \langle s, t \rangle, \tau$)
  $\mathcal{H}' \leftarrow [\,]$
  for all $H \in \mathcal{H}$ **do**
    $\mathcal{E} \leftarrow$ EXTEND(H, b_i, τ)
    for all $H' \in \mathcal{E}$ **do**
      $\mathcal{H}' \leftarrow \mathcal{H}' + H'$
  return $\mathcal{H}'$

Pruning. To keep the problem a bit more tractable, we prune the solution tree after each branch operation. Given a set of candidate haplotype sets $\mathcal{H}$ PRUNE returns a subset $\mathcal{H}' \subseteq \mathcal{H}$ containing only sufficiently likely haplotype sets. All candidates with a relative likelihood score lower than a given fraction κ of the most likely candidate is removed. In other words, if $\omega = \max_{H \in \mathcal{H}} RL[H|R'(b_i)]$, then a haplotype set $H \in \mathcal{H}$ is added to $\mathcal{H}'$ if $RL[H|R'(b_i)] \geq \kappa\omega$. Here $0 \leq \kappa \leq 1$, and $R'(b_i)$ are the subreads defined earlier in Definition 2.9. The complete algorithm can be seen in Algorithm 4.

In this algorithm, it is not needed to calculate the relative likelihood again, because it is possible to store the value calculated during EXTEND. The runtime of this algorithm is therefore linear in the size of $\mathcal{H}$.

Algorithm 4 An algorithm to prune a set of candidate haplotype sets $\mathcal{H}$ and only keep sufficiently likely candidates.

function PRUNE($\mathcal{H}, b_i = \langle s, t \rangle, \kappa$)
  $\mathcal{H}' \leftarrow [\,]$
  $\omega \leftarrow \max_{H \in \mathcal{H}} RL[H|R'(b_i)]$
  for all $H \in \mathcal{H}$ **do**
    if $RL[H|R'(b_i)] \geq \kappa\omega$ **then**
      $\mathcal{H}' \leftarrow \mathcal{H}' + H$
  return $\mathcal{H}'$

When the number of candidates is very high, our implementation performs multiple rounds

of pruning, each round with an increased κ, until the number of candidates falls below a configurable threshold, or when the maximum number of pruning rounds has been reached.

Phasing algorithm. The above algorithms are combined in the main algorithm PHASE. Based on candidate haplotype sets created with the first $i-1$ superbubbles we generate extensions at superbubble b_i, from which we obtain a new set of candidate haplotype sets, but now with paths through superbubble b_i included. We prune unlikely candidates, and then we move to the next superbubble. At the end, the list of haplotype sets is pruned with $\kappa = 1$, so that only the most likely haplotype sets remain, and if there are multiple most likely candidates one of these haplotype sets is returned at random. The phasing algorithm can be seen in Algorithm 5.

Algorithm 5 The main phasing algorithm, combining EXTEND, BRANCH and PRUNE.

function PHASE($G = (V, E), \tau, \kappa$)
  ▷ Find superbubbles in topological order
  $B \leftarrow$ FINDSUPERBUBBLES(G, nested=FALSE)
  $\mathcal{H} \leftarrow \{H_{empty}\}$
  for all $b_i \in B$ **do**
    $\mathcal{H} \leftarrow$ BRANCH($\mathcal{H}, b_i, \tau$)
    $\mathcal{H} \leftarrow$ PRUNE($\mathcal{H}, b_i, \kappa$)
  return $\mathcal{H}$

2.5. Simulation of aneuploid and polyploid organisms

2.5.1. Synthetic haplotype generation To properly evaluate our algorithm we need to know the ground truth completely. Therefore, we have created a tool called `aneusim` to generate synthetic polyploid or aneuploid genomes. Our method to generate synthetic genomes closely resembles the method described by Motazedi *et al.* [25].

Based on a given reference genome, it creates k copies for each chromosome and randomly mutates each copy. It is possible to generate substitutions, insertions and deletions, and the density of each kind of mutation can be specified using a log-normal distribution. Using a log-normal distribution for mutation densities closely resembles the densities observed in real data [25]. The size distribution of a deletion can also be specified, and configured to follow an exponential or log-normal distribution. All generated mutations are biallelic.

For each mutation, the dosage d_m, i.e. how many chromosomes should get the alternative allele, is picked according to configurable probabilities. For example, a dosage distribution for ploidy level three could be $\frac{1}{2}, \frac{1}{3}, \frac{1}{6}$ for $d_m = 1, 2, 3$ respectively. The dosage distributions we have used at different ploidy levels are shown in the supplemental material. We randomly select which chromosome copies should get the mutation, in concordance with the selected dosage d_m. To account for linkage disequilibrium, a mutation m is assigned to the same chromosomes (with same dosage) as the previous mutation $m-1$ with a given probability p.

2.5.2. Simulation of read data In this work we focus on error-free read data. The tool `aneusim` is also capable of producing PacBio-like reads in terms of read length distribution, but without any errors. Our code is based on SimLoRD [42], but with the error model removed.

2.6. Benchmark genomes generation

We use the genome of the well known yeast *Saccharomyces cerevisiae* as base for our benchmark genomes. We have arbitrarily picked chromosome XIV as our model, and used our synthetic

Genome	Covered	Number of bubble chains	Average number of superbubbles
Ploidy 2	99%	2	21
Ploidy 3	100%	2	26
Ploidy 4	100%	2	22
Ploidy 6	100%	2	5
Ploidy 8	100%	2	12

Table 1 For all our synthetic genomes the resulting assembly graphs are completely covered by the constructed bubble chains, except for ploidy 2, where two edges on the periphery of the assembly graph form an Y-shaped fork (not shown). Still, the vast majority of the assembly graph is covered by a bubble chain. Each genome has two bubble chains, one for the forward strand, and one for the reverse strand, and the average number of superbubbles in a single bubble chain is shown in the last column of the table. The assembly graphs are created from read datasets with 60x coverage per haplotype of the corresponding genome.

haplotype generator to generate genomes of ploidy 2, 3, 4, 6, and 8. We randomly generate substitutions and deletions on each individual homologue. The distance between two substitutions follows a log-normal distribution with $\mu = 3.0349$ and $\sigma = 1.293$. This is the same distribution used in Motazedi *et al.* [25], and resembles the heterozygousity of the potato genome. The mean value of this distribution is 48 bp, with a standard deviation of 100 bp. Additionally, we generate random deletions. Deletions are often more deleterious because of frame shifts, and therefore less conserved. We distribute deletions therefore more sparsely, using a log-normal distribution with $\mu = 7.34$ and $\sigma = 1.123$. The mean value of this distribution is 2895 bp, with a standard deviation of 4603 bp. The size of a deletion follows an exponential distribution, with $\lambda = 0.2$, which shows a nice balance between small deletions (3 bp), and medium sized deletions (20 bp). Supplemental Figures 2, 3 and 4 show the distributions of distances between substitutions, the distances between deletions and the deletion size distribution in all our generated genomes respectively.

3. Results

3.1. Bubble chains cover the majority of the assembly graph

To identify what fraction of the assembly graph is covered by the constructed bubble chains, we check what fraction of vertices are included in a bubble chain. In the ideal case, each set of homologous chromosomes should result in two connected components: one connected component for the forward and one for the reverse strand. For each connected component, we identified its bubble chains, and count how many vertices are included in the bubble chain. Divide this number by the total number of vertices in the connected component to obtain the fraction of vertices part of a bubble chain. In all our generated genomes the bubble chains cover the complete assembly graph, except for ploidy 2, where two edges at the end of the assembly graph form a Y-fork (not shown). Still, the vast majority of the assembly graph is covered by bubble chains. All the results can be seen in Table 1.

3.2. Most superbubbles are small, but the number of paths through a superbubble grows quickly

To investigate the complexity of phasing using a bubble chain we check how large the average superbubble is. For each identified superbubble across all synthetic genomes we collect the size of its interior (the number of vertices inside a superbubble), and determine the number possible paths.

Most superbubbles only contain few vertices in its interior, as shown in Figure 9. But, even a superbubble with few vertices can become complex as the number of vertices does not necessarily equal the number of paths through this superbubble — in a fully connected graph, the number of simple paths is of the order $O(n!)$, with n the number of vertices.

The number of paths through a superbubble, from its entrance to exit, is low most of the time, as shown in Figure 10. There are instances however, with a much higher number of possible paths. For example, we observe a superbubble with 76 possible paths from its entrance to exit. These kind of instances would require too much time to phase, because there are p^k k-tuples of these paths, where p is the number of paths and k the ploidy. If we encounter a superbubble of 76 paths in an organism with ploidy 6, then there are $76^6 = 192699928576$ possible k-tuples, too much to check them all. Even with ten paths through a superbubble the number of possible configurations grows too quickly. PHASM will not attempt to phase these superbubbles. Any superbubble with ten or more possible paths from its entrance to exit will be "popped" similar to traditional *de novo* assemblers: only output the DNA sequence for the path that is best supported by read data.

Recall that in Section 2.4 we assumed that the number of paths through a superbubble is of the order $O(k)$. We do not observe any bias towards the ploidy levels. This can be explained by recalling that all generated mutations are biallelic, so at each mutation there are only two options, either the reference allele or the alternate allele. Furthermore, chromosome homologues may be locally similar due to similar dosages of mutations.

3.3. Paths through a superbubble can span multiple and structural variants

One of the disadvantages of SNV based phasing is that these approaches do not handle structural and large variants very well. Phasing with an assembly graph should be able to capture these kind of variants better because a single path through a superbubble should be able to span multiple substitutions and large indels.

To see if that is indeed the case, we have plotted the length of each path through all superbubbles found across all genomes, from its entrance to corresponding exit in Figure 11. The length of a path is defined as the sum of edge lengths part of the path, i.e. the sum of unmatched prefixes of each local alignment corresponding to an edge.

It can be seen that most paths through superbubbles are long: the average path length is 49740 bases, with the smallest path being 1457 bases and the longest path 256643 bases. Paths with these lengths are long enough to capture multiple and structural variants.

3.4. The distance between two consecutive bubbles largely determines the number of spanning reads

To phase paths between two consecutive superbubbles the algorithm needs reads that connect different paths through these two superbubbles, so called spanning reads (Definition 2.8). To

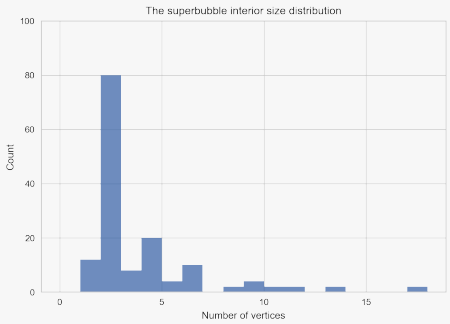


Figure 9 The distribution of the number of vertices inside a superbubble (its interior). This figure contains the results for all superbubbles found across all genomes. The number of vertices give an initial indication on the complexity of most superbubbles.

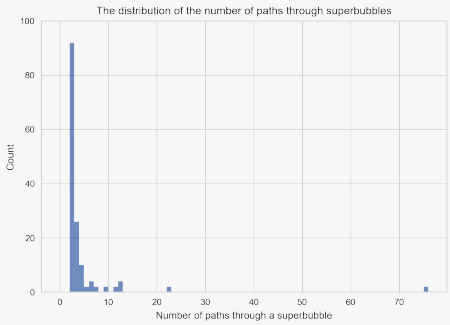


Figure 10 The distribution of the number of possible paths through a superbubble. This figure contains the results for all superbubbles found across all genomes. We see that most superbubbles have two paths through them, which is to be expected when generating biallelic mutations. In a few rare cases the superbubbles are very large, even up to 76 possible paths. These superbubbles are too large to phase because there are too many possible configurations to check.

investigate what determines the number of spanning reads and verify that our definition of spanning reads makes sense we have plotted the number of spanning reads between any pair of consecutive superbubbles against the distance between these two superbubbles. This is shown in Figure 12. This plot does not include any superbubble without spanning reads, and we have normalised the number of spanning reads for ploidy, by dividing the actual number of spanning reads by the ploidy level. The data shows what is to be expected given our read length distribution (Supplementary Figure 1).

The distance between two consecutive superbubbles $b_1 = \langle v_1, v_2 \rangle, b_2 = \langle v_2, v_3 \rangle$ is defined as the minimum edge length of all outgoing edges from vertex v_2. v_2 is the exit of the first superbubble and the entrance of the second superbubble. The smallest outgoing edge, i.e. the shortest unmatched prefix, is therefore the first encounter of something different at superbubble b_2, coming from b_1.

3.5. Aggressive pruning is required to keep the problem tractable

To investigate the effect of discarding candidate extensions with a relative likelihood lower than a threshold τ and the pruning step in the phasing algorithm, we have logged all calculated relative likelihood values for all generated candidate extensions across all genomes. The relative likelihood values are plotted in Figure 13. The black dashed line represents the threshold τ we have used in our runs (0.001) — few candidates will be discarded using this threshold.

Additionally, PHASM employs a pruning step: at each superbubble, the algorithm searches for the candidate extension with the highest relative likelihood ω, and any other candidate with a relative likelihood lower than $\kappa\omega$ is discarded. To obtain an idea how many candidates will be discarded at different values for κ, we plot the relative likelihood for each candidate divided by

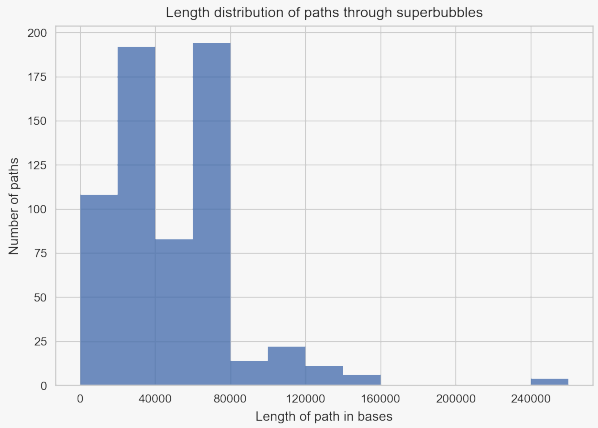


Figure 11 The distribution of path lengths through superbubbles, from its entrance to exit. Paths are long enough to span multiple and/or structural variants.

by the maximum relative likelihood ω at the corresponding superbubble. This distribution is shown in Figure 14. This plot excludes the candidates for which hold that $\frac{RL[E|H]}{\omega} = 1$, that is the candidates that are the most likely at a certain superbubble. This is to prevent bias in the bin that contains all candidates with $\frac{RL[E|H]}{\omega} = 1$.

Taking a closer look at Figure 14, we observe a large peak around $\frac{RL[E|H]}{\omega} = 0.5$. Because most superbubbles contain two paths (Section 3.2), an explanation for this peak is that candidates with $\frac{RL[E|H]}{\omega} \approx 0.5$ only use one path for its haplotypes, while the candidate with the highest relative likelihood uses both paths. Similar reasoning can be applied for the peaks around 0.25 and 0.75.

In our PHASM runs we try to keep the number of *likely* candidates at each superbubble below 500, otherwise it will take too much time to phase a bubble chain. Keeping the number of candidates below 500 means when moving to the next superbubble, we have at most 500 "true" candidate haplotypes H built using the previous superbubbles which will be connected to possible extensions E. To achieve this PHASM often applies multiple rounds of pruning. It starts with $\kappa = 0.1$, and increases this value with steps of 0.1 until the number of candidates falls below 500 or κ becomes 1.0. In this last case, only candidates that are equally likely as the candidate with the maximum relative likelihood remain.

From our data we can recover how often multiple pruning rounds are applied and we have plotted how often a certain κ value is reached in Figure 15. This figure shows what fraction of pruning rounds reached a certain κ value, aggregated across all our runs, and plotted separately per superbubble position in a bubble chain or haploblock.

We see that the longer a bubble chain or haploblock, the more often we apply pruning with a pruning factor of $\kappa = 1.0$. This means that only candidates that are equally likely as the candidate with the highest relative likelihood remain. The more stringent pruning can be partially explained by the exponential growth of the problem, but we did not expect to require such high values for κ that often. This can hamper the accuracy of the algorithm.

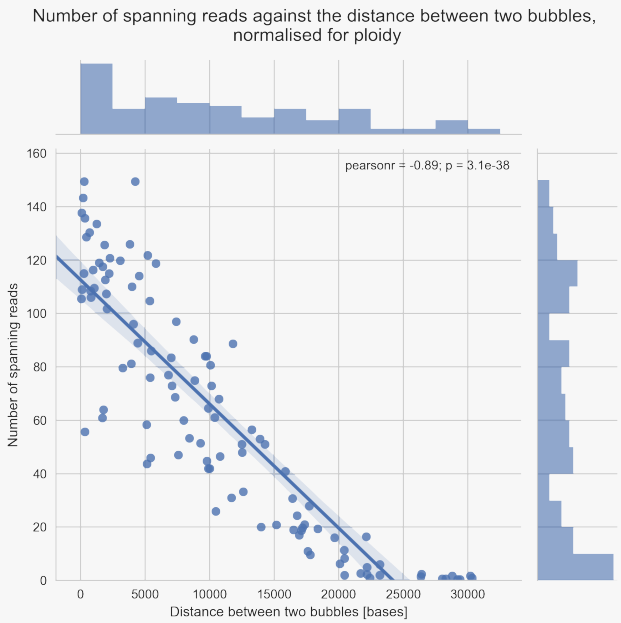


Figure 12 The number of spanning reads against the distance between two superbubbles. Each dot in this figure represents a superbubble in any of the genomes with at least one spanning read. We normalised the number of spanning reads across different genomes by dividing the actual number of spanning reads by the ploidy level. Superbubbles without spanning reads lie in general much farther to the right (higher distance between superbubbles). The shaded area represents the 95% confidence interval of the regression line.

3.6. PHASM outputs inaccurate sequences

To evaluate the sequences produced by PHASM, and see if each sequence matches one of the haplotypes, we used mummer to produce mummerplots. How we have run the program "nucmer" is explained in the supplemental material. PHASM does not output exactly k DNA sequences, because sometimes there are not enough spanning reads between bubbles, and a new "haploblock" is started. PHASM does output k DNA sequences (contigs) for each haploblock. The expected output is that each contig within a haploblock matches with exactly one of the reference haplotype.

The sequences produced by PHASM are inaccurate. A contig within a haploblock rarely matches one of the reference haplotypes exactly. Furthermore, not all reference haplotypes are evenly covered. In Figure 16 shows the results for the genome with ploidy 2. The purple lines represent the forward strand, the blue lines the reverse strand. Contigs within a haploblock are grouped together. We see that the first reference haplotype is covered by more contigs (there are more lines in the first column), and that there are several "switch errors", where one column shows a gap and the other column shows a match.

The mummer plots for the genomes with higher ploidy show even worse results, with less contiguity and less exact matches. These plots become unreadable on paper, and therefore not included in this work, but these can be viewed online in our Github repository.

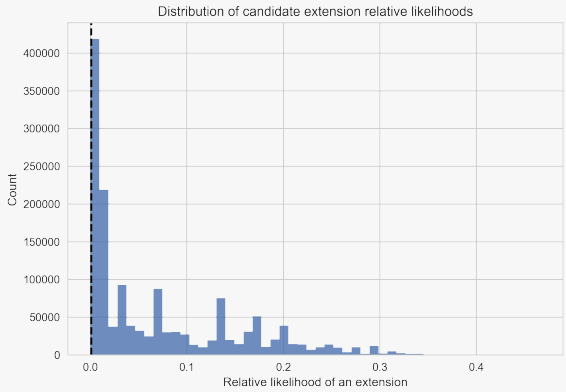


Figure 13 The relative likelihood $R[E|H]$ for all generated candidate extensions E. Our runs of PHASM used a threshold $\tau = 0.001$ (black dashed line). This figure shows that this threshold could have been increased a bit, because few candidates fall below this threshold.

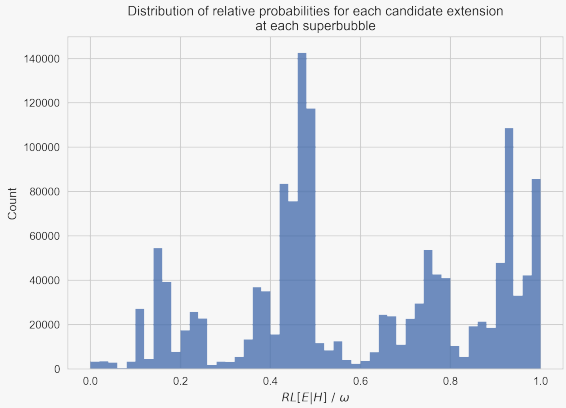


Figure 14 In Section 2.4 we describe a pruning algorithm that searches for an $\omega = \max_{E \in \mathcal{E}} RL[E|H]$ at each superbubble. Any other candidate with a relative likelihood lower than $\kappa\omega$ is pruned, with κ the prune factor. This plot shows how the relative likelihood of each candidate divided by the maximum relative likelihood at the corresponding superbubble is distributed. It gives an indication on how many candidates will be pruned for different pruning factors. This plot does not include candidates that have the highest relative likelihood at its corresponding superbubble, that is candidates for which holds $\frac{R[E|H]}{\omega} = 1$.

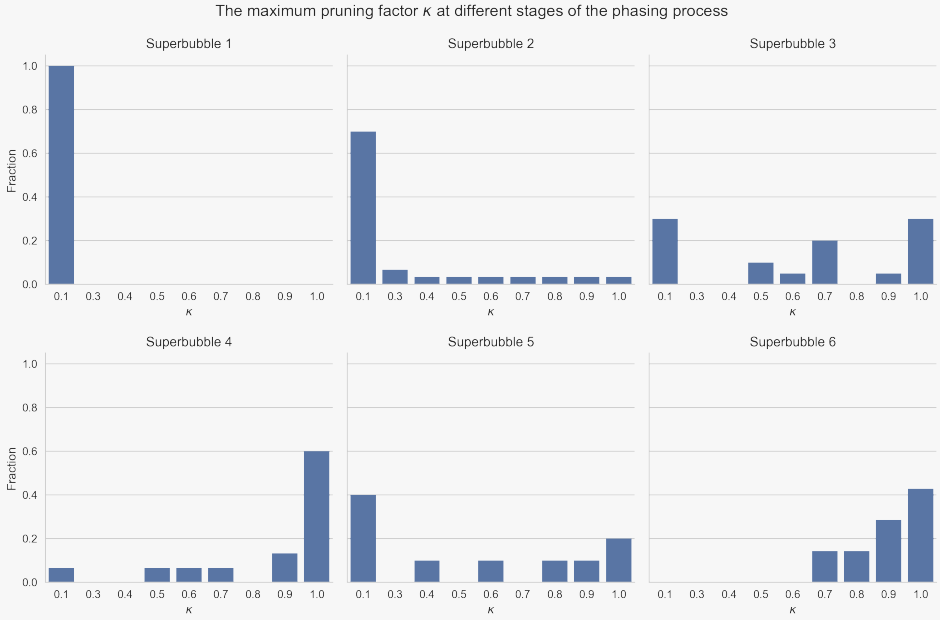


Figure 15 The number of pruning rounds, plotted per superbubble position in a bubble chain (haploblock). This figures shows what fraction of pruning rounds reached a certain κ value, plotted separately per position in a bubble chain or haploblock. We see that the longer the bubble chain, the more often strict pruning is required to keep the number of candidates below 500.

3.7. The majority of the generated alternative alleles have representation in the assembly graph

To investigate why the sequences are inaccurate, we start by checking the generated mutations on each synthetic haplotype. For each mutation, we have kept metadata about its position on the reference sequence, its position on the actual haplotype sequence, the kind of mutation (substitution, deletion, insertion) and its size. Furthermore, our read simulator also outputs a metadata file, containing from which haplotype a read is sampled, its starting position and its length. We can use this data to check whether each alternative allele has a representation in the assembly graph.

Using the read metadata, we build separate interval trees for each haplotype, using the read starting position and its length to denote the interval. These interval trees can be used to quickly query which reads cover a certain position in the haplotype. For each mutation we obtain its position on the haplotype sequence and check which reads cover this mutation, taking into account that a mutation can have a dosage $d_m > 1$, and that the alternative allele can occur on multiple haplotypes. By taking the intersection between the set of reads covering an alternative allele and the set of reads playing a role in the assembly graph, we can answer the question whether this alternative allele has representation in the assembly graph.

We see that almost all alternative alleles have representation in the assembly graph, but certainly not all of them. This problem is more prominent in higher ploidy. The results can be seen in Figure 17.

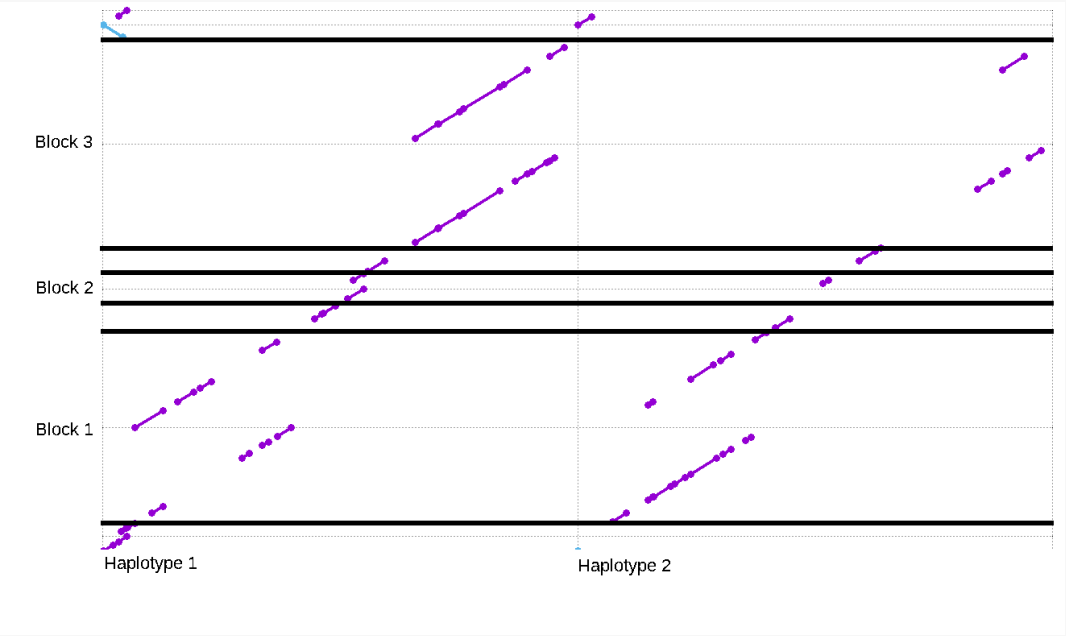


Figure 16 A mummerplot to compare the output produced by PHASM with the reference haplotypes. This figure shows the result for the genome with ploidy 2. We have marked each haploblock in the figure with thick black lines. Within each block, PHASM outputs two sequences: one for each haplotype. The figure shows that the sequences are inaccurate: we see switch errors, and not all reference haplotypes are equally covered. Regions without a block number are either large bubbles or edges not part of a bubble chain.

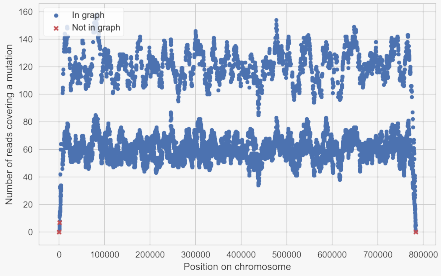
3.8. Paths through a single superbubble are rarely consistent with one of the reference haplotypes

To check whether superbubbles capture all variation between the homologous chromosomes we spell the sequence for each path in a single superbubble and check with which reference haplotypes it is consistent. Note that a path could be consistent with multiple reference haplotypes if these haplotypes are locally similar.

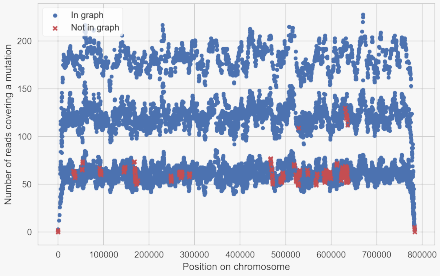
Surprisingly, we see that the sequence of a path through a superbubble is rarely consistent with *any* of the haplotypes. For each superbubble found in any of the synthetic genomes, we have plotted the fraction of paths that are consistent with at least one of the reference haplotypes. This is shown in Figure 18.

Only one superbubble out of all 172 found superbubbles has all paths consistent with at least one of the haplotypes. But, even this superbubble with all paths consistent is incomplete: this superbubble comes from a genome with ploidy three, and this superbubble only explains two out of the three haplotypes. This superbubble therefore misses any variation included in the third haplotype.

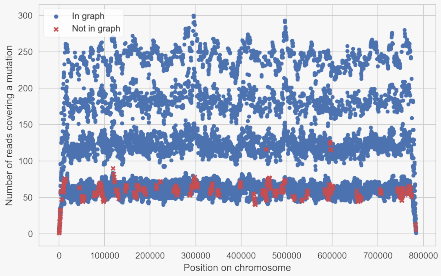
Apparently, bubble chains are an incomplete and wrong representation of variation among homologous chromosomes.



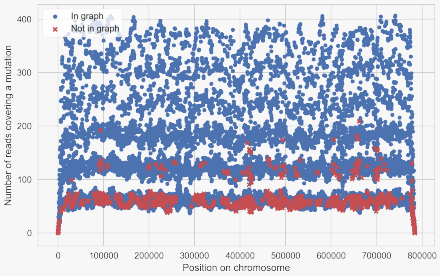
(a) Ploidy 2



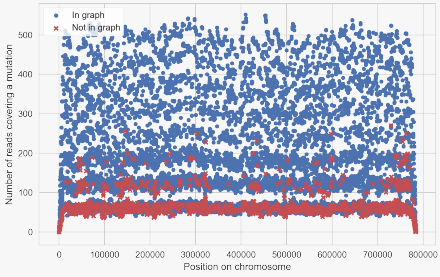
(b) Ploidy 3



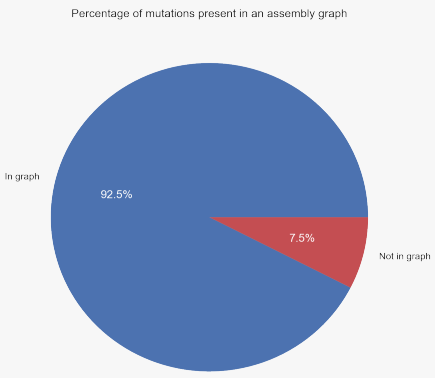
(c) Ploidy 4



(d) Ploidy 6



(e) Ploidy 8



(f) Pie chart showing what fraction of the mutations are included in the assembly graph.

Figure 17 These figures show all our generated alternative alleles, and whether they have representation in our assembly graph. The difference in the number of covering reads represent the differences in dosages d_m. Please note that the y-axis differs for each ploidy.

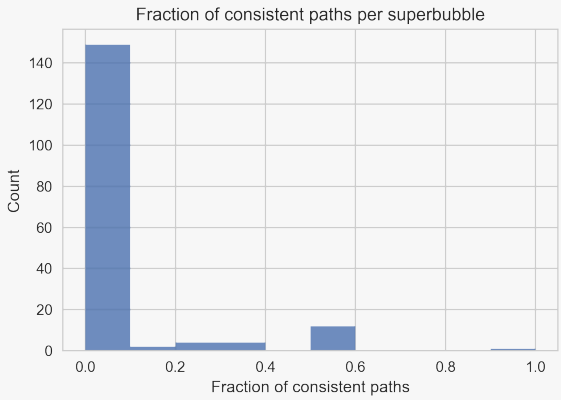


Figure 18 This plot shows the fraction of paths that are consistent with at least one of the reference haplotypes for each superbubble. The results are aggregated across all found superbubbles in all our synthetic genomes. Very few superbubbles contain paths that are consistent with one of the haplotypes.

3.9. DALIGNER reports many more local alignments than an exact overlapper

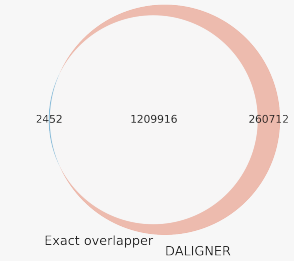
To investigate the influence of the approximate alignments reported by DALIGNER, even when using strict settings, we have built a tool to identify exact pairwise overlaps. This tool uses a suffix tree to search for overlapping reads, and the full algorithm is described in the supplemental material.

We have compared the reported alignments by DALIGNER with the reported exact overlaps by our own overlapper. The results can be seen in Figure 19. We see that DALIGNER reports many more pairwise alignments than the exact overlapper. This is unexpected given the high heterozygousity of our synthetic genomes and the strict settings used for DALIGNER.

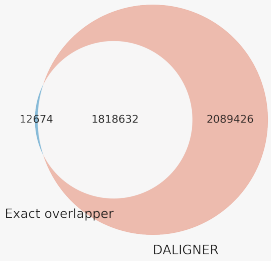
DALIGNER is designed for PacBio data, and employs some heuristics to make it fast. We may trigger some edge case that can explain the additional reported alignments.

Although the exact overlapper reports less pairwise overlaps, the assembly graph created using these overlaps does represent almost all alternative alleles for our generated mutations. This is shown in Figure 20. The only alleles without representation in the assembly graph occur on the periphery, with no read covering these alleles, or where their covering reads get removed by the tip removal step.

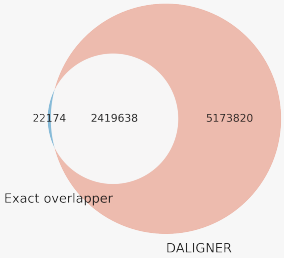
As a last note, none of the assembly graphs constructed from exact overlaps contain superbubbles.



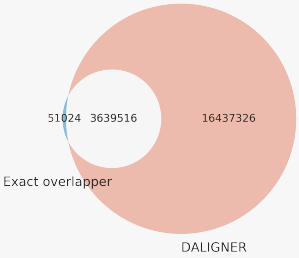
(a) Ploidy 2



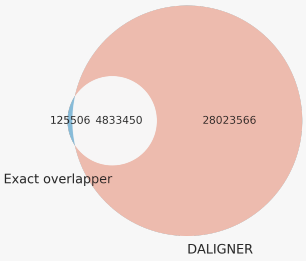
(b) Ploidy 3



(c) Ploidy 4

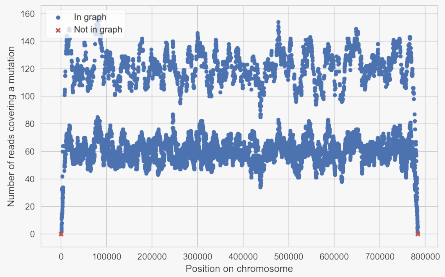


(d) Ploidy 6

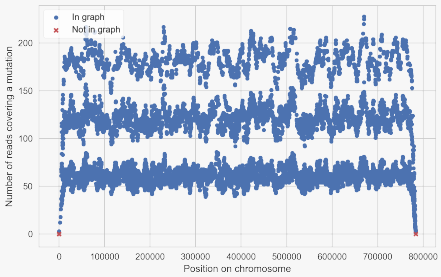


(e) Ploidy 8

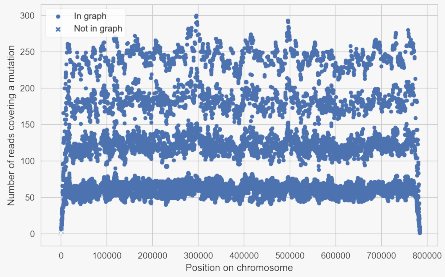
Figure 19 The number of reported pairwise overlaps/alignments using our exact overlapper versus using DALIGNER. We see that DALIGNER reports many more pairwise alignments, even though we have used strict settings.



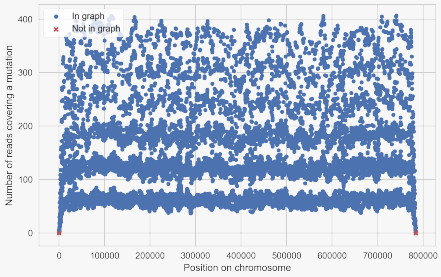
(a) Ploidy 2



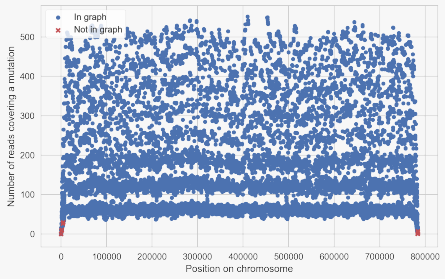
(b) Ploidy 3



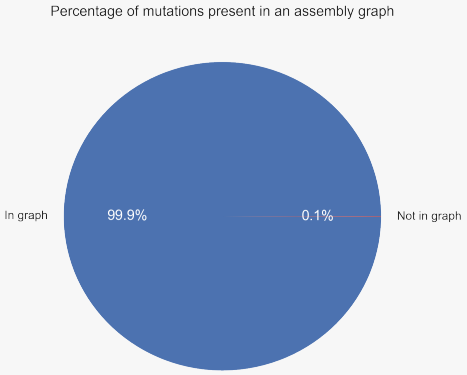
(c) Ploidy 4



(d) Ploidy 6



(e) Ploidy 8



(f) Pie chart showing what fraction of the mutations are included in the assembly graph.

Figure 20 When the assembly graph is constructed from pairwise exact overlaps, almost all alternative alleles have representation. Only mutations on the periphery sometimes have no representation in the assembly graph. This is caused when no reads cover this mutation, or the read gets removed by the tip removal stage. Please note that the y-axis differs for each ploidy.

4. Discussion

In this work we have introduced PHASM: a prototype *de novo* genome assembler that outputs DNA sequences for each chromosome homologue. Unfortunately, the output is inaccurate and we are unable to answer all our research questions in this work.

While bubble chains initially looked like a promising answer to the question *how to capture variation among homologous chromosomes in an assembly graph?*, we see that paths through a superbubble are rarely consistent with one of the haplotypes. Furthermore, not all variation among homologous chromosomes is included in the assembly graph.

The phasing algorithm is built on the assumption that each path through a superbubble represents a set of alleles consistent with at least one of the haplotypes, and the goal would be to link paths between two consecutive superbubbles. Because this assumption is false and paths in superbubbles rarely represent any chromosome, we are unable to properly evaluate the phasing algorithm. We see some positive signs, for example the peak around $\frac{RL[E|H]}{\omega} = 0.5$ in Figure 14, but we can not answer yet how well the probabilistic model and algorithm perform in building correct haplotypes. Therefore, we have no definitive answer for the question *how to find the best set of paths through an assembly graph such that each path represents a haplotype?*

Fully haplotype resolved polyploid genomes are not widely available, so we have presented a tool to generate synthetic polyploid genomes. Using synthetic genomes for evaluation ensures we know the ground truth completely. Our tool also stores metadata for all generated mutations which in aid in more detailed analyses of the results. We believe we have presented a solid approach for the evaluation of the algorithm, although there are still possibilities for extension. For example, we have not investigated the effect of depth of read coverage per haplotype, heterozygousity of the genome, or the effect of the read length distribution.

PHASM does not take any sequencing error into account. Real sequencing data, especially reads from the third generation sequencing platforms, do contain error. In this work we have shown that even with error-free data and strict approximate alignments, it is already tricky to capture the variation among homologous chromosomes correctly in an assembly graph. With more relaxed approximate alignments and sequencing error included, this problem will become more difficult. Reads will be classified more quickly as contained as thus ignored; the transitive reduction step will remove more edges because of the additional pairwise alignments, possibly removing alternative paths; and with approximate alignment there is a higher chance that reads from different chromosomes get chained together, possibly resulting in a path that is inconsistent with all chromosomes.

5. Reflection and Future Work

Although we have not answered all our research questions, we do have gained more insight in the problem of reconstructing haplotypes using an assembly graph. We have shown that extra care is required when building the assembly graph, to prevent the creation of paths that are inconsistent with all haplotypes.

Genome assemblers are complex computational pipelines consisting of steps as pairwise overlap/local alignment identification, read error correction, assembly graph construction and cleaning, building contigs, generating consensus sequences and scaffolding. Adapting the pipeline such that it is able to deal with multiple haplotypes requires modifications to almost all steps. It would be ambitious to fix all steps in a ten month project. We therefore have bound the project to error-free data, and focussed on the phasing algorithm. But, we have neglected to check some

assumptions made early, resulting in an algorithm that could not be properly evaluated in this work. An important lesson to take away would be "check your data early, and check your data often".

Nevertheless, we still think haplotype phasing for polyploid organims using an assembly graph and long read data is a promising approach. In the rest of this section we propose a different approach to answer our research question.

In this work we have approached the problem too much from a "phasing" point of view. We started out by investigating the current SNV based haplotype assembly algorithms, and attempted to apply a similar algorithm on an assembly graph. We believe a more successful approach would start from a "*de novo* genome assembly" point of view. Start with the read data and their pairwise local alignments. Classify whether local alignments between two reads are either repeat induced, between different haplotypes, or a correct overlap in the underlying genome. With this information the assembly graph can be created with more care, preventing incorrect overlaps. For example, Canu does not create a complete assembly string graph, an edge between two reads u and v is only created if v is the best overlapping read with u [33]. Furthermore, they apply a statistical method to detect repeat induced overlaps. Recently, Tischler presented a new tool called *d'accord* which is able to separate correct and incorrect (from different haplotype, repeat induced) read overlaps near perfectly, even for higher ploidy organisms [43]. The pile of correct overlaps is then used to perform an error correction step [43, 44].

We believe the work by Tischler is a good starting point for a new attempt to answer the research question posed in this work. We envision the following pipeline:

- perform approximate pairwise alignment, allowing for mismatches and gaps, to deal with sequencing error;
- use *d'accord* to separate correct and incorrect overlaps, and perform read error correction.
- create a sparse assembly string graph, by only creating edges for the best overlapping reads. In this step, any overlaps classified as incorrect in the previous step are ignored;
- find paths through the assembly graph for each haplotype.

Availability

PHASM is implemented in Python and C++ and available at https://github.com/AbeelLab/phasm. The complete pipeline, combining importing reads to DAZZ_DB, finding pairwise local alignments with DALIGNER, converting this to GFA2, building the assembly graph, building bubble chains and phasing these bubble chains is implemented as a Snakemake workflow [45]. This full pipeline can be found on https://github.com/AbeelLab/phasm-benchmarks. The synthetic genome simulator tool `aneusim` can be found on https://github.com/AbeelLab/aneusim.

Acknowledgements

First and foremost, I would like to thank my supervisor Dr. Thomas Abeel a lot, for his sharpness, his critical eye, for being a very good mentor, and for helping me obtain a position at the Broad Institute. Thank you very much. I would like to thank prof. dr. ir. Marcel Reinders for giving me a 9 for the first course I took in the bioinformatics master programme, igniting the ambition to get similar grades for the following courses. I would like to thank all people from the Delft Bioinformatics Lab for their nice presentations, discussions and colloquia.

Furthermore, I would like to express my extreme gratitude towards my parents, who have

supported me throughout all my years of study, who have supported all my switches between study programmes, and have given me the freedom to search for a field of study where I can actually see my self working the rest of my life.

References

[1] M. L. A. Jansen, J. M. Bracher, I. Papapetridis, M. D. Verhoeven, and H. De, "Saccharomyces cerevisiae strains for second-generation ethanol production : from academic exploration to industrial implementation", *FEMS Yeast Research*, no. June, pp. 1–20, 2017.

[2] F. H. Lam, A. Ghaderi, G. R. Fink, and G. Stephanopoulos, "Engineering alcohol tolerance in yeast", *Science*, vol. 346, no. 6205, pp. 71–75, 2014.

[3] M. van den Broek, I. Bolat, J. F. Nijkamp, E. Ramos, M. A. H. Luttik, F. Koopman, J. M. Geertman, D. de Ridder, J. T. Pronk, and J. M. Daran, "Chromosomal copy number variation in Saccharomyces pastorianus is evidence for extensive genome dynamics in industrial lager brewing strains", *Applied and Environmental Microbiology*, vol. 81, no. 18, pp. 6253–6267, 2015.

[4] M. Hebly, A. Brickwedde, I. Bolat, M. R. Driessen, E. A. de Hulster, M. van den Broek, J. T. Pronk, J.-M. Geertman, J.-M. Daran, and P. Daran-Lapujade, "S. cerevisiae × S. eubayanus interspecific hybrid, the best of both worlds and beyond", *FEMS Yeast Research*, vol. 15, no. 3, pp. 1–14, May 2015.

[5] K. Krogerus, F. Magalh??es, V. Vidgren, and B. Gibson, "New lager yeast strains generated by interspecific hybridization", *Journal of Industrial Microbiology and Biotechnology*, vol. 42, no. 5, pp. 769–778, 2015.

[6] J. Steensels, E. Meersman, T. Snoek, V. Saels, and K. J. Verstrepen, "Large-scale selection and breeding to generate industrial yeasts with superior aroma production", *Applied and Environmental Microbiology*, vol. 80, no. 22, pp. 6965–6975, 2014.

[7] B. R. Gibson, E. Storgårds, K. Krogerus, and V. Vidgren, "Comparative physiology and fermentation performance of Saaz and Frohberg lager yeast strains and the parental species Saccharomyces eubayanus", *Yeast*, vol. 30, no. 7, pp. 255–266, Jul. 2013.

[8] D. M. Church, V. A. Schneider, K. Steinberg, M. C. Schatz, A. R. Quinlan, C.-S. Chin, P. A. Kitts, B. Aken, G. T. Marth, M. M. Hoffman, J. Herrero, M. L. Mendoza, R. Durbin, and P. Flicek, "Extending reference assembly models", *Genome Biology*, vol. 16, no. 1, p. 13, 2015.

[9] C.-S. Chin, P. Peluso, F. J. Sedlazeck, M. Nattestad, G. T. Concepcion, A. Clum, C. Dunn, R. O'Malley, R. Figueroa-Balderas, A. Morales-Cruz, G. R. Cramer, M. Delledonne, C. Luo, J. R. Ecker, D. Cantu, D. R. Rank, and M. C. Schatz, "Phased diploid genome assembly with single-molecule real-time sequencing", *Nature Methods*, vol. 13, no. 12, pp. 1050–1054, Oct. 2016.

[10] S. R. Browning and B. L. Browning, "Haplotype phasing: existing methods and new developments", *Nature Reviews Genetics*, vol. 12, no. 10, pp. 703–714, Sep. 2011.

[11] M. W. Snyder, A. Adey, J. O. Kitzman, and J. Shendure, "Haplotype-resolved genome sequencing: experimental methods and applications", *Nature Reviews Genetics*, vol. 16, no. 6, pp. 344–358, 2015.

[12] The Computational Pan-genomics Consortium, "Computational pan-genomics: status, promises and challenges", *Briefings in Bioinformatics*, no. May, bbw089, 2016.

[13] M. W. Bevan, C. Uauy, B. B. H. Wulff, J. Zhou, K. Krasileva, and M. D. Clark, "Genomic innovation for crop improvement", *Nature*, vol. 543, no. 7645, pp. 346–354, 2017.

[14] C.-S. Chin, D. H. Alexander, P. Marks, A. A. Klammer, J. Drake, C. Heiner, A. Clum, A. Copeland, J. Huddleston, E. E. Eichler, S. W. Turner, and J. Korlach, "Nonhybrid, finished microbial genome assemblies from long-read SMRT sequencing data", *Nature Methods*, vol. 10, no. 6, pp. 563–569, 2013.

[15] S. Koren and A. M. Phillippy, "One chromosome, one contig: Complete microbial genomes from long-read sequencing and assembly", *Current Opinion in Microbiology*, vol. 23, pp. 110–120, 2015.

[16] R. Lippert, R. Schwartz, G. Lancia, and S. Istrail, "Algorithmic strategies for the single nucleotide polymorphism haplotype assembly problem.", *Briefings in bioinformatics*, vol. 3, no. 1, pp. 23–31, 2002.

[17] P. Bonizzoni, R. Dondi, G. W. Klau, Y. Pirola, N. Pisanti, and S. Zaccaria, "On the Fixed Parameter Tractability and Approximability of the Minimum Error Correction Problem", in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9133, 2015, pp. 100–113.

[18] V. Bansal and V. Bafna, "HapCUT: An efficient and accurate algorithm for the haplotype assembly problem", *Bioinformatics*, vol. 24, no. 16, pp. 153–159, 2008.

[19] Z. Z. Chen, F. Deng, and L. Wang, "Exact algorithms for haplotype assembly from whole-genome sequence data", *Bioinformatics*, vol. 29, no. 16, pp. 1938–1945, 2013.

[20] D. He, A. Choi, and K. Pipatsrisawat, "Optimal algorithms for haplotype assembly from whole-genome sequence data", *Bioinformatics*, vol. 26, no. 12, pp. i183–90, 2010.

[21] D. He, B. Han, and E. Eskin, "Hap-seq: An Optimal Algorithm for Haplotype Phasing with Imputation Using Sequencing Data", *Journal of Computational Biology*, vol. 20, no. 2, pp. 80–92, 2013.

[22] V. Kuleshov, "Probabilistic single-individual haplotyping", *Bioinformatics*, vol. 30, no. 17, pp. 379–385, 2014.

[23] M. Patterson, T. Marschall, N. Pisanti, L. van Iersel, L. Stougie, G. W. Klau, and A. Schonhuth, "WHATSHAP: Haplotype Assembly for Future-Generation Sequencing Reads", *Research in Computational Molecular Biology, Recomb2014*, vol. 8394, pp. 237–249, 2014.

[24] Y. Pirola, S. Zaccaria, R. Dondi, G. W. Klau, N. Pisanti, and P. Bonizzoni, "HapCol: Accurate and memory-efficient haplotype assembly from long reads", *Bioinformatics*, vol. 32, no. 11, pp. 1610–1617, 2016.

[25] E. Motazedi, R. Finkers, C. Maliepaard, and D. de Ridder, "Exploiting next-generation sequencing to solve the haplotyping puzzle in polyploids: a simulation study", *Briefings in Bioinformatics*, bbw126, Jan. 2017.

[26] D. Aguiar and S. Istrail, "Haplotype assembly in polyploid genomes and identical by descent shared tracts", *Bioinformatics*, vol. 29, no. 13, pp. 352–360, 2013.

[27] D. Aguiar and S. Istrail, "HapCompass: a fast cycle basis algorithm for accurate haplotype assembly of sequence data.", *Journal of computational biology*, vol. 19, no. 6, pp. 577–90, 2012.

[28] S. Das and H. Vikalo, "SDhaP: haplotype assembly for diploids and polyploids via semi-definite programming", *BMC Genomics*, vol. 16, no. 1, p. 260, Dec. 2015.

[29] E. Berger, D. Yorukoglu, J. Peng, and B. Berger, "HapTree: A Novel Bayesian Framework for Single Individual Polyplotyping Using NGS Data", *PLoS Computational Biology*, vol. 10, no. 3, I. Rigoutsos, Ed., e1003502, Mar. 2014.

[30] S. Levy, G. Sutton, P. C. Ng, L. Feuk, A. L. Halpern, B. P. Walenz, N. Axelrod, J. Huang, E. F. Kirkness, G. Denisov, Y. Lin, J. R. MacDonald, A. W. C. Pang, M. Shago, T. B. Stockwell, A. Tsiamouri, V. Bafna, V. Bansal, S. A. Kravitz, D. A. Busam, K. Y. Beeson, T. C. McIntosh, K. A. Remington, J. F. Abril, J. Gill, J. Borman, Y. H. Rogers, M. E. Frazier, S. W. Scherer, R. L. Strausberg, and J. C. Venter, "The diploid genome sequence of an individual human", *PLoS Biology*, vol. 5, no. 10, pp. 2113–2144, 2007.

[31] D. R. Zerbino and E. Birney, "Velvet: Algorithms for de novo short read assembly using de Bruijn graphs", *Genome Research*, vol. 18, no. 5, pp. 821–829, 2008.

[32] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. Jones, and I. Birol, "ABySS: A parallel assembler for short read sequence data", *Genome Research*, vol. 19, no. 6, pp. 1117–1123, Jun. 2009.

[33] S. Koren, B. P. Walenz, K. Berlin, J. R. Miller, N. H. Bergman, and A. M. Phillippy, "Canu: scalable and accurate long-read assembly via adaptive k -mer weighting and repeat separation", *Genome Research*, vol. 27, no. 5, pp. 722–736, May 2017.

[34] H. Li, "Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences", *Bioinformatics*, vol. 32, no. 14, pp. 2103–2110, Jul. 2016.

[35] E. W. Myers, "Efficient Local Alignment Discovery amongst Noisy Long Reads", in *Algorithms in Bioinformatics: 14th International Workshop*, 2014, pp. 52–67.

[36] E. W. Myers, "The fragment assembly string graph", *Bioinformatics*, vol. 21, no. SUPPL. 2, pp. 79–85, 2005.

[37] T. Onodera, K. Sadakane, and T. Shibuya, "Detecting Superbubbles in Assembly Graphs", in *Algorithms in Bioinformatics: 13th International Workshop*, 2013, pp. 338–348.

[38] Wing-Kin Sung, K. Sadakane, T. Shibuya, A. Belorkar, and I. Pyrogova, "An O(m log m)-Time Algorithm for Detecting Superbubbles", *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 12, no. 4, pp. 770–777, Jul. 2015.

[39] L. Brankovic, C. S. Iliopoulos, R. Kundu, M. Mohamed, S. P. Pissis, and F. Vayani, "Linear-Time Superbubble Identification Algorithm for Genome Assembly", *arXiv*, 2015.

[40] J. F. Nijkamp, M. A. Van Den Broek, J. M. A. Geertman, M. J. T. Reinders, J. M. G. Daran, and D. De Ridder, "De novo detection of copy number variation by co-assembly", *Bioinformatics*, vol. 28, no. 24, pp. 3195–3202, 2012.

[41] M. C. Martínez, "Aneuploid chromosome copy number estimation with dynamic coverage analysis", Master Thesis, Delft University of Technology, 2017.

[42] B. K. Stöcker, J. Köster, and S. Rahmann, "SimLoRD: Simulation of Long Read Data", *Bioinformatics*, vol. 32, no. 17, pp. 2704–2706, Sep. 2016.

[43] G. Tischler, "Haplotype And Repeat Separation In Long Reads", *bioRxiv*, no. 1, pp. 1–6, 2017.

[44] G. Tischler and E. W. Myers, "Non Hybrid Long Read Consensus Using Local De Bruijn Graph Assembly", *bioRxiv*, pp. 1–42, 2017.

[45] J. Köster and S. Rahmann, "Snakemake-a scalable bioinformatics workflow engine", *Bioinformatics*, vol. 28, no. 19, pp. 2520–2522, 2012.

SUPPLEMENTAL MATERIAL

A Bayesian approach to haplotype-aware *de novo* genome assembly for polyploid organisms

Lucas Roeland van Dijk
Delft University of Technology

Table of Contents

A. Supplemental Figures

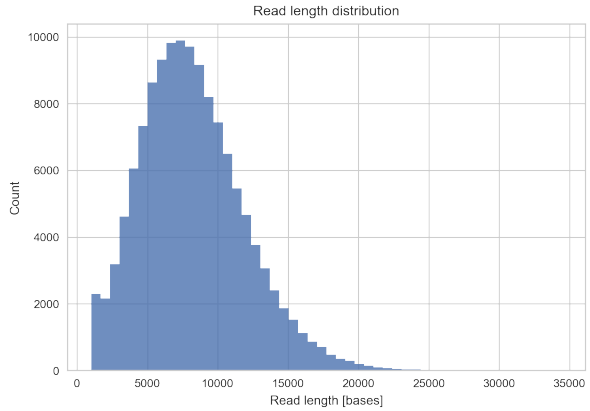


Figure 1 The read length distribution aggregated over all read datasets

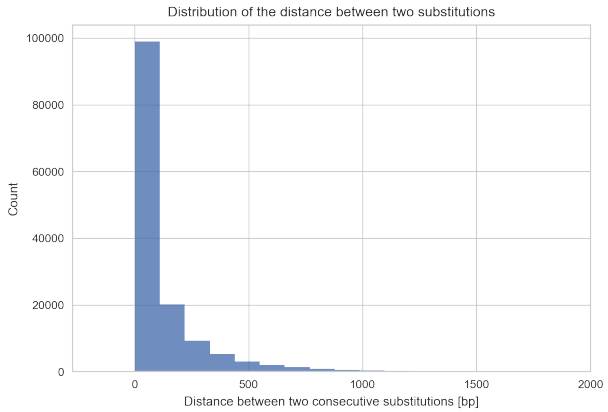


Figure 2 In our synthetic genomes, we generate random mutations. This figure shows the distribution of the distances between two consecutive *substitutions*, across all generated genomes.

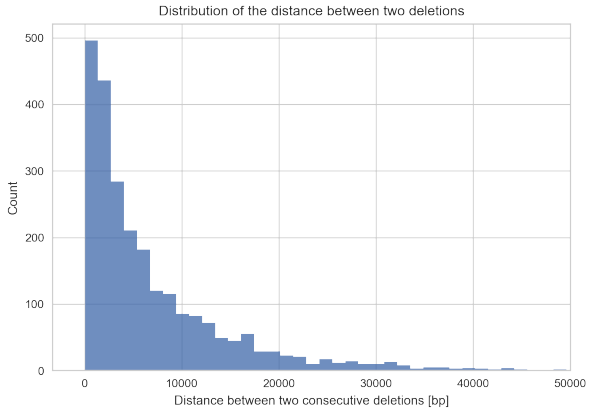


Figure 3 In our synthetic genomes, we generate random mutations. This figure shows the distribution of the distances between two consecutive *deletions*, across all generated genomes.

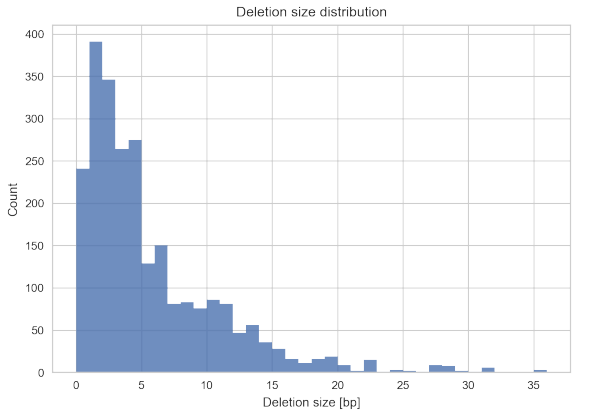


Figure 4 The distribution of the size of a deletion.

B. Finding haplotypes as a network flow problem

One of our research questions is how to find the best set of paths through an assembly graph such that each path represents a haplotype. Inspired by methods using network flow like copy number estimation of edges in a string graph [1], *de novo* assembly with a De Bruijn graph [2], and RNA transcript assembly (including isoforms) [3], we initially set out to search for a method to build haplotypes using network flow.

To recap, a network flow problem is defined as follows: given a directed graph $G = (V, E)$, with a single source s and a single sink t. Each edge $e \in E$ has a *capacity* c_e. We now search for a maximum "flow" from s to t such that the flow at each edge does not exceed its capacity, i.e. $f(e) \leq c_e$, and that for every vertex v other than s or t it holds that the flow into a vertex equals the flow leaving that vertex, or $\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$. Here, $f(e)$ represents the flow at a given edge.

The basic network flow problem can be extended to a problem with multiple sources and sinks (so called circulation problems), it is possible to put lower bounds on edges (in addition to the capacities, the upper bounds), and put costs on edges, such that putting flow on one edge can

be cheaper than putting flow on another edge. The problem then transforms to a min-cost flow problem. Network flow, circulation and min-cost flow problems can all be solved in polynomial time, which makes it an attractive model for several optimisation problems.

The approach for RNA transcript assembly in Trapnell *et al.* is backed by a min-cost flow problem [3]. They create an overlap graph between fragments and ensure this graph is acyclic. Then they solve the "minimum cost minimum path cover" problem: search for a minimum number of paths such that all vertices (fragments) belong to at least one path and that the cost of flow is minimised. Each path then represents an RNA transcript. The cost per flow on each edge represents the belief that the two linked fragments originate from the same transcript, which is calculated using a probabilistic model. The cheaper the edge, the higher the probability that these two fragments originate from the same transcript.

The acyclicity of the graph is important here. Path covering problems in a general directed graph are NP-hard [4]. For example, if it would be possible to cover all vertices with a single path then we have found a Hamiltonian path. But, a directed acyclic graph can be seen as a partial order, and using Dilworth's theorem the problem of finding a path cover becomes tractable [5, 6].

The problem of building haplotypes shows several similarities with the RNA transcript assembly problem:

- Both problems have fragments of the complete sequence that needs to be assembled;
- In both problems the fragments can be very similar, while originating from different transcripts or chromosomes;
- Both problems build a graph with vertices for each fragment and edges that denote overlaps;
- Both problems search for a set of paths through the graph such that each path represents a transcript/haplotype.

But, there are several key differences:

- RNA fragments are mapped to the original reference genome, and can be ordered by their genomic coordinates. This is used by Trapnell *et al.* to determine the direction of each edge, and ensures the constructed graph is acyclic.
- A *de novo* assembly graph is in general not acyclic, caused by repepitive regions in the genome to assemble.
- In the haplotype reconstruction problem we are searching for k paths such that each path represents a haplotype, not the minimum amount of paths.
- If considering sequencing error, the method by Trapnell *et al.* is aimed at short read data, which has a lower error error rate than long read data.

To devise a method for haplotype reconstruction using network flow we envision the following requirements:

1. A method to build acyclic subgraphs, and deal with repeats that induce cycles, otherwise the path cover problem becomes intractable;
2. A model that that can be used to calculate the belief that two reads come from the same chromosome. We think that purely looking at the overlap between two reads is not enough: you may ignore then a lot of information from longer reads that span multiple variants;
3. A method to create an assembly graph such that each path represents a DNA sequence consistent with one of the chromosomes;

4. A method that find k paths through this assembly graph such that each path is a haplotype.

Especially point (2) is not trivial, because a global look at the read data is required. If at a vertex v (a read), choosing the best edge out of v (and thus extending our current haplotype) depends on decisions made earlier, i.e. the path, or current partial haplotype, used to reach v. This is hard to capture in a network flow problem.

Early in the project we decided not to continue down this path mainly due to point (1) and (2). In hindsight, especially given the assumptions made in our current bubble chain phasing algorithm, this might still be an interested method to explore.

C. All-versus-all exact overlaps between reads

The algorithm for identifying all-versus-all exact overlaps between reads uses a generalised suffix tree as core datastructure. It is based on the algorithm for finding suffix-prefix overlaps described in the book "Genome Scale Algorithm Design" [7], but changed such that the algorithm outputs overlaps where one read is fully contained in the other too. This ensures the output is similar to the output of DALIGNER. The algorithm is summarised as follows:

1. The input is a set of reads $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$, and a parameter τ denoting the minimum overlap length.
2. Create a generalised suffix tree $\mathcal{ST}_T$ of $T = R_1\$_1R_2\$_2R_3\$_3 \ldots \$_{n-1}R_n\$_n\#$. Each separation character $\$_i$ is unique.
3. Initialise an empty stack for every read $R \in \mathcal{R}$.
4. Traverse the suffix tree $\mathcal{ST}_T$ in pre-order.
    (a) When the algorithm encounters for the first time any node v with a string depth of at least τ, then for every outgoing edge from v where the label on that edge starts with $\$_i$, we add the string depth of v on the stack of read R_i.
    (b) Symmetrically, when visiting any node v with a string depth of at least τ for the last time (travelling the tree upwards again), we pop from every stack R_i if there is an outgoing edge from v that starts with $\$_i$.
    (c) Now, when the algorithm enters a leaf node corresponding to the suffix $R_i\$_i$ (i.e. the leaf node corresponding to a whole read), then the non-empty stacks correspond to reads R_j where a *suffix* of R_j matches a *prefix* of R_i, with a length of at least τ. The top of the stack represents the longest suffix of R_j that equals a prefix of R_i.
    (d) It is easy to check whether a read R_i is contained in another read when at the leaf node $R_i\$_i$. If the edge from the parent to the current leaf node is solely labeled with its separation character $\$_i$, then R_i is contained in another read. By traversing the tree downwards from its siblings, it is possible to obtain the start and end positions of the containment. This is the same as the classic problem of searching for a pattern P in a text T, the pattern is now the contained read, and the other reads are the text T.

We have implemented an iterative version of this algorithm with the help of SeqAn [8], a C++ library for biological sequence analysis. We also take the reverse complement of reads into account.

D. Dosage probabilities for different ploidy

Ploidy / d_m =	1	2	3	4	5	6	7	8
Ploidy 2	$\frac{3}{4}$	$\frac{1}{4}$						
Ploidy 3	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{6}$					
Ploidy 4	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{8}$				
Ploidy 6	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{6}$	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{12}$		
Ploidy 8	$\frac{6}{16}$	$\frac{3}{16}$	$\frac{2}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$

E. DALIGNER settings

- Min overlap length (-l): 1000
- Average correlation rate (-e): 0.999. A value of 1.0 is not possible. Because the distance between mutations in our synthetic genomes is on average 48 bp, we expected that this wouldn't have a large influence.
- -k: 16
- -w: 1
- -h: 40

The -k, -h and -w options affect the initial filtration search. To quote from the official website:

"The options -k, -h, and -w control the initial filtration search for possible matches between reads. Specifically, our search code looks for a pair of diagonal bands of width 2^w (default 2^6 = 64) that contain a collection of exact matching k-mers (default 14) between the two reads, such that the total number of bases covered by the k-mer hits is h (default 35). k cannot be larger than 32 in the current implementation."

F. Nucmer settings

- Output all maximal exact matches, not just maximal unique matches (–maxmatch).
- Minimum length of an exact match is 7500 (-l 7500)
- Do not try to align the sequence between exact matches (–nodelta)
- Do not allow gaps between clusters of exact matches (-g 0)

G. Personal Reflection

This work marks the end of many years at Delft university of Technology. After switching between study programmes several times, I am very happy to found the field of bioinformatics. In this field I feel I can contribute to some relevant problems in the world, while using my programming skill set I have progressively built since I was young.

I have found this an incredible interesting project and had loads of fun building my own genome assembly pipeline. I am a bit disappointed the output is inaccurate because of a stupid choice

early in the project. Anyway, in the beginning we still tried to tackle this problem with sequencing error, so back then the choice for DALIGNER made more sense.

I found this project very interesting because it is a perfect combination of computer science (data structures, algorithm design, software engineering), statistics (probabilistic model formulation) and biology. This is something different than your average financial accounting software. There was never a dull moment, and I have learned and implemented quite a few algorithms: range min/max queries, superbubble identification, all-vs-all exact overlaps, and of course the phasing algorithm. I am glad this project touched upon so many computer science topics.

In May 2016 the plan was to do my thesis project at the company DSM. In hindsight, I had chosen the wrong project there. When I did the interview they presented me a few options, and I chose a project that was more geared towards systems biology rather than bioinformatics. I think DSM does good work, I like the concept of microbial cell factories, and they presented an interesting problem on this topic. But, I am a computer scientist with some molecular biology knowledge, while a biochemist with some computer science knowledge would have been a better fit for that project.

The past year I have gained a better view on where bioinformatics is able to contribute in biology research. Sequencing data is cheap and abundant nowadays, and the strengths of bioinformatics lie in the analysis of this sequencing data. This happens in fields like comparative genomics, gene expression (RNA-seq), metagenomics, etc. Data on the metabolite level is still much more expensive to generate and less publicly available, which makes any data science project much harder. My project at DSM also suffered from this, where the available data was insufficient, and I am glad I made the switch to this project when I finished the my literature review.

Doing your project in an actual research group is a of course different than at a company. I have very much enjoyed the weekly presentations from other people in the group, the paper discussions, and the colloquia. It is nice to have some recurrent method to stay up to date, because the field moves really fast. Furthermore, I have enjoyed the conferences like ECCB2016 and see how the process of sharing knowledge is organised. I have been a teaching assistent for several courses, which I have found a nice experience. Explaining bioinformatics topics forces you to take a step back and is also a good test to see if you actually understand it yourself. While reviewing tens of submitted practicals and projects can be a bit boring, you can get a lot of satisfaction from helping students, especially if you see that they are capable and really try to make something of the practical assignments or projects.

One other thing I have noticed: I think people in research tend to be a bit more individualistic by nature (that includes me), and you see that in the group too. I think the social cohesion could be improved a bit by organising a few more activities outside a research setting. I think this will also improve interaction during work hours. This is something that is usually more present in companies.

Furthermore, I hope that the field of bioinformatics develops a bigger software engineering culture the coming years. I know research is mostly about ideas, and engineering is often seen as of lesser importance, but I think proper software can be just as important in this field. I have seen many examples on how *not* to write Python code. A big collection of individual scripts full of copy-pasted code seem to me as a recipe for bugs, hard to debug mistakes and other problems, plus it hampers the reproducibility and reusability of research projects. I will not claim my code is one hundred percent clean and tidy, but I am confident enough to say it is more organised than the majority of bioinformatics software.

Anyway, “be the change you want to see” they often say, and I am very happy I can continue in this field at the Broad Institute, together with the TU Delft for my PhD-project.

Supplementary References

[1] E. W. Myers, "The fragment assembly string graph", *Bioinformatics*, vol. 21, no. SUPPL. 2, pp. 79–85, 2005.

[2] P. Medvedev and M. Brudno, "Maximum likelihood genome assembly.", *Journal of computational biology*, vol. 16, no. 8, pp. 1101–1116, 2009.

[3] C. Trapnell, B. A. Williams, G. Pertea, A. Mortazavi, G. Kwan, M. J. van Baren, S. L. Salzberg, B. J. Wold, and L. Pachter, "Transcript assembly and quantification by RNA-Seq reveals unannotated transcripts and isoform switching during cell differentiation", *Nature Biotechnology*, vol. 28, no. 5, pp. 511–515, 2010.

[4] V. Makinen, D. Belazzougui, F. Cunial, and A. I. Tomescu, *Genome-Scale Algorithm Design*. Cambridge: Cambridge University Press, 2015.

[5] R. P. Dilworth, "A Decomposition Theorem for Partially Ordered Sets", *The Annals of Mathematics*, vol. 51, no. 1, p. 161, Jan. 1950.

[6] W. Pijls and R. Potharst, "Another Note on Dilworth's Decomposition Theorem", *Journal of Discrete Mathematics*, vol. 2013, pp. 1–4, 2013.

[7] V. Makinen, D. Belazzougui, F. Cunial, and A. I. Tomescu, "Applications of the Suffix Tree", in *Genome Scale Algorithm Design*, Cambridge: Cambridge University Press, 2015, ch. 8.4.

[8] A. Döring, D. Weese, T. Rausch, and K. Reinert, "SeqAn an efficient, generic C++ library for sequence analysis.", *BMC bioinformatics*, vol. 9, p. 11, 2008.