Hypothesis-free detection of genome-changing events in pedigree sequencing

Kiran V Garimella
Green Templeton College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy

Michaelmas 2015

Dedication

Acknowledgements

Acknowledgements

Abstract

In high-diversity populations, a complete accounting of *de novo* mutations can be difficult to obtain. Most analyses involve alignment of genomic reads to a reference genome, but if the haplotypic background upon which a mutation occurs is absent, events can be easily missed (as reads have nowhere to align) and false-positives may abound (as the aligner forces the reads to align elsewhere). In this thesis, I describe methods for *de novo* mutation discovery and genotyping based on a so-called "pedigree graph" - a de Bruijn graph where all available sequencing data (trusted and untrusted data alike) is represented. I constrain genotyping efforts to locations containing "novel kmers" - sequence present in the child but absent from the parents. I then apply Dijkstra's shortest path algorithm to perform the genotyping, even in the presence of sequencing error. In simulation, this approach provides a vastly more sensitive and specific set of *de novo* variants than traditional methods.

Contents

List of Figures

List of Tables

List of Algorithms

1 Motivation

1.1 Introduction

Increasingly frequent reports of antimicrobial resistance and immune escape have lead to worries about a "post-antibiotic era": a time when common pathogens no longer respond to drugs and for which no effective vaccines exist.[2] For influenza A, a single serine to asparagine amino acid substitution (S31N) alters the properties of the M_2 ion channel,[3] interferes with the action of the adamantane class of antiviral drugs, and has reached fixation in the population.[4] In *Mycobacterium tuberculosis*, a four-year *in vitro* drug challenge experiment on nine drug-susceptible isolates from a single patient demonstrated the strain could acquire resistance to nearly all first-line and most second-line drugs through just 12 single nucleotide polymorphisms (SNPs).[5] *Staphylococcus aureus* is the most common cause of post-operative infection worldwide and is increasingly unresponsive to β-lactam treatment and drugs in the penicillin group (methicillin, dicloxacillin, nafcillin, oxacillin, etc.). Sequencing of methicillin-susceptible (MSSA) in the penicillin group and resistant (MRSA)[1] strains reveals the acquisition of resistance genes via mobile genetic elements and horizontally transferred genomic islands (e.g. the SCC*mec* cassette chromosome upon which the methicillin and other β-lactam antibiotic resistance gene, *mecA*, resides).[6]

De novo mutations (DNM), genomic variants arising anew in a sample and absent from progenitors, underlie many medically relevant pathogenic phenotypes. The examples above all involve virulence phenotypes arising from spontaneous point mutations, structural variants, or horizontal gene transfer - all mutational methods that occur outside methods of traditional reproduction. They are critical tools for pathogenic evolution and come in myriad forms.

[1] The term "methicillin resistant" is used in the literature, though it is taken to mean all drugs - including more stable variants of methicillin used in modern clinical practice

Table 1.1: Phenotypes of *P. falciparum* isolates used for genetic crosses.

	3D7	HB3	DD2	7G8	GB4	803
pyrimethamine sensitivity	-	+				
chloroquine sensitivity		+	-			
infects mosquitoes easily		+	-			
infects *Aotus nancymaae*				-	+	-
artemisinin sensitivity					+	-

1.2 *De novo mutations (DNMs) in Plasmodium falciparum*

Experimental crosses of *Plasmodium falciparum* parasites, the causative agent for the most deadly form of malaria, have enabled the discovery of a number of inherited and *de novo* virulence factors. The contrasting phenotypes for various strains of *P. falciparum* are listed in Table 1.1. For inherited factors, crossing of the pyrimethamine-resistant 3D7 and sensitive HB3 strains[7] revealed a nonsynoymous point mutation in the *dhfr-ts*[2] gene, inhibiting binding of (and thus conferring resistance to) the drug.[8] Analysis of the HB3 x DD2 cross,[9] the latter of which is resistant to chloroquine, localized the determinant to a previously undetected gene on chromosome 7, labelled *pfcrt*[3], a member of a new family of transporters. Additional investigation into differences in mosquito infection efficacy revealed down-regulation of the *pfmdv-1*[4] gene,[10,11] disruption of which results in marked reduction of mature and functional male gametocytes.

For uninherited factors, cytoadherence and antigenic properties of parasites facilitate evasion from host immune attack, and can differ substantially from the properties of their progenitors. In 2000, Freitas-Junior *et al.* showed that some 3D7 x HB3, HB3 x DD2, and HB3 x HB3 progeny harbored non-parental forms of subtelomeric *var* genes, key members of an antigenic gene family.[12] These altered forms were likely generated during mitosis by non-allelic homologous recombination[5] (NAHR) of telomeres from two different chromosomes.[13] The resulting genes are novel, functional, and never before observed by the host immune system.

These DNMs are critical tools for malaria to evade drug and immune pressure. NAHR can further diversify a pathogen's antigenic repertoire, enabling continued evasion of immunological actors. Duplications of a transporter gene may enable faster drug clearance in a parasite, thus conferring resistance. Spontaneous point mutations may alter the binding site of a drug to a receptor, thus conferring immunity.

[2] dihydrofolate reductase-thymidylate synthase
[3] *P. falciparum* chloroquine resistance transporter
[4] *P. falciparum* male gametocyte development gene 1
[5] sometimes referred to as "ectopic" (aberrant) recombination in the literature

1.2.1 *A reference-based approach for DNM discovery and genotyping*

With the advent of sequencing technologies, it is now straightforward to discover many DNMs. Long reads (~500 bp) from the first-generation sequencing technology, Sanger sequencing, can be stitched together *in silico* by considering the overlaps of sequences generated from many copies of the genome.[14] This has enabled the reconstructions of full-length genomes and subsequent gene annotations for a single representative (or "reference") individual in a population. Second-generation sequencing is leveraging economies of scale to reduce sequencing costs by several orders of magnitude.[15] The reads it produces are shorter (~100 bp) and more error-prone, but billions of them can be produced quickly. Like the long reads, the short read data can also be assembled into a new genome, albeit with more errors and gaps, owing to the difficulty of overcoming large repetitive regions with short genomic fragments.[16] Alternatively, assuming the sequence for the reference genome and a new individual are highly similar, it is far more common (and computationally more efficient) to align the reads to the reference.[17] String matching algorithms that allow mismatches, insertion, and deletions to appear in the alignments allow millions of sequnced reads to be placed on the reference genome quickly. Separate tools can then examine the alignments, looking for the presence of non-reference alleles, and using statistical approaches to call and genotype variants with high accuracy.[18]

Variant calling software has been successfully applied to many sequencing datasets for the discovery of DNMs. These variant callsets have provided insight into the development of drug resistance,[19] the genetic architecture of some common diseases,[20] and mutational rates in humans and chimpanzees with a strong paternal age effect.[21–24] Many groups have released sophisticated software packages to facilitate these analyses and detailed instructions on their use.[25,26]

1.2.2 *A reference-free approach for assessing DNM sensitivity*

Specificity and sensitivity are crucial metrics to consider for any variant callset. There are many approaches to establishing the specificity of a DNM callset. For select variants (typically on the order of ~100 variants from a callset), Sanger sequencing, Sequenom assays, and even targeted third-generation sequencing (i.e. PacBio sequencing, which can generate reads up to ~50,000 bp as of this writing) have been used successfully to validate mutations. Establishing sensitivity is much more difficult. In theory, one would need complete and error-free reference genomes for both parents and each child in which the mutation calls are made. Except for the smallest genomes, such an approach is prohibitively expensive.

Instead, it may be possible to establish the sensitivity of the reference-based protocol by considering how DNMs alter the genome of a sample with respect to its progenitors. Consider a site where a DNM - say, a single SNP - has occurred, as depicted in Figure 1.1. Although a single base of the genome has been altered, when the genome is divided into fixed-length words of length k, or "kmers", we find k kmers that are present in the child but absent in the parents. In this manner, DNMs can be considered generators of "novel" kmers - kmers present in the child but absent in the parents. These kmers can be used as a signal to indicate the presence of a *de novo* variant. By choosing k to be reasonably large so as to avoid analyzing short sequences that pervade the genome (half to two-thirds the length of a read will suffice), we can simply count continuous stretches of novel kmers as a proxy for the number of DNMs.

This approach gives us a powerful, reference-free mechanism to verify the results of the reference-based analysis. Sequencing of the whole genome is independent of any reference sequence that may already exist for the sample, and given sufficient coverage (Table 1.2 shows the requirements, assuming 76 bp reads and a 23 megabase genome), the raw data from a sequencing experiment should contain the full set of DNM-generated novel kmers, regardless of any mapping issues.[27] Taking the reads that map, calling *de novo* variants, and extracting the novel kmers from the immediate vicinity should theoretically reproduce that set. Comparing the expected (reference-free) set to the observed (reference-based) kmer set should thus provide the sought sensitivity measure.

1.3 DNM sensitivity of the reference-based approach

We now demonstrate these ideas on real data sets that will be used throughout this dissertation: experimental crosses between malaria parasites (*Plasmodium falciparum*).

The datasets that we'll be referring to in this work are summarized in Table 1.3. For simplicity, we shall focus on the samples from the 3D7xHB3 cross.

1.3.1 Data processing

We first generated the list of expected novel kmers by processing the raw sequencing data for each sample with the *de novo* assembly software, Cortex[28] at $k = 47$, standard settings for other parameters, and no error cleaning. We produced the initial list of putative novel kmers by selecting kmers present in a child but absent in both parents. We further produced a "confident" list of novel kmers by imposing a kmer coverage threshold, automatically determined by a custom algorithm designed to find a local minimum on the LOESS regression of the coverage distribution, as shown in Figure 1.2. Finally, we produce a "trusted" list of novel kmers by removing kmers originating from possible

Table 1.2: Theoretical percentage of the genome recovered at a target depth of coverage.

coverage	numReads	numNucleotides	pctGenome
1	302631	2.3e+07	63.21
2	605263	4.6e+07	86.47
3	907894	6.9e+07	95.02
4	1210526	9.2e+07	98.17
5	1513157	1.15e+08	99.33
6	1815789	1.38e+08	99.75
7	2118421	1.61e+08	99.91
8	2421052	1.84e+08	99.97
9	2723684	2.07e+08	99.99
10	3026315	2.3e+08	100.00
11	3328947	2.53e+08	100.00
12	3631578	2.76e+08	100.00
13	3934210	2.99e+08	100.00
14	4236842	3.22e+08	100.00
15	4539473	3.45e+08	100.00
16	4842105	3.68e+08	100.00
17	5144736	3.91e+08	100.00
18	5447368	4.14e+08	100.00
19	5750000	4.37e+08	100.00
20	6052631	4.6e+08	100.00

Table 1.3: The *P. falciparum* datasets that will be referred to throughout this work.

	3D7xHB3	HB3xDD2	7G8xGB4	803xGB4
progeny	16	39	41	34
read length	76	76	76	100
fragment size	304 ± 32	251 ± 50	295 ± 23	222 ± 10
coverage	96 ± 40			

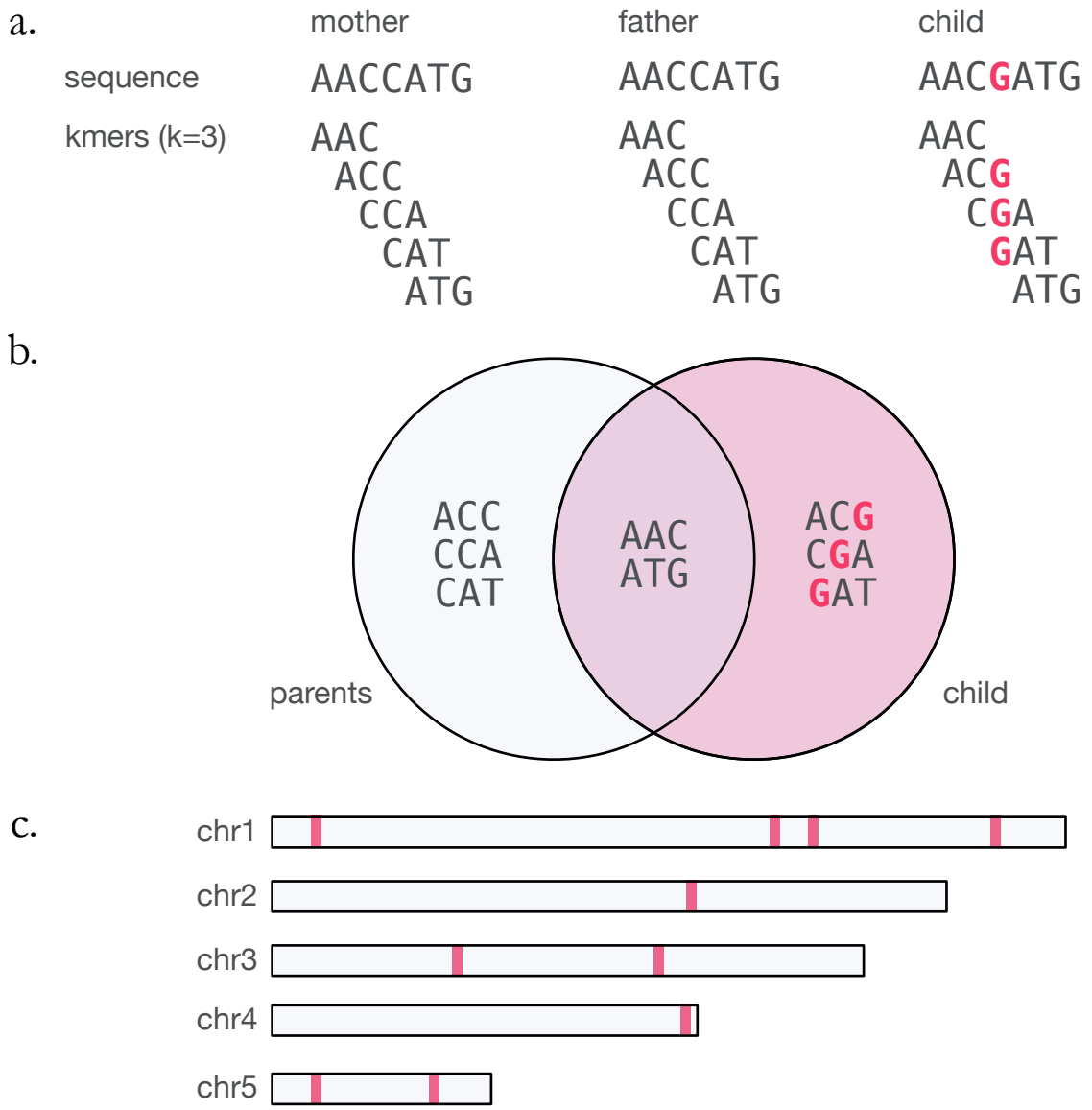

Figure 1.1: a. Parental and child sequences at the site of a *de novo* mutation, and the kmers generated at $k = 3$. b. The resulting Venn diagram of kmers found exclusively in the parents, the child, or common to both. c. Novel kmers found around the genome indicate the presence of a *de novo* mutation.

contaminants, as determined by performing a BLAST search on the confident list and removing all non-*Plasmodium* hits. These steps ensure that the list of trusted novel kmers is very restrictive.

We aligned each sample's reads to the PlasmoDB 9.0 release of the *Plasmodium falciparum* genome[29,30] using BWA-MEM.[31] We followed data processing guidelines as specified in the Genome Analysis Toolkit (GATK) best-practices documentation,[25,32] flagging

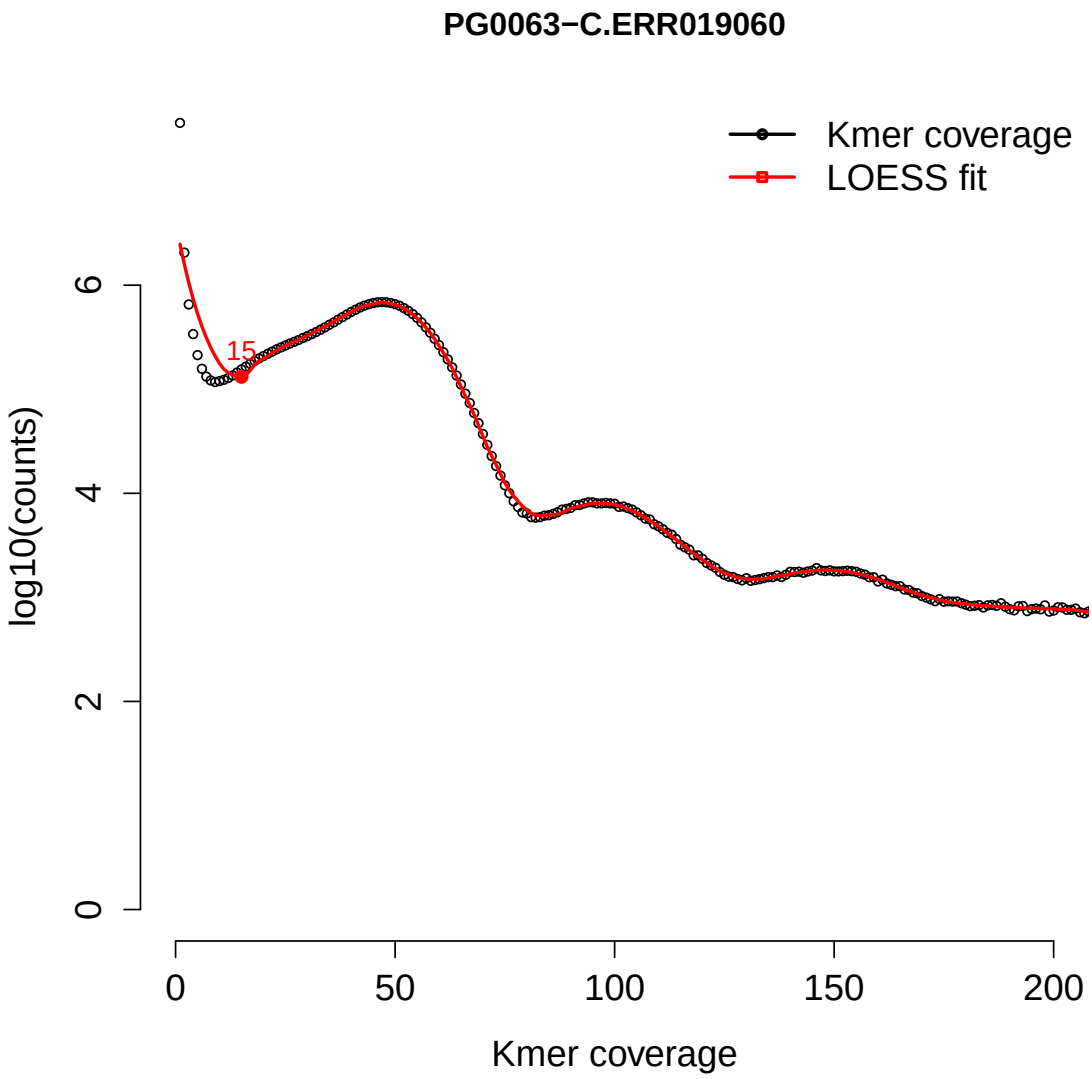

Figure 1.2: Kmer coverage distribution for a single 3D7xHB3 progeny, PG0063-C. Red line indicates non-parametric LOESS fit upon which the local minimum is detected.

duplicate reads so that they are ignored downstream, and recalibrating base quality scores using the MalariaGen data release on the crosses as a truth set.[?] As recent guidance by the authors indicates the GATK's variant calling software has internalized the local indel realignment functionality, we opted to forego running this step separately. We called variants across all 18 samples (parents and progeny) simultaneously using the GATK's HaplotypeCaller with standard settings and a ploidy of 1.

A complete and perfect variant callset details the alterations that must be performed on the reference sequence in order to generate the genome of the sequenced sample. Unfortunately, it is typically not possible to obtain a perfectly sensitive and specific callset.

False positives and false negatives proximal to true positives may interfere with our ability to generate the true underlying haplotype sequence. To bypass this problem, we did not filter the variant callset. Instead, we combinatorically generate all possible subsets of variants found within 100 bp of each other using a recursive strategy to leave single elements out of a given set of kmers, presented in Algorithm 1. This will certainly generate vastly more kmers than truly exist in the sample's genome. However, as we are only interested in verifying that the variant-induced kmers are present in our trusted novel kmer set, this approach has the benefit of providing maximum sensitivity.

Algorithm 1 Given a set of variants, generate all possible subsets of variants

```
1:  function generateAllPossibleSubsets(variants)
2:      loos ← []
3:      for i ← 0 to length(variants) do
4:          loo ← []
5:          for j ← 0 to length(variants) do
6:              if i ≠ j then
7:                  loo.add(variants[j])
8:          if loo.size() ≥ 0 then
9:              loos.add(loo)
10:         if loo.size() ≥ 1 then
11:             generateAllPossibleSubsets(loo)
12:     return loos
```

1.3.2 *Results*

Figure 1.3 shows the results of our reference-based vs reference-free analysis. Per sample, our restrictive reference-free analysis has generated hundreds of trusted novel kmers to explain. However, the reference-based analysis recapitulates only a fraction of these - only about 13% on average.

Attempting to explain where these missing kmers have gone, we searched the reads for every trusted novel kmer. More than 80% of reads containing these kmers were found to map to multiple homes in the genome (summarized per sample in Figure 1.4).

1.4 *Discussion*

1.4.1 *Failure of the mapping approach*

The preponderance of reads containing novel kmers fail to map uniquely to the reference genome, explaining why we should find such a massive discrepancy between the novel kmers we expect versus explain - reference-based calling approaches cannot call variants

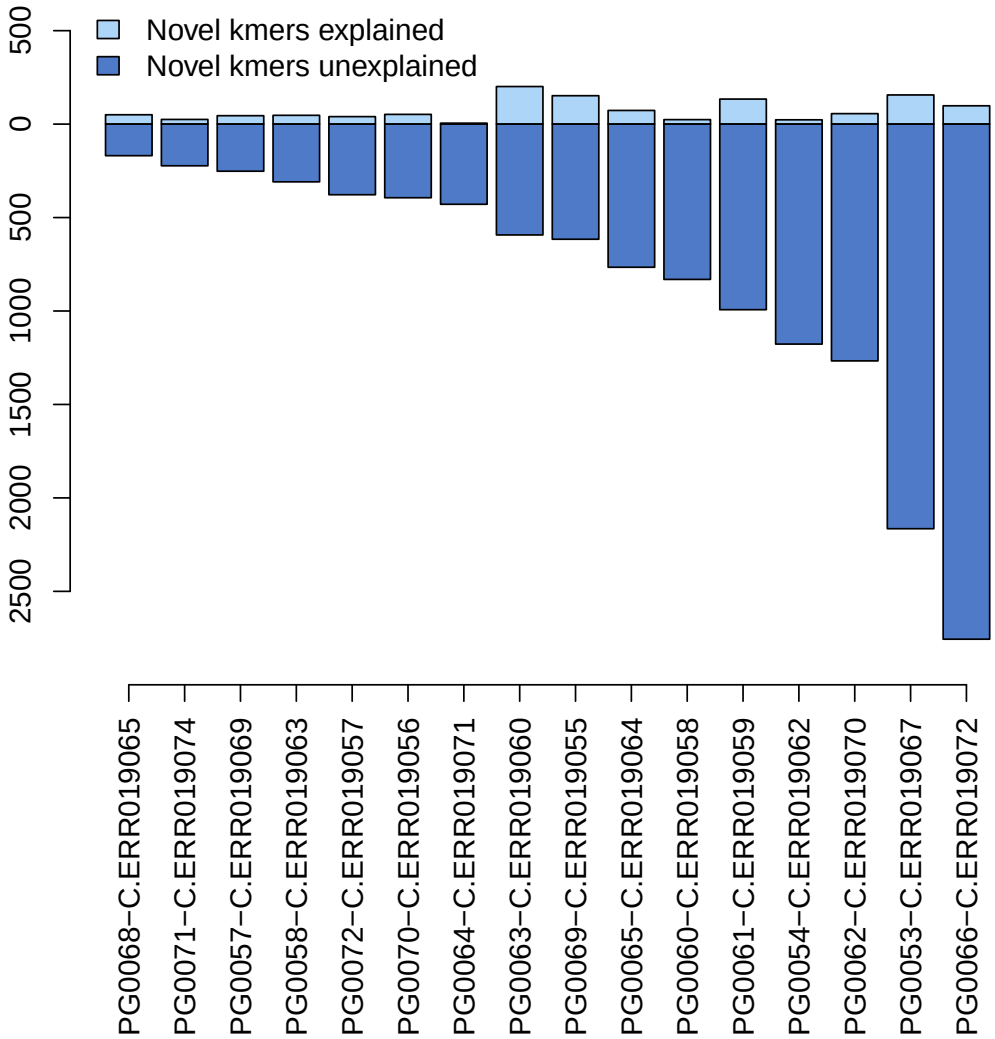

Figure 1.3: Novel kmers observed in the reference-based analysis vs trusted novel kmers expected from the reference-free analysis.

on unplaced reads. That there should be so many reads that fail to map is perhaps not surprising, given the divergent nature of the reference genome to other samples. Figure 1.5 shows the overlap between kmer sets between the 3D7 and HB3 genomes. More than 20% of the total set of kmers between these two samples is unique to each sample.

These unique kmers are not simply repetitive, intergenic, or otherwise uninteresting kmers. They often reflect interesting biology. Figure 1.6 shows one example: three *var* genes from 3D7's 60-member antigenic repertoire. These genes do not overlap with the HB3 repertoire. One of the 3D7xHB3 progeny, has inherited one of the 3D7 *var* genes in full, but curiously exhibits mosaic recovery of two others. This is known to be an NAHR

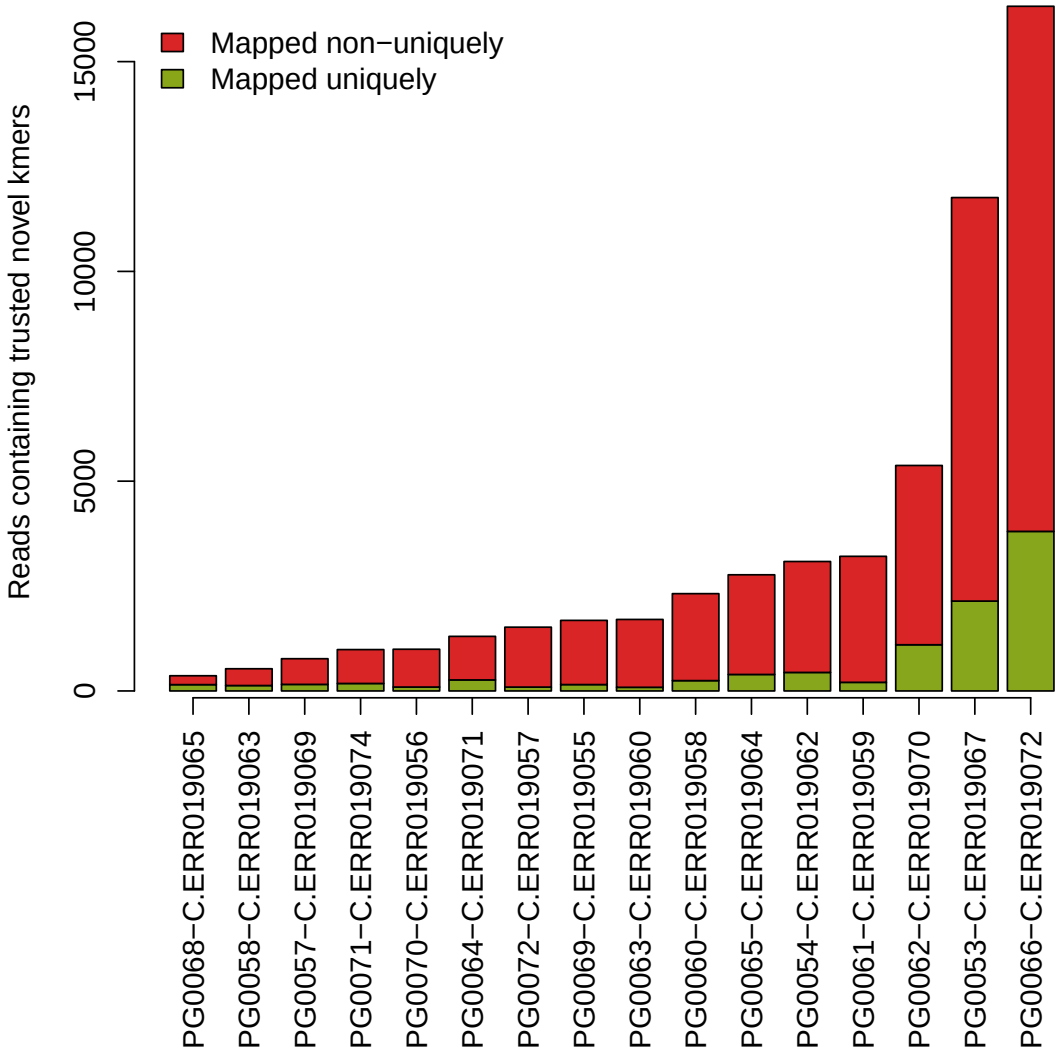

Figure 1.4: Reads that map once to the reference genome versus mapping multiple times, conditioned on the read containing a trusted novel kmer.

event between the telomeres of chromosomes 1 and 2, likely to have occured during mitosis, preserving the domain architecture and yielding a functional product.[33]

Alignment of short reads to a single reference and subsequent application of variant callers is a poor strategy for *de novo* variant discovery in *P. falciparum* (and likely higher-order species as well) for a number of reasons:

1. **Absent or divergent loci in genome**

   The underlying assumption that two genomes from the same species should be very similar is inappropriate for highly diverse populations (e.g. pathogens) or

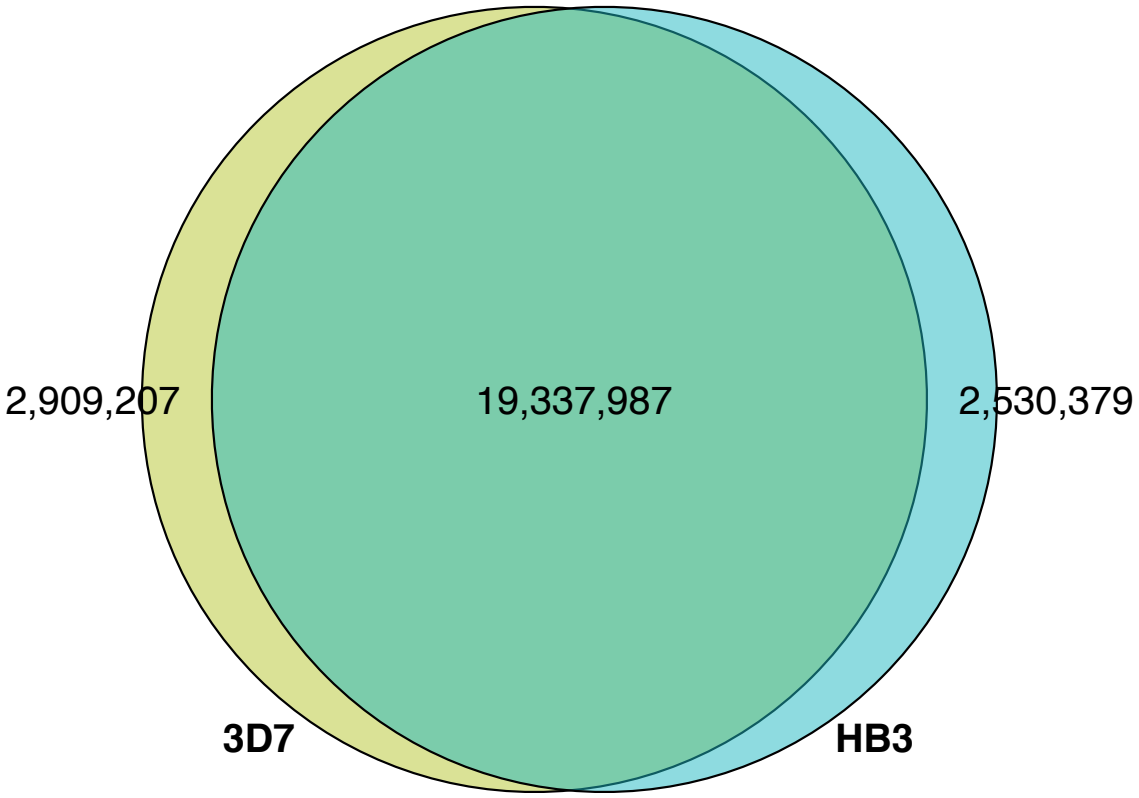

Figure 1.5: Venn diagram of kmers present in the 3D7 and HB3 genomes at $k = 47$. Both forward and reverse-compliment kmers are considered the same.

hypervariable regions (e.g. immune loci in mammals). If a haplotype present in the sample is too dissimilar to the reference, or perhaps even completely absent, read aligners may return incorrect results. Reads may align to the wrong location, the resultant mismatches mistaken for real mutations. Alternatively, they may fail to align at all, thus obscuring evidence of real variation.

2. **Incomplete or errorful reference sequences**
   Reference sequences are often incomplete and/or contain errors due to technical artifacts. Chaisson *et al.* provide an excellent review on the various errors that may arise.[34] Regions of the genome may fail to amplify during library preparation, leading to coverage dropout and subsequent gaps in the assembly. Insufficiently long reads used in the reference genome construction may lead to the misestimation of lengths of repetitive regions, causing repeats to have a collapsed representation relative to the true genome. Segmental duplications, gene families, or other loci with high sequence identity may generate ambiguous read overlaps that cannot be resolved without very long reads.

3. **Difficult to include prior information about variation in a species**
   Read aligners must make a decision as to how many apparent mismatches to permit with respect to the reference sequence. However, these software packages typically operate per-read, without information on prior population variation throughout the genome.

4. **Inability to include improved or project-specific data**
   Improvements in sequencing technology, or new platforms altogether, can yield supplementary datasets that adds missing information to a genome or repairs a misrepresented locus. There is no natural framework for incorporating these additional datasets to the alignment framework.

1.4.2 De novo assembly as an alternative approach

Consider Table 1.2 again, which demonstrates that we can expect to recover the full genome at as little as 10-fold coverage. For small genomes (*P. falciparum* is approximately 23 megabases in length), modern sequencing experiments can routinely return excess of 100-fold coverage. It is therefore clear that deeply-sequenced samples will have reads representing the entirety of the genome despite our inability to align all of them to the genome. Rather than relying on mapping to an imperfect and incomplete reference, we can attempt to assemble the genome *de novo* - from the sequenced data itself, ignoring the availability of a reference sequence.

The problem of performing *de novo* assembly essentially reduces to computing read-to-read alignments, rather than read-to-reference alignments. As each read represents recovery of some small region of the genome, the supersequence of overlapping reads (the aligned nucleotides flanked by the non-overlapping sequences from each read) represent some larger linear stretch of the genome. Brute-force computation of all possible pairwise alignments is $\mathcal{O}(N^2)$ in the number of reads, which is impractical for second-generation sequencing datasets with tens or hundreds of millions of reads. Fortunately, there are many ways to compute and represent these overlaps efficiently. We shall focus on one $\mathcal{O}(N)$ method in this work: assembly via construction of a de Bruijn graph.

Formal definitions can be found below. Informally, a graph is simply a data structure representing a collection of objects (termed "vertices" or "nodes") and connections ("edges" or "links"). In a de Bruijn graph, each vertex is a unique element. Applied to sequencing, a de Bruijn graph encodes linear stretches of sequence, while each edge represents an overlap with the connected vertices. Commonly, each read is decomposed into fixed-length words of an arbitrary length k, or "kmers". As each kmer is sequentially extracted

from a read and added to the graph as vertices, edges between adjacent kmers in the read are stored as well. Overlapping reads will share kmers, and since each kmer can only appear once in the graph, adding kmers and edges from the overlapping read effectively records the alignment without needing to literally compute all possibilities. Figure 1.7 depicts a simple 16 bp genome, sequenced with 7 bp reads, and the resulting de Bruijn graph constructed at a kmer size of 3.

Construction of this data structure is not without its challenges. First, errors in second-generation sequencing data are very common, and therefore the graph produced from raw sequencing data will contain many branches that are not in the actual genome (examine Figure 1.7 again, observing the highlighted base - a sequencing error - and the resultant perturbation to the otherwise linear graph). These can be mitigated (but perhaps not completely solved) by error-cleaning algorithms that remove low coverage kmers, as presumably in high coverage data, random errors are rare and can be detected and discarded. Second, long repetitive stretches of the genome that feature the same kmers multiple times will be collapsed into a single copy, as de Bruijn graphs only store each kmer once. Finally, homology in the genome can cause two separate regions of the genome to appear proximal to one another in the graph. This may result in a vertices with multiple outgoing edges, causing unresolvable ambiguity when traversing the graph.

Nevertheless, such an approach should resolve many of the deficiences of an alignment-based approach. Absent or divergent loci should be recovered. The uncleaned graph should be complete (barring any regions of the genome that suffer from an amplification bias that prevents them from being sequenced). Prior information about variation in the species (or in this case, just the parents) are included by assembling the parents and comparing to them directly, rather than indirectly via the reference. Finally, additional information can be added at graph construction time or by adding a separate color to the graph encoding the supplementary data.

1.4.3 Formal definitions

We denote a **graph** as $G = \{\mathcal{V}, \mathcal{E}\}$ where $\mathcal{V}$ represents a unique set of **vertices**, and $\mathcal{E}$ a set of **edges**.[35] We shall assume the set of vertices $\mathcal{V} = V_1, V_2, ..., V_n$. A pair of vertices may be connected by an edge. For the purposes of this dissertation, we shall require that all edges are **directed**, though generally, it is also possible for edges to be **undirected**. Written as $V_i \rightarrow V_j$ or equivalently $V_j \rightarrow V_i$, directed edges restrict graph traversal in one direction (thus enforcing a traversal order when reconstructing a region of the genome). A vertex may have a number of incoming (outgoing) edges, the precise count being referred to

as a vertex's **in- (out-) degree**. We shall also refer to any vertex with an in-degree or out-degree greater than 1 as a **junction**.

A **path** is a sequence of adjacent vertices that respects these edge relationships, i.e. $V_i, ..., V_k$ such that for every $j = i, ..., k$, we have $V_j \rightarrow V_{j+1}$. In Figure 1.8, the vertex sequence A, B, C, F forms a path. Similarly, a **trail** denotes a sequence of adjacent vertices, but does not respect the directionality of the edges. In Figure 1.8, the sequence F, E, D, A forms a trail. Edges can optionally carry **weight**, indicating an arbitrary cost for traversing from one vertex to another via particular edges.

A **multi-color graph** is a useful extension for multi-sample scenarios. Each edge carries additional metadata used to denote sample identity (each sample is assigned a unique "color"). All kmers in all samples are added to the graph, and following the edges with the same color yields the genome of that sample. Many kmers will not be accessible when traversing paths or trails in one color versus another. These motifs are the hallmark of most variation in a graphical setting.

Often in this dissertation, we will perform operations on a limited graph region wherein we process only vertices of interest and all edges present between them in the original graph. This is referred to as an **induced subgraph** (which is *not* technically synonymous with a subgraph, but for simplicity in this work, we will often drop the "induced" qualifier). Let $\mathbf{V} \subset \mathcal{V}$. The **subgraph** $G[\mathbf{V}] = \{\mathbf{V}, \mathcal{E}'\}$ where $\mathcal{E}'$ are all the edges $V_i \rightleftharpoons V_j \in \mathcal{E}$ such that $V_i, V_j \in \mathbf{V}$.

1.5 *Overview of this work*

In this dissertation, we present a novel multi-color graph-based approach to *de novo* mutation detection and allele identification. We show how knowledge of the pedigree enables us to identify so-called **novel kmers** - kmers present in children and absent from parents - that serve as an exceedingly strong signal as to the presence of *de novo* variation. We use these kmers to analyze subgraphs in the genome likely to represent DNMs and navigate color-specific paths and trails to determine the precise allele. We also demonstrate how the novel kmers act as "sign posts" during graph traversal, indicating that a traversal is following a fruitful path. This simultaneously constrains th runtime of the algorithm and allows us to overcome sequencing error that could not otherwise be overcome. We demonstrate that this approaches yields vastly superior sensitivity and specificity to DNMs than conventional methods, and can easily access events that occur on the haplotypic background of the non-reference parent.

In Chapter 2, I present a review on mutational mechanisms that generate *de novo* mutations, their rates, factors that influence their generation, and known events in various species.

Chapters 3 and 4 detail the software packages I have written for this work, the former including descriptions of the realistic variant read simulations, the latter detailing the graph genotyping algorithm.

Chapters 5 and 6 present results from applying the algorithm to real data. The former chapter focuses on the aforementioned *P. falciparum* crosses. The latter addresses a *Pan troglodytes* pedigree.

Finally, Chapter 7 discusses the work in a larger context and details various improvements that can be made in future work.

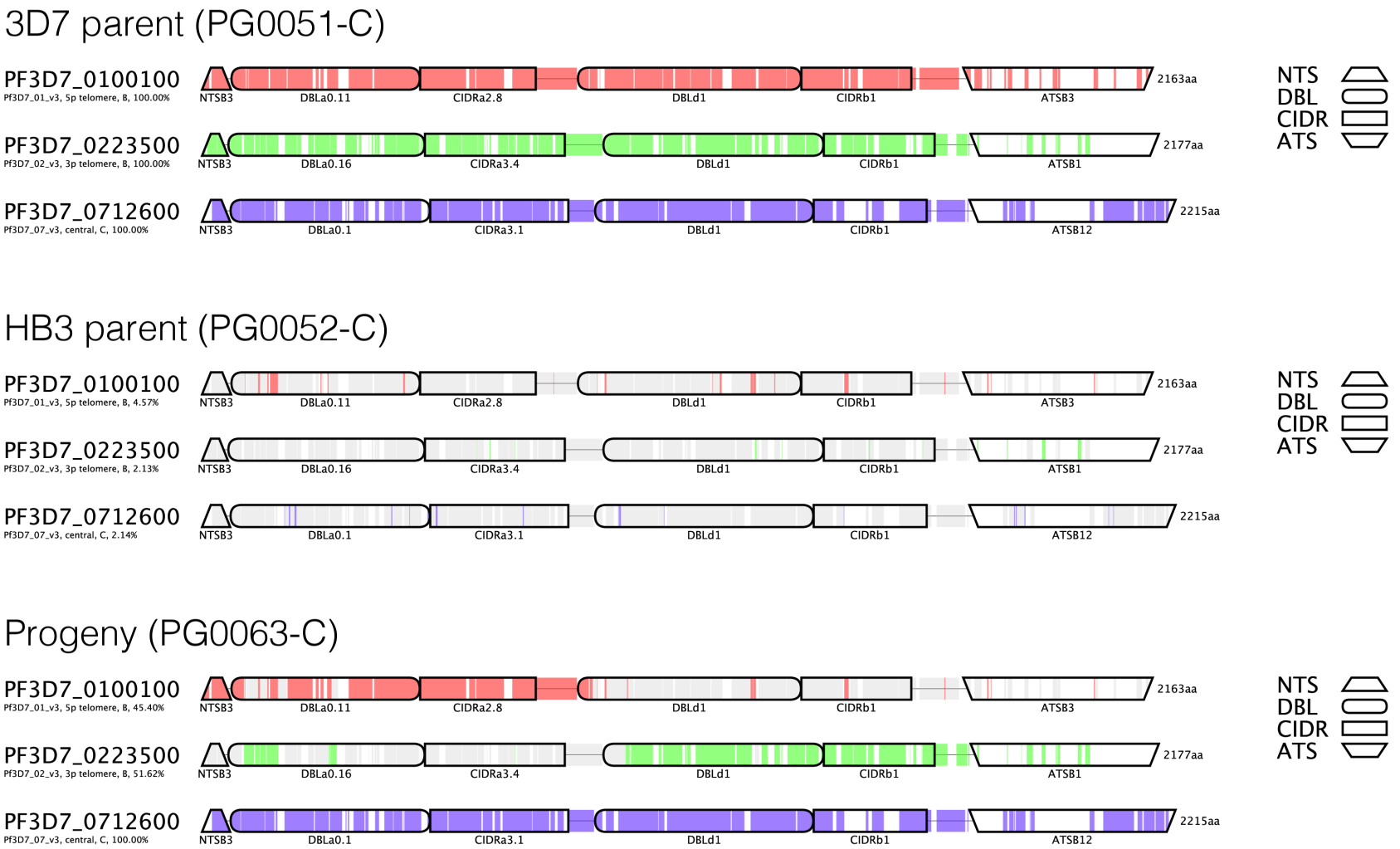

Figure 1.6: Presence and absence of unique kmers in three 3D7 *var* genes. Each vertical line represents a kmer. Colored kmers represent those unique 3D7 kmers that are recovered in the sample. Grey indicates no recovery. White indicates the kmer at that position was not unique in the 3D7 genome. Only the coding regions of the respective genes are shown, with domain annotations obtained from the VarDom server.[1]

a. Genome to be sequenced

```
ACCATAGCTTTAATTA
```

b. Sequenced reads and resulting kmers (k=3)

```
ACCATAG (ACC, CCA, CAT, ATA, TAG)
TAGCTTT (TAG, AGC, GCT, CTT, TTT)
TTAATTA (TTA, TAA, AAT, ATT, TTA)
ATAGCTT (ATA, TAG, AGC, GCT, CTT)
CATAGCT (CAT, ATA, TAG, AGC, GCT)
AGCGTTA (AGC, GCG, CGT, GTT, TTA)
GCTTTAA (GCT, CTT, TTT, TTA, TAA)
```

c. de Bruijn graph reconstruction (k=3)

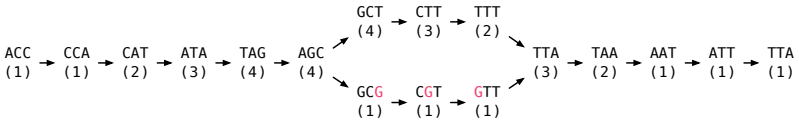

Figure 1.7: The process of generating a de Bruijn graph representation of sequence data. a. The underlying genome. b. Reads sequenced from the genome (including one read with a sequencing error). Reverse-complement reads are not shown for clarity. c. The $k = 3$ de Bruijn graph reconstruction, including kmer coverage annotations.

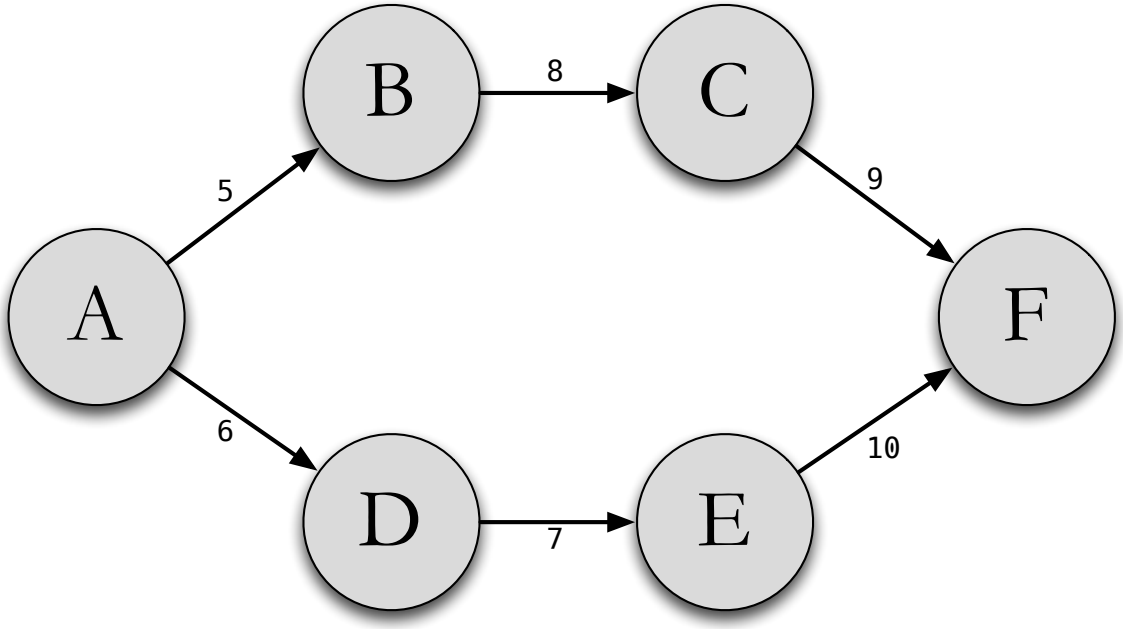

Figure 1.8: An example directed graph with six vertices.

2 Background

2.1 How genome changes

2.1.1 Cross-over

2.1.2 Gene conversion

2.1.3 Point mutations

2.1.4 Structural variants

2.1.4.1 Small (indels)

2.1.4.2 Large (fusions, NAHR)

2.1.4.3 Chromosomal changes

2.2 Rates

2.3 Factors influencing

2.3.1 Replication time

2.3.2 Mat/pat age effects

2.3.3 Biases/locality

2.4 Known events in species

2.4.1 P.f.

2.4.2 Human

2.4.3 Chimp

2.4.4 Others

3 Simulation

De novo MUTATIONS WILL UNDOUBTEDLY TAKE MYRIAD FORMS (SNPs, insertions and deletions of varying length, expansions and contractions of short tandem repeats, tandem duplications, non-allelic homologous recombinations, and possibly even inversions). Detecting all types of variants is a considerable challenge, and the software to do so will be introduced in the next chapter. In order to measure that software's expected sensitivity and specificity to such variation, we require truth datasets to which we can compare our calls. A sufficiently small genome (on the order of tens of megabases), could be run on third-generation sequencers, the long reads used to assemble full-length genomes. Then, the variants called from short-read second-generation sequencing data could be compared to the "truth" dataset established by the newer platform. However, this is still very expensive (thousands of dollars for each sample), which limits the number of samples that could be feasibly obtained. Furthermore, if variants of the classes we are attempting to identify are absent in the handful of genomes we can afford to sequence, we would not be able to accurately ascertain our power.

We chose instead to pursue a simulation strategy in order to measure our DNM calling performance[1]. There are two components to these simulations: first we must generate the genomes of the parents and several children, including each type of DNM we hope to discover. From these genomes, we must then simulate reads that realistically model errors inherent in our data (matching read lengths, fragment size distributions, single base mismatch errors, indel errors, read pair chimeras, coverage profile, etc.). We discuss both of these components below.

3.1 Simulating genomes

Simulating a genome merely involves generating an artificial reference sequence in FASTA format. Our framework is a simple forward simulation of samples. We first generate the

[1]Several months after the simulation framework was completed, we obtained PacBio sequencing data on the parents of the 803xGB4 cross and three randomly selected progeny. These results will be discussed in a later chapter.

genomes of the parents. To generate a child's initial genome, we perform recombination *in silico*. We then add *de novo* mutations on this substrate, thus producing the child's final genome.

We make use of the Variant Call Format (VCF),[36] a text file that encodes one variant per line, specifying the genomic locus, reference and alternate alleles, and metadata for the variant, to describe differences between the two parents and the DNMs to incorporate into the child's genome. While the `FastaAlternateReferenceMaker` module in the GATK does purport to generate a new reference sequence based on variants in a VCF, we note that at the time of this writing, it silently fails to incorporate multinucleotide polymorphisms (MNPs) (simultaneous deletion and insertion). We generate many of these events to remove a reference allele and add an alternate allele in its place (e.g. inversions or gene repertoire replacements). To include this critical functionality, we developed our own tool to permute an existing reference sequence based on a single-sample VCF file.

Our algorithm, `IncorporateVariantsIntoGenome`, is described in Algorithm 2. Briefly, the sequence of each chromosome is loaded into an array, one nucleotide per array element. We then iterate over each record in the VCF file. For each SNP or insertion, we replace the reference nucleotide at that position with the entire alternate allele (for insertions, more than a single nucleotide). For deletions, we replace each corresponding array elements with empty strings. To generate the new genome, we iterate through each element of the array, emitting the string found in each position.

Note that we chose not to process each variant iteratively as insertions (deletions) would increase (decrease) the size of the array, altering the mapping between the VCF positions and the array positions. Keeping track of the mapping in spite of the changes is cumbersome. Instead, out scheme of placing all of the variants on the chromosome array first and then emitting the resulting sequence preserves the mapping. We will revisit this strategy later on in this chapter when we introduce an algorithm to lift data over from the reference genome coordinates to a simulated genome's coordinates.

Algorithm 2 could fail to produce a correct FASTA file in the pathological case that there are multiple overlapping variants called at a single locus. We are careful to avoid that scenario; we set our simulated variants to be placed no closer than 1000 bp from one another.

Many algorithms presented in this chapter rely on empirical distributions to model cross-over rates, read fragment size, indel lengths, and positional errors in reads. In all cases, we use the inverse transform sampling method for pseudo-random number generation from an arbitrary probability distribution given its cumulative distribution function

Algorithm 2 Generate an alternative reference sequence based on a VCF file.

```
1: function IncorporateVariantsIntoGenome(ref, vcf)
2:     for all chr in ref do
3:         vcs ← vcf.getVariants(chr)
4:         for all vc in vcs do
5:             if vc.getType() == DEL || vc.getType() == MNP then
6:                 for pos in vc.getPosition() : (vc.getPosition() + vc.getReferenceAllele().length()) do
7:                     chr[pos] = ""
8:             chr[vc.getPosition()] = vc.getAlternateAllele()
9:         write(chr)
```

Table 3.1: Assembly statistics on publicly available finished and draft *P. falciparum* references, ordered by scaffold N50 length. Parental samples are shown in boldface.

Isolate	Origin	Length (Mb)	Scaffolds	Scaffolds N50 (Kb)	%Q40
3D7	**Unknown**	**23.30**	**16**	**1,690.00**	**-**
HB3	**Honduras**	**24.26**	**1,189**	**96.47**	**93.17**
IGH-CR14	India	21.74	849	37.02	95.49
DD2	**Indochina/Laos**	**20.88**	**2,837**	**19.11**	**85.66**
RAJ116	India	14.11	1,199	13.00	89.68
VS/1	Vietnam	18.89	5,856	4.42	74.79
7G8	**Brazil**	**14.28**	**4,843**	**3.87**	**71.00**
Senegal_V34.04	Senegal	13.24	4,329	3.76	76.22
D10	PNG	13.38	4,471	3.71	71.80
RO-33	Ghana	13.71	4,991	3.47	69.91
K1	Thailand	13.29	4,772	3.42	73.30
FCC-2/Hainan	China	12.96	4,956	3.30	69.39
D6	Sierra Leone	13.22	5,011	3.23	71.62
SL	El Salvador	13.19	5,193	3.08	69.41
PFCLIN	Ghana	42.19	18,711	2.99	-

(CDF).[37] Simply put, we compute the CDF for an empirical probability distribution, generate a random uniform deviate between 0 and 1 for the x value, and interpolate the y value from the CDF.

3.1.1 Parents

We began by simulating the genomes of two parents using a workflow depicted in Figure 3.1. For convenience, we simulated samples from the 3D7xHB3 cross. Choosing the existing reference sequence for the 3D7 sample obviates the need to generate any data for the first parent. The second parent is tricker; while an existing draft reference sequences does exist for HB3, it is of low quality, assembled into thousands of pieces rather than

the simple 14 autosomes we expect (Table 3.1 shows metrics on every *P. falciparum* sample publicly available). Using the supercontigs from draft reference sequences is hugely cumbersome for simulating recombination as it is not straightforward to decide which chromosomes in the 3D7 genome and which supercontigs should be processed together.

Instead, we chose to produce a new HB3 reference genome sequence by taking the 3D7 reference and inserting the appropriate modifications using Algorithm 2. These modifications are comprised of two parts: introducing the appropriate variants, and replacing the *var* gene repertoire.

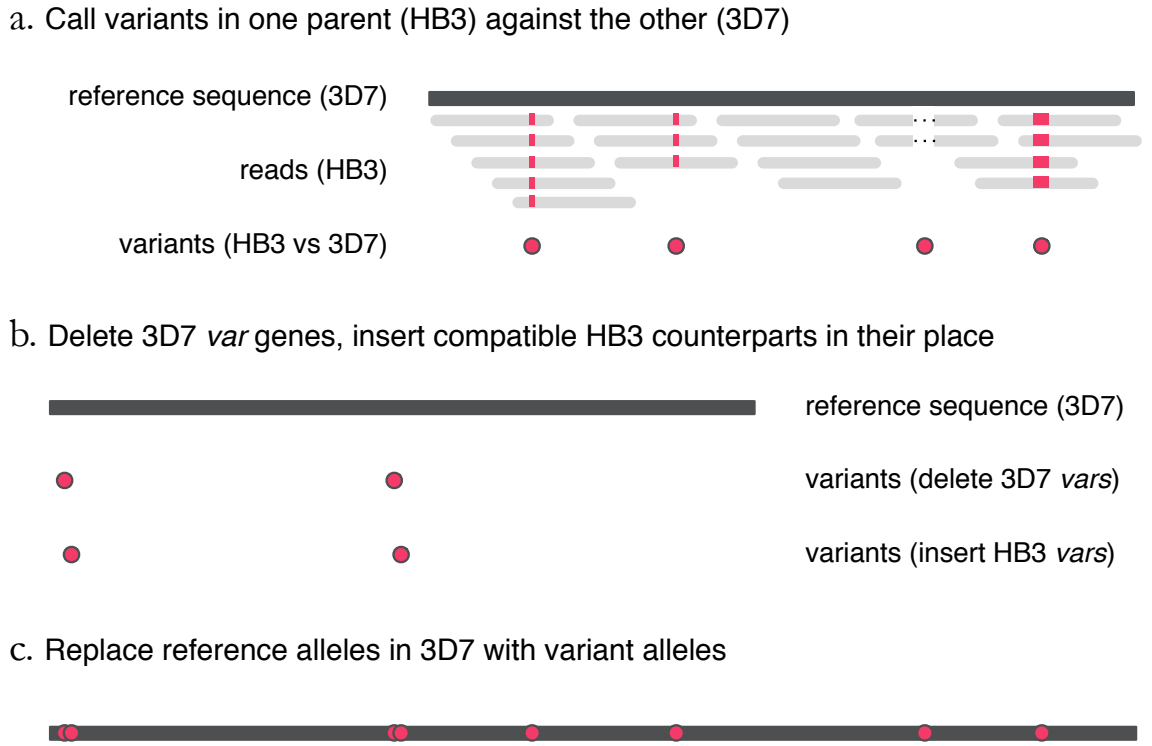

Figure 3.1: Workflow for generating the HB3 parental genome. a. Reads from HB3 sample, PG0052-C, are mapped to the 3D7 reference genome, and variants (SNPs and indels) are called and stored as a VCF file. b. We remove the 3D7 *var* gene repertoire, replacing each with a reasonable HB3 *var* counterpart, and encode the changes to the 3D7 reference genome as a VCF file. c. We alter the reference genome using Algorithm 2, thus producing the simulated HB3 genome.

We first obtained a VCF of variants found in the HB3 sample, PG0052-C, from the MalariaGen 3D7xHB3 cross dataset.[38] Variant counts are described in the Table 3.2, and include SNPs, insertions, deletions, and complex (simultaneous insertions and deletions) events. These calls were made by combining the results of the reference-based `UnifiedGenotyper` module in the GATK[25] and the reference-free bubble-calling algorithm in the Cortex[28] software. All calls were restricted to the core genome; subtelomeric regions

Table 3.2: Variants found the HB3 (PG0052-C) sample from the MalariaGen 3D7xHB3 dataset.

variants	SNPs	insertions	deletions	complex
42,054	15,376	11,807	14,643	228

were masked out due to poor mapping properties (owing to the tremendous diversity in these regions of other *P. falciparum* parasites with respect to the 3D7 reference).

Next, we produced a VCF describing *var* gene replacements. The sequences for HB3 *var* genes and upstream promoter metadata were obtained from the VarDom server.[39] No positional information from this data source is available, thus the exact placement of these *var* genes in a chromosomal context is unclear. However, previous work has established a strong association between conserved sequences of upstream promoters (phylogenetically grouped into five classes: A through E) and placement in the genome.[40] We therefore replaced 3D7 *var* genes with HB3 *var* gene counterparts, taking care to replace genes with similar UPS classes whenever possible, and grouping genes with suspiciously incomplete metadata otherwise. The replacements were described in the resulting VCF as simultaneous deletions of the 3D7 allele and insertions of the HB3 allele. No effort was made to match the orientation of the replacement *var* gene with the replaced *var* gene. The precise replacements are summarized in Table 3.3.

These two VCFs were combined to produce a complete set of changes required to transform the 3D7 genome into a pseudo HB3 genome. The transformation was applied using Algorithm 2.

3.2 *Children*

Generating the genomes of the children is slightly more involved, as there is much more biology to simulate, and many more considerations to be made when placing variants. We generated VCF descriptions of the children using a multi-stage workflow shown in Figure 3.2. In order, we simulate:

1. homologous recombination between 3D7 and HB3 genomes
2. gene conversion events
3. NAHR events between compatible *var* genes
4. *de novo* SNPs, insertions, deletions, and inversions

Table 3.3: *Var* gene replacements from 3D7 to HB3 repertoire.

3D7 gene	3D7 ups class	HB3 gene	HB3 ups class
PF3D7_0421100	UPSB5	PFHG_02500	UNKNOWN
PF3D7_0600200	UPSB2	PFHG_02495	UNKNOWN
PF3D7_0632500	UPSB5	PFHG_05132	UNKNOWN
PF3D7_0800300	UPSB2	PFHG_04012	ND
PF3D7_1200400	UPSB5	PFHG_05200	ND
PF3D7_1240900	U	PFHG_05483	ND
PF3D7_0400400	UPSA1	PFHG_03840	UPSA1
PF3D7_0425800	UPSA1	PFHG_03671	UPSA1
PF3D7_1100200	UPSA1	PFHG_04861	UPSA1
PF3D7_1150400	UPSA1	PFHG_05052	UPSA1
PF3D7_1300300	UPSA1	PFHG_03234	UPSA1
PF3D7_0533100	UPSA2	PFHG_03521	UPSA2*
PF3D7_0100300	UPSA3	PFHG_02274	UPSA3
PF3D7_0100100	UPSB1	PFHG_04081	UPSB1
PF3D7_0115700	UPSB1	PFHG_04749	UPSB1
PF3D7_0200100	UPSB1	PFHG_03516	UPSB1
PF3D7_0223500	UPSB1	PFHG_04277	UPSB1
PF3D7_0300100	UPSB1	PFHG_03717	UPSB1
PF3D7_0324900	UPSB1	PFHG_04035	UPSB1
PF3D7_0400100	UPSB1	PFHG_04491	UPSB1
PF3D7_0426000	UPSB1	PFHG_04620	UPSB1
PF3D7_0500100	UPSB1	PFHG_04593	UPSB1
PF3D7_0632800	UPSB1	PFHG_04770	UPSB1
PF3D7_0700100	UPSB1	PFHG_04057	UPSB1
PF3D7_0712300	UPSB1	PFHG_03232	UPSB1
PF3D7_0733000	UPSB1	PFHG_04928	UPSB1
PF3D7_0800100	UPSB1	PFHG_03416	UPSB1
PF3D7_0413100	UPSB3	PFHG_03476	UPSB3*
PF3D7_0712400	UPSB3	PFHG_02421	UPSB3
PF3D7_1240300	UPSB4	PFHG_02272	UPSB4
PF3D7_0809100	UPSB6	PFHG_02276	UPSB6
PF3D7_0712800	UPSB7	PFHG_04014	UPSB7
PF3D7_0808700	UPSB7	PFHG_02425	UPSB7
PF3D7_1240400	UPSB7	PFHG_04769	UPSB7
PF3D7_0412400	UPSC1	PFHG_03480	UPSC1
PF3D7_0412700	UPSC1	PFHG_03478	UPSC1
PF3D7_0412900	UPSC1	PFHG_00592	UPSC1
PF3D7_0420700	UPSC1	PFHG_02419	UPSC1
PF3D7_0420900	UPSC1	PFHG_02429	UPSC1
PF3D7_0421300	UPSC1	PFHG_02277	UPSC1
PF3D7_0617400	UPSC1	PFHG_02273	UPSC1
PF3D7_0712900	UPSC2	PFHG_04015	UPSC2
PF3D7_1200600	UPSE	PFHG_05046	UPSE

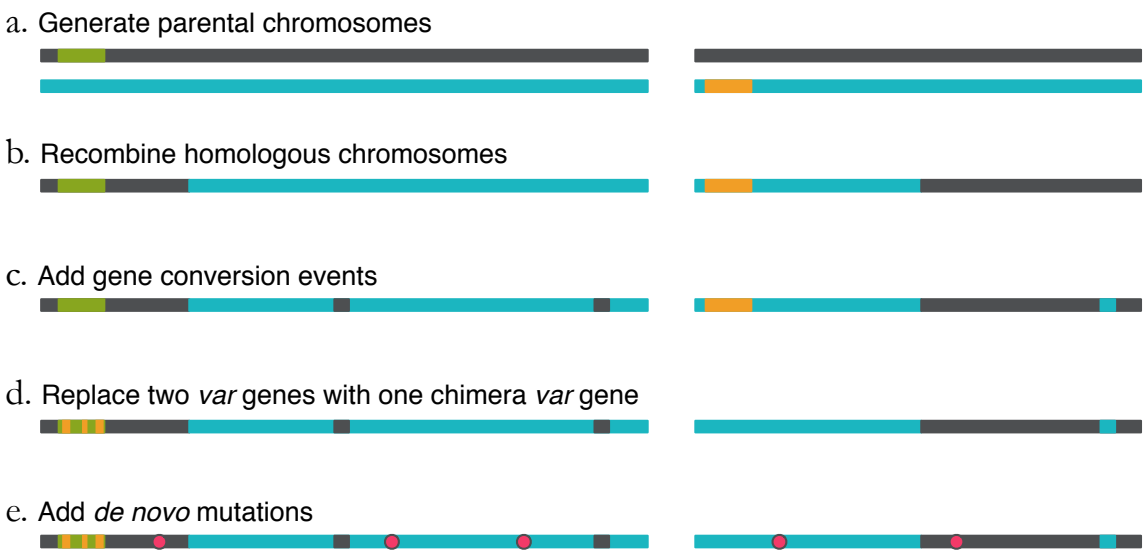

Figure 3.2: Workflow for generating children's genomes. a. Generate chromosomes from the parental genomes (compatible *var* genes shown in green and orange). b. Recombine homologous chromosomes. c. Add gene conversion events (by incorporating variants from the alternative haplotypic background over a limited genomic window). d. Replace one of the *var* genes with a chimera of compatible genes. e. Add *de novo* mutations.

3.2.1 Homologous recombination

3.2.1.1 Allelic homologous recombination

For each bivalent chromosome, the cross-over rate will be dependent on the length of the chromosome. The empirical distributions for bivalent formation and crossover were generated from the 75 samples in the MalariaGen crosses data. For each sample, only half of the chromosomes are expected to exhibit cross-over events. The cross-over rates are plotted in Figure 3.3. We simulated recombination in a sample by first drawing a binary number indicating whether a chromosome should be recombined, and if so, drawing the number of cross-over events per chromosome from these empirical distributions. The recombination sites themselves were chosen by drawing a uniform random variate between 1 and the length of the chromosome. Although there are certainly hotspots and coldspots of recombination in the genome, we have ignored this complication. An example haplotype mosaic of chromosome 12 for five samples is shown in Figure 3.4.

3.2.1.2 Gene conversion

To simulate gene conversion events, we first chose a handful of random sites known to be variant in HB3, and determined the size of the event (number of adjacent HB3 variants involved in the gene conversion) by choosing a random uniform variate between 1 and 3.

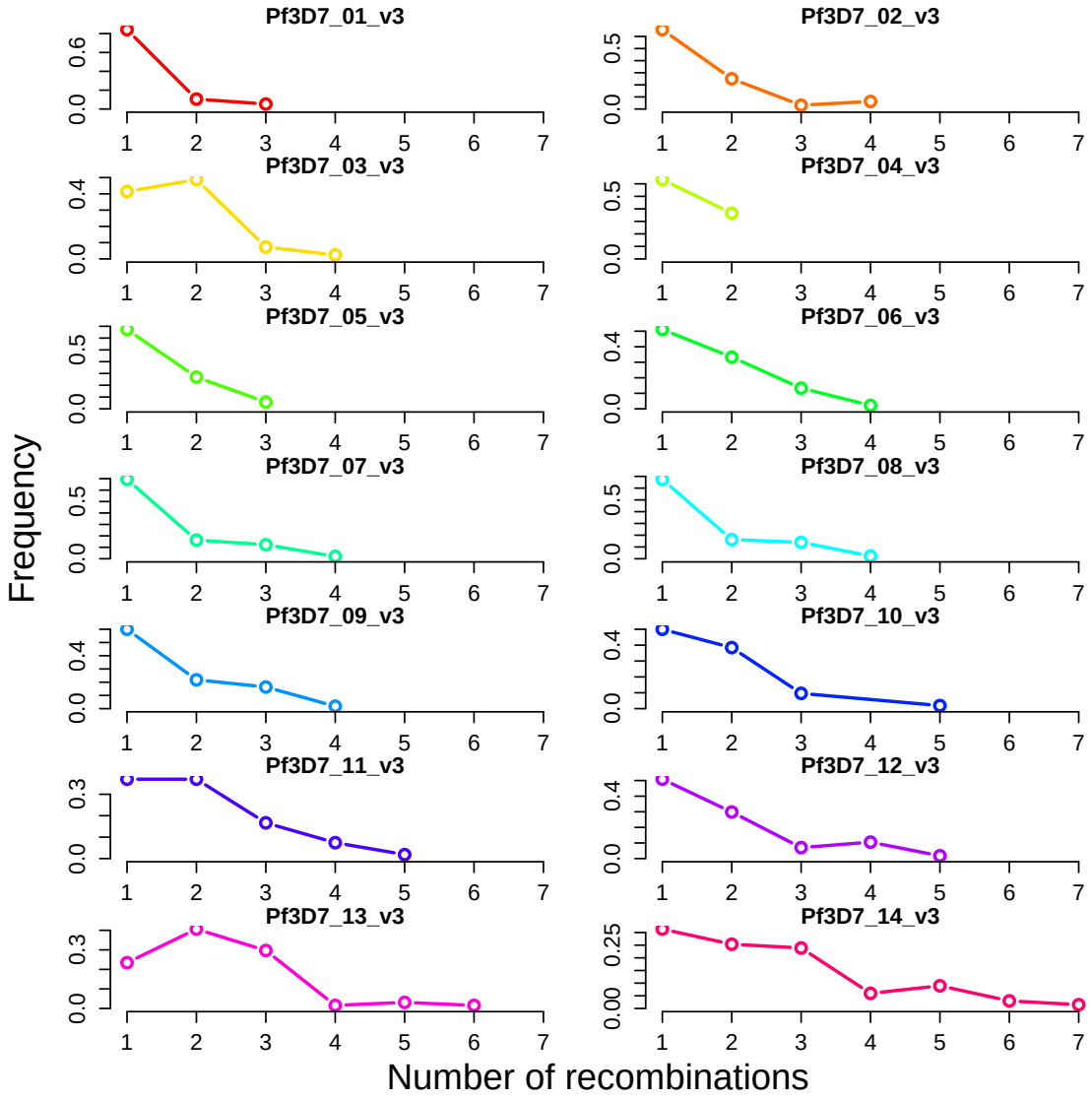

Figure 3.3: Empirical recombination frequencies per chromosome

If these sites were originally transmitted to the child, they were removed from the VCF. If they had not been transmitted, they were added. The homologous recombinations and gene conversion events are displayed below for each chromosome and sample.

3.2.1.3 Non-allelic homologous recombination

NAHR events were generated by first finding compatible recombination partners. This list was generated by determining which 3D7 *var* genes had been transmitted to the child, grouped by upstream promoter class and telomeric positioning (5′ or 3′). In each group, two random genes were chosen. If necessary, the gene sequences were reverse complemented to have matching orientation (note that the sequences may not have the same

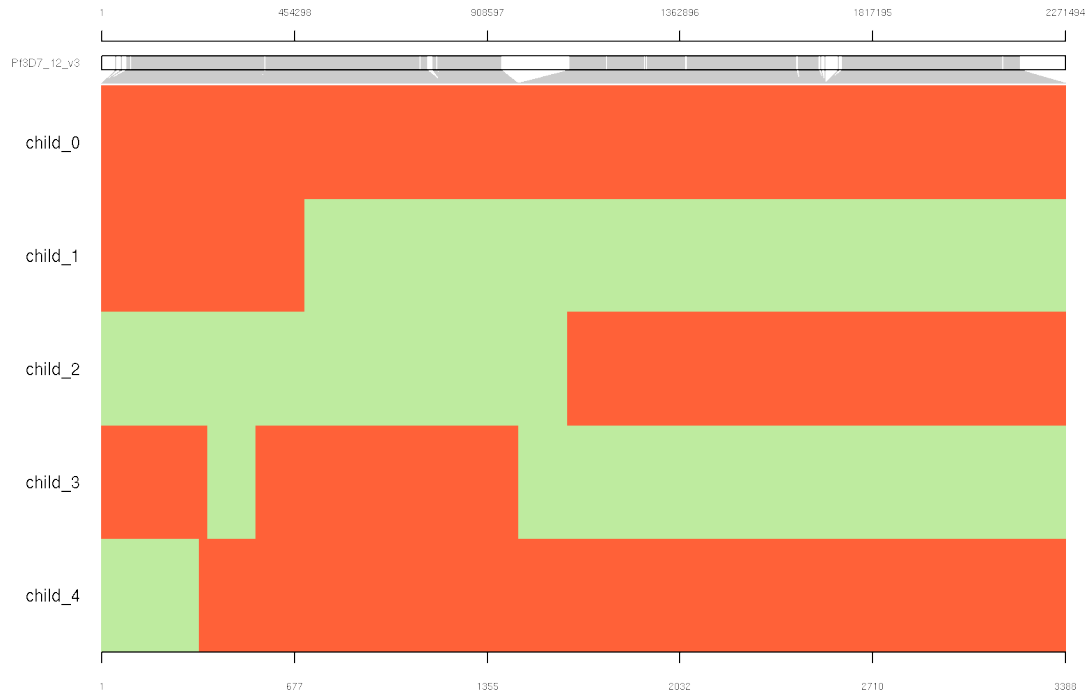

Figure 3.4: Simulated haplotype mosaics for chromosome 12. Genomic position is shown at the top of the figure, while variant number is shown at the bottom. Each variant is depicted as a vertical grey line attached to the mosaic plot at the appropriate location. In the mosaic, every variant is color-coded by parent of origin.

orientation in the simulated genomes themselves). As NAHR events between two *var* genes appear to occur in regions of homology, we identified shared 9-mers, positioned between 20 bp and 100 bp between the two genes, to act as possible recombination sites. We randomly selected between 2 and 5 of these positions to act as recombination breakpoints, switching between the sequences and copied sequence data accordingly. Keeping in mind that our previous work has shown that the recombined *var* genes are lost in order to produce the chimera, we added VCF records to delete the previous *var* alleles from the child's genome and replace one of them with the recombined sequence.

3.2.2 *SNPs, insertions, and deletions*

With the ground state genome now generated, we further generated simple events - SNPs, insertions, and deletions - on this foundation to complete the production of the child's genome. The precise number of events can be specified by the user at runtime, and for each simulated genome, different counts were specified.

To simulate *de novo* SNPs, we added sites with random (non-reference) alleles at random positions throughout the genome. We also simulated insertion, deletion, and inver-

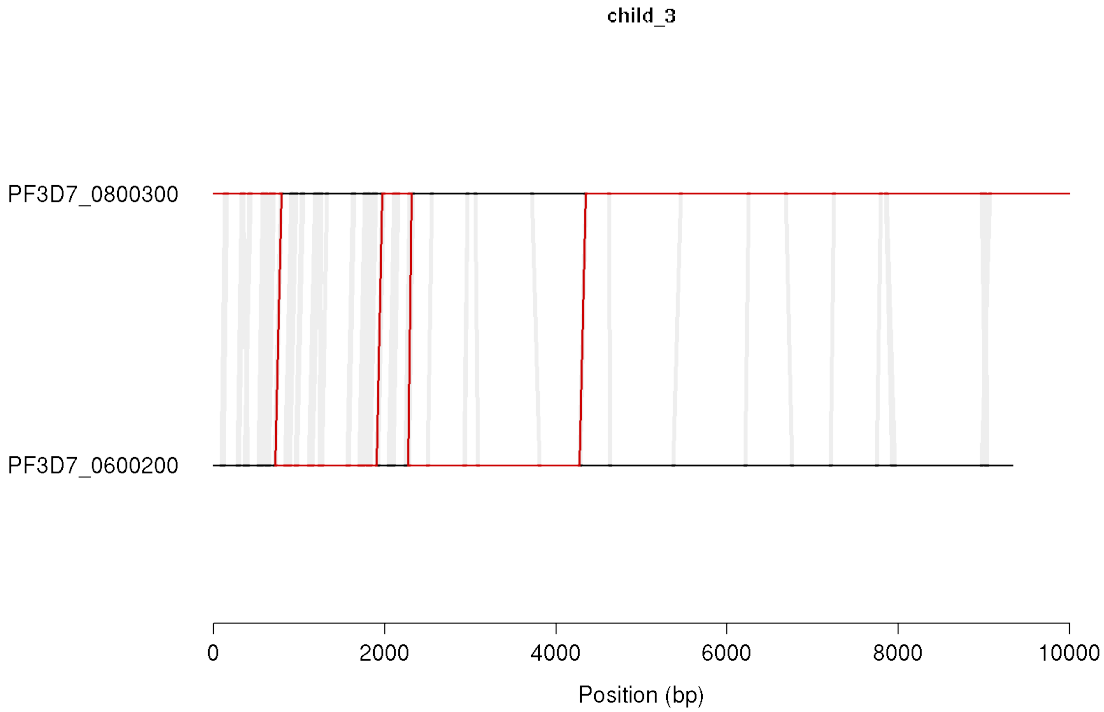

Figure 3.5: Non-allelic recombinations for two compatible *var* genes.

a. parental haplotype ATAAATATTACTCGTCGTCGTTGTGTATACTGCAGT

b. SNP ATAAATATTACTCGTC**A**TCGTTGTGTATACTGCAGT

c. insertion ATAAATATTACTCGTC**TC**ATCGTTGTGTATACTGCAGT

d. deletion ATAAATATTACTCGTCGTTGTGTATACTGCAGT

e. inversion ATAAATATTACTCGTC**CGAG**TTGTGTATACTGCAGT

f. STR expansion ATAAATATTACTCGTCGTCGT**CGT**TGTGTATACTGCAGT

g. STR contraction ATAAATATTACTCGTCGTTGTGTATACTGCAGT

h. tandem duplication ATAAATATTACTCGTCGTCGTTG**CGTCGTCGTTG**TGTATACTGCAGT

Figure 3.6: Simulated variant types. a. Original, parental haplotypic background upon which variants will be placed. b. A single nucleotide polymorphism. c. An insertion of two nucleotides. d. A deletion of three nucleotides. e. An inversion of four nucleotides. f. Expansion of a 3 bp short tandem repeat (STR) by one unit. g. A contraction of an STR by one unit. h. A tandem duplication of 11 nucleotides.

sion events at every length between 1 and 100 bp. For insertions, random alleles were generated and tested to ensure they did not match the allele already in the reference sequence. For deletions, we simply replaced the reference allele with a truncated allele of the prescribed length. For inversions, we replace the reference allele with its complement.

3.2.2.1 *Expansion and contraction of short tandem repeats (STRs)*

As a special case of indels, we sought specifically to simulate the expansion and contraction of short tandem repeats (STRs), depicted in Figure 3.6f-g. STRs are constrained to occur at loci where there are already existing repeats, and expansions (contractions) should manifest as the insertion (deletion) of whole units at a time. We first built a map of repeated 2-bp, 3-bp, 4-bp, and 5-bp sequences in the 3D7 genome. We filtered these lists, retaining only STRs where the repeated unit occurred at least three times. For each simulated event, we randomly chose a position from the appropriate list and select a number of units to add or remove. The number of units is constrained to be less than the length of the number of repeat units of the existing STR. With these considerations, we simulated expansions and contractions at the aforementioned repeat unit sizes.

3.2.2.2 *Tandem duplications*

For tandem duplications (as depicted in Figure 3.6h) of length l, we chose positions in the genome at random, copied the next l bases, and inserted an identical copy at the same locus. Events at each length between 10 bp and 50 bp were produced.

3.3 *Simulating reads*

We now turn our attention to simulating second-generation sequencing reads given an underlying genome. There are existing tools that will simulate perfect reads and uniform coverage, which will be important for initial tests of our variant identification software. However, the crucial simulation is of imperfect reads with non-uniform coverage. Without this, any estimate of our sensitivity and specificity is unlikely to be predictive of performance in real data.

There are many tools that can simulate imperfect reads from a given sequence. Almost every solution involves user-specified parameters controlling the properties of the sequencing data. For instance, a tool may model fragment size as a normal distribution, requiring the user to specify the requisite shape parameters. It may also permit the user to specify a desired mismatch rate to simulate the presence of sequencing error. Some tools have presets that automatically set parameters to those consistent with average behavior for various sequencing platforms.

The problem with all of these tools is an over-reliance on assumed parameters of real data. Should a fragment size distribution for real data deviate from the typical normal distribution, this will not be captured in the simulated dataset. Mismatch and indel errors do not happen at any position in the read with equal probability, but rather are biased

towards later cycles and certain sequence contexts. Read coverage is not simply Poisson-distributed, but varies across the length of the genome depending on GC bias, secondary structure, even the particular sequencing chemistry used. There are currently no tools that are capable of capturing such nuance.

Instead, we chose to learn empirical distributions of major sequencing properties using an exemplar dataset and sample from those distributions directly in order to generate reads. This frees us from having to make any assumptions about our dataset, easily generalizes to any dataset regardless of when, where, or how it was sequenced. It's also vastly more realistic than other approaches.

Our approach is detailed below. Briefly, it consists of three components:

1. A "coverage profile": a description of where every fragment and read in the genome should fall and which should contain errors.
2. A "read profile": a description of where to place errors within a read, including mismatches, insertions, and deletions.
3. The read simulator: samples from the profiles to generate reads in the new reference sequence.

3.3.1 *Constructing the coverage profile*

Construction of a coverage profile consists of two sub-problems: providing a complete description of where reads fall on the existing reference sequence, and transferring that information sensibly to the modified reference sequence. This is depicted in Figure 3.7. We address each of these needs with custom tools.

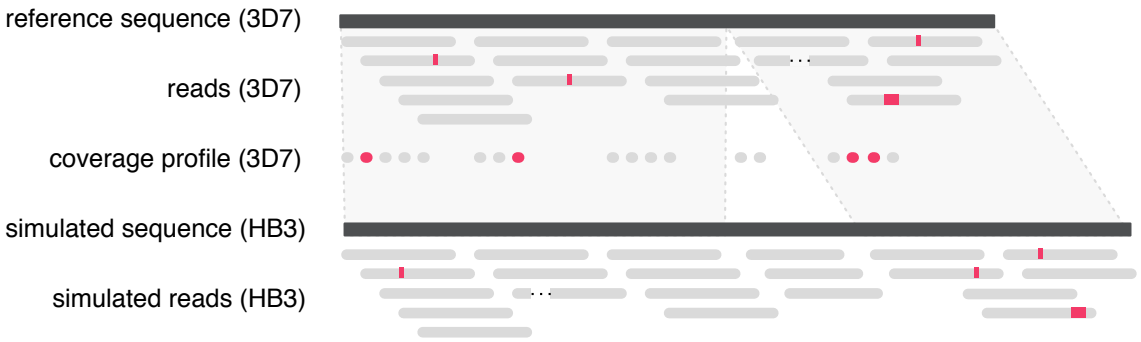

Figure 3.7: Construction of the coverage profile for the reference genome and liftover to the altered genome. Each read start (and fragment start, not shown) is stored along with a count of the number of reads at that location that contain errors. This information is then lifted over to the simulated sequence, and gaps in the table are filled in with neighboring values.

3.3.1.1 *Computing read and fragment starts, and error rates*

We developed a tool, `ComputeBaseAndFragmentErrorRates`, which provides a precise specification for the number of fragments and reads found starting at of every position in the canonical reference genome, as well as an accounting as to which reads and fragments contained any kind of error (mismatch or indels). Briefly, we iterate over every chromosome in the reference genome, advancing through every aligned read stored in an exemplar sample's coordinate-sorted BAM file. We instantiate a chromosome-length array indicating the number of reads that start at a given position and the number of reads that contain apparent errors (any discrepancy from the reference sequence).

We must also store information about read fragment starts and error rates, which requires us to keep track of read pairs as we traverse the BAM file. To do so, we hash the read name to the first instance of the read we see along the length of a chromosome. As paired-end reads have the same name, the second time we see the same read name, we will have found the second end of the pair. We then increment the count at the chromosome array position that corresponds to the 5′-most end of the pair, and increment the fragment errors array based on errors in either end of the pair.

This algorithm is described in Algorithm 3.

3.3.1.2 *Lifting read profile over from reference to simulated genome*

The genomic coordinates present in the table produced by Algorithm 3 must be transformed from the reference sequence to the simulated genome before it can be used. To do so, we developed a tool that could liftover the coordinates appropriately when given the table, reference sequence, and a VCF file describing the alterations made to the reference to transform it into the simulated genome. This is accomplished with an algorithm similar to Algorithm 2. We iterate through each chromosome as we did before, storing the entire sequence as an array of strings, placing alternate alleles in the place of reference alleles, or in the case of deletions, replacing reference alleles with blank strings. Once the array is populated, we advance through the coverage table one position at a time. At each position, we emit the contents of the previously constructed table with the number of reads, fragments, and errors for each. At some positions, insertions will have changed the length of the sequence in the bin, and the extra positions will not have explicit read and fragment information to emit. To account for this, we keep running statistics on previously seen read and fragment statistics from which we can compute running means and standard deviation. We generate new values for the number of reads, fragments, and error rates as necessary as the mean plus standard deviation of the relevant metric, ensuring the values are never less than 0, and that we round up or down to the nearest integer

Algorithm 3 Emit all fragment starts, read starts, and error rates per position.

```
1: function ComputeBaseAndFragmentErrorRates(BAM)
2:     for all chr in chrs do
3:         nReadsErrors ← []
4:         nReads ← []
5:         nFragmentsErrors ← []
6:         nFragments ← []
7:         seenReads ← []
8:         reads ← BAM.getAllReads(chr)
9:         for all read in reads do
10:            readErrorPositions ← getPositionsErrors(read)
11:            refErrorPositions ← convertReadPositionsToReferencePositions(readErrorPositions)
12:            for all refErrorPosition in refErrorPositions do
13:                nReadsErrors[refErrorPosition] + +
14:            for readPosition in 0 : read.length() do
15:                nReads[convertReadPositionsToReferencePositions(readPosition)] + +
16:            if !seenReads.contains(read.getName()) then
17:                seenReads ← read.getName()
18:            else
19:                mateErrorPositions ← getPositionsErrors(seenReads[read.getName()])
20:                if then readErrorPositions.size() + mateErrorPositions.size() ≥ 1
21:                    nFragmentsErrors[seenReads[read.getName()].getAlignmentStart() + +
22:                nFragments[seenReads[read.getName()] + +
23:        for pos in 0 : nReads.length() do
24:            print chr, pos, nReadsErrors[pos], nReads[pos], nFragmentsErrors[pos], nFragments[pos]
```

(error rates are rounded up and read/fragment counts are rounded down to increase our self-penalty).

This algorithm is described in 4.

Algorithm 4 Lift a table from reference to child coordinates.

```
1: function LiftoverFromRefToChild(ref, vcf, table)
2:     for all chr in ref do
3:         newchr = IncorporateVariantsIntoGenome(ref, vcf)
4:         for i in 0 : newchr.length() do
5:             emit(table[i])
6:             if newchr[i].length() > 1 then
7:                 for j in 1 : newchr[i].length() do
8:                     emit(extrapolate(table[i]))
```

3.3.2 *Constructing the read profile*

Next, we generate the read profile, describing the various properties of fragments and read. As in previous examples, we iterate over each read in the exemplar sample's BAM file, storing relevant information in tabular form.

3.3.2.1 *Scalar properties*

We first store the following scalar information (as the data we are modelling is assumed to be Illumina data generated from a single library, some properties which could otherwise be stored as distributions are instead treated as a single number that will be simple constants in our simulation):

1. read length
2. number of reads
3. number of reads with errors
4. number of read pairs
5. number of chimeric pairs (each end aligned to different chromosomes)
6. number of mismatches
7. number of insertions
8. number of deletions

Table 3.4: Example read and fragment scalar properties for sample PG0063-C.

	PG0051-C
readLength	76
numReads	38,846,418
numReadsWithErrors	3,696,708
numPairs	19,473,634
numChimericPairs	348,294
numMismatches	6,525,504
numInsertions	139,941
numDeletions	317,076

Example values for each of these metrics can be found in Table 3.4. Note that some of the mismatches, insertions, and deletions may represent true variation between the sample and the reference sequence. We do not mask these sites out. While this increases the apparent error rate, the variant rates are typically low compared to the number of bases in the reference sequence itself, making the difference negligible. Furthermore, we will choose the reference sample as the exemplar dataset for the read simulations. Since this sample should theoretically be identical to the reference sequence, this choice obviates the need for any masking of variants in the reference sample against the reference sequence.

3.3.2.2 Empirical distributions

Other properties take on a range of values with some probability. Rather than trying to fit the underlying distributions explicitly (which can often fail as datasets contain more nuance than what sequencing platforms should theoretically produce), we instead store empirical distributions and generate random values from them accordingly. We store the following empirical distributions:

1. number of errors (of any type - mismatch, insertion, or deletion) in a read
2. fragment size
3. insertion size
4. deletion size

Note that these distributions are not conditioned on position in the read. Example distributions derived from an exemplar sample are shown in Figure 3.8.

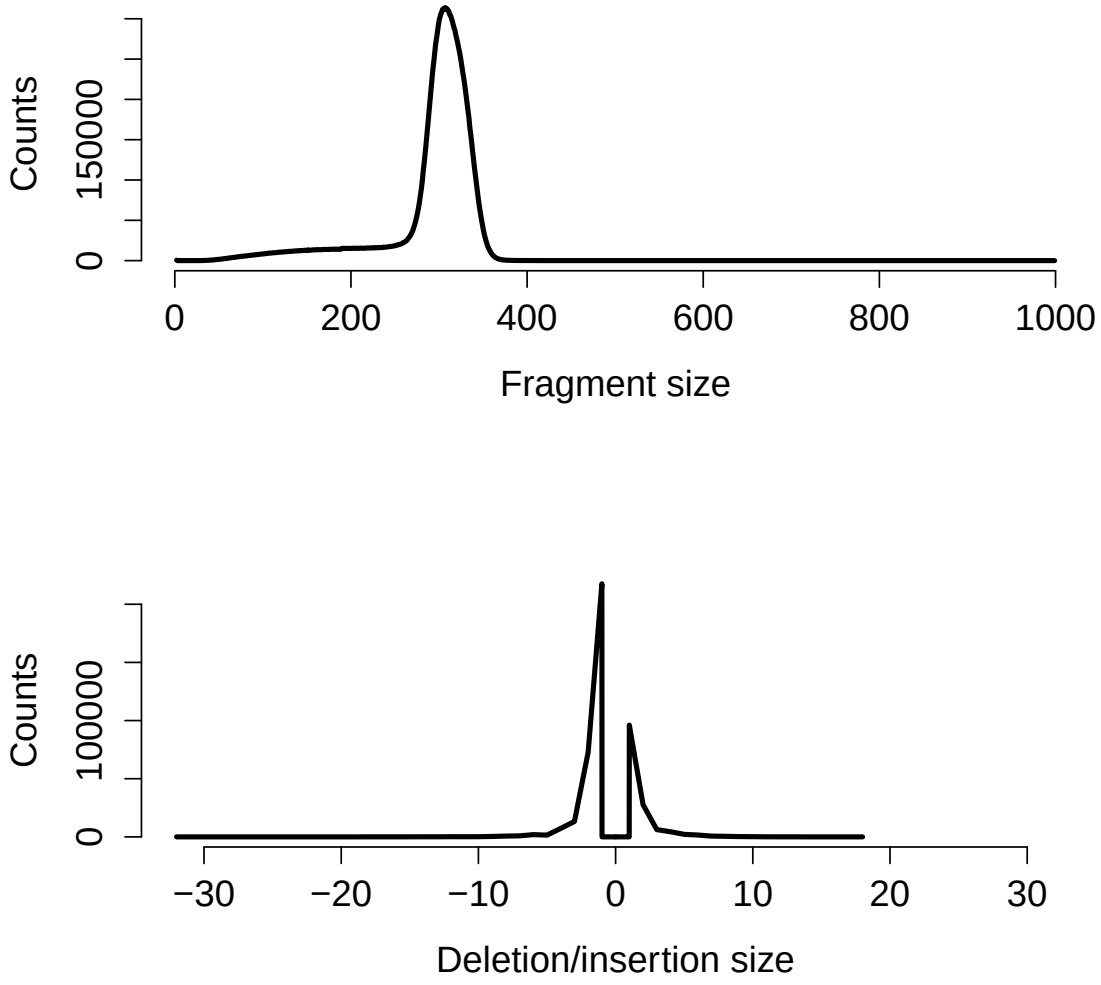

Figure 3.8: Top: empirical fragment size distribution for PG0051-C. Bottom: empirical deletion (negative values) and insertion (positive values) length distribution for the same sample.

3.3.2.3 Covariate table

Finally, we construct the covariate table: a set of empirical distributions for various error events conditioned on the following:

1. type (SNP, insertion, deletion)
2. end of pair (first end or second end)
3. strand (positive or negative)

4. 5′ dinucleotide context

5. position in read

6. first base of error sequence (empty for deletions)

For each read, we increment an element in a table that corresponds to these six covariates. The code to do so is contained in our `GenerateReadSimProfile` module.

3.3.3 *The read simulator*

With the coverage and read profiles in hand, we are finally ready to simulate reads. We advance through each base of the simulated genome, reading the coverage profile as we traverse. At each site, we use the coverage profile to dictate the number of fragments that must be generated and how many of those fragments should contain errors, thus modelling regions of the genome with higher or lower error rates. We then generate fragments, sampling fragment lengths from the empirical fragment length distribution. If a read is to contain an error, as specified by the coverage profile, we construct a read-specific error profile for mismatches, insertions, and deletions. These profiles take into account the preceeding dinucleotide context for each position in the read[2], the end of the pair being simulated, whether the fragment came from the positive or negative strand, and the first nucleotide of the error to be incorporated (not applicable for deletions).

3.4 *The simulated dataset*

With our genome and read simulation framework now complete, we simulated the genomes of 20 progeny from a pseudo 3D7xHB3 cross. The precise counts of variant types per sample are shown in Table 3.5. For each sample, we generated two datasets: a so-called "perfect" dataset (uniform coverage, perfect reads), and a so-called "realistic" dataset (non-uniform coverage, imperfect reads). Both datasets contain approximately $150x$ coverage.

[2]To handle the first and second positions in the read, which do not have a dinucleotide context, we simply extract a read by starting two nucleotides into the simulated fragment.

Table 3.5: *De novo* variant counts for each of the 20 simulated children.

	DEL	GC	INS	INV	NAHR	RECOMB	SNP	STR_CON	STR_EXP	TD
0	22	25	22	22	2	10	5	3	3	26
1	21	21	21	21	2	10	23	11	11	26
2	11	22	11	11	2	8	13	22	22	8
3	25	17	25	25	2	32	23	17	17	21
4	4	23	4	4	2	8	14	16	16	12
5	26	23	26	26	2	37	28	5	5	8
6	13	17	13	13	2	15	27	23	23	4
7	1	22	1	1	2	30	28	5	5	12
8	26	16	26	26	2	16	16	13	13	8
9	4	23	4	4	2	25	23	23	23	21
10	12	20	12	12	2	16	23	19	19	27
11	13	19	13	13	2	7	20	18	18	17
12	19	20	19	19	2	8	14	5	5	28
13	10	20	10	10	2	25	13	28	28	11
14	28	18	28	28	2	8	3	29	29	17
15	27	22	27	27	2	30	5	19	19	2
16	23	22	23	23	2	12	29	24	24	17
17	17	22	17	17	1	10	28	13	13	11
18	23	18	23	23	2	8	1	21	21	18
19	22	22	22	22	2	17	19	3	3	2

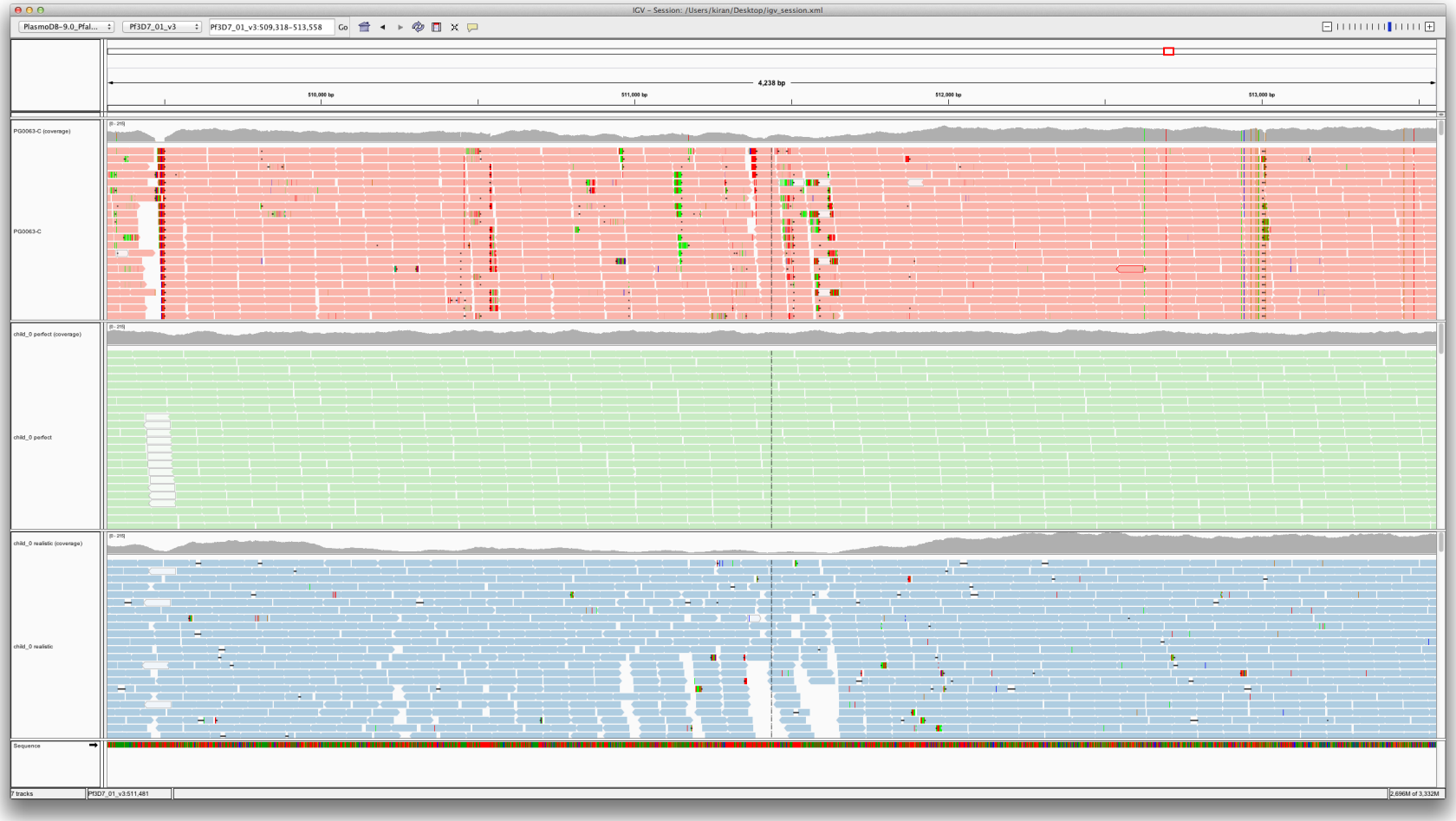

Figure 3.9: Read datasets for real (top panel), simulated perfect (middle panel), and simulated realistic (bottom panel) data.

4 Detection and classification

We now present a novel graphical method for detecting and classifiying *de novo* variants. We shall briefly present the workings of the algorithms, relying on the *de novo* assembly foundational material presented in Chapter 1. The simulation framework and dataset we described in Chapter 3 will be used to establish the method's accuracy.

4.1 Variant motifs

Just as reference-based methods will search for motifs in the data representing variants (e.g. mismatches, gaps, or unusual truncations in the read alignments; read pairs aligning much further apart than expected; chimeras or inter-chromosomal alignments; etc.), so must we scan for indicative motifs in the assembly graph. Before we discuss the precise nature of these motifs, it is useful to draw a distinction between "simple" and "complex" variants. A "simple" variant is a SNP, insertion, or deletion that occurs within a single chromosome. A "complex" variant is a homologous or non-homologous recombination, translocation, or other interchromosomal exchange. The patterns inherent to these two categories of variants are very different.

4.1.1 Simple variant motifs

Simple variants in *de novo* assembly data are typically described "bubbles" in the de Bruijn graph: regions where a variant has broken the homology between sequences, resulting in flanking kmers that are shared between the samples and spanning kmers that differ through the variant itself. In a single diploid sample, this could be a heterozygous SNP or indel between two homologous chromosomes. In haploid samples, one or more samples may differ from the others, resulting in the bubble.

As an illustration, consider three sequences from a mother-father-child pedigree, shown in figure 4.1a. While the maternal and paternal haplotypes (green and blue, respectively) are identical, the child's haplotype (red) differs by a single C to G SNP. Figure 4.1b shows the resulting multi-color de Bruijn $k = 3$ graph built from this data. The mutation has

given rise to the canonical bubble motif in the graph. Three novel kmers (kmers present in the child and absent in the parents) spanning the variant allele are present. Figure 4.2 is an equivalent graph for another simulated SNP, shown with more context and constructed with a much larger value of k appropriate for 76 - 100 bp read lengths, typical of NGS datasets (in this case, $k = 47$).

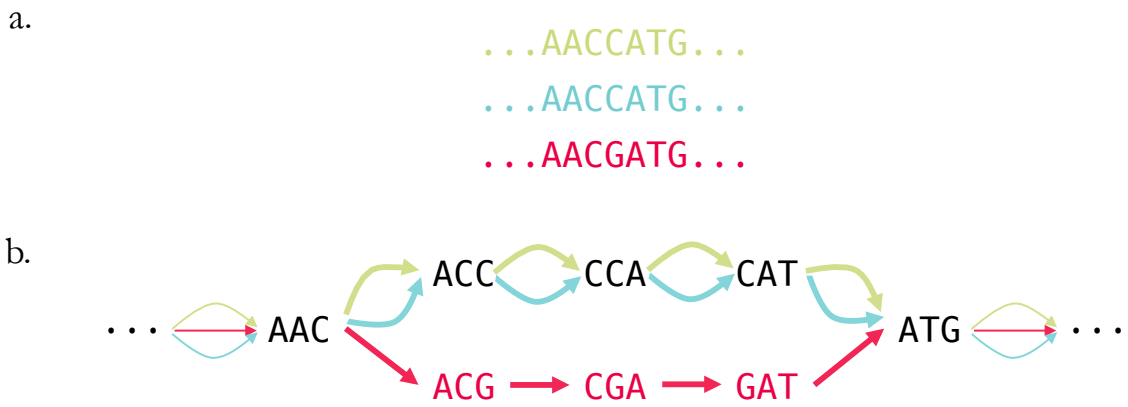

Figure 4.1: a. Haploid sequences from a mother (green), father (blue), and child (red), the last differing from the first two by a single SNP. b. The resulting multi-color de Bruijn graph for $k = 3$. Red vertices denote kmers that are deemed "novel", i.e. present in the child and absent in the parents. Edge colors reflect the samples in which the connected pairs of kmers are found. Edges that are part of the bubble (variant call) are displayed with thicker lines.

All simple variants will have this basic structure: a bubble in the graph that separates the variant samples from the non-variant samples. The only major difference is the length of each branch: longer for an insertion in the child, shorter for a deletion (note that for short events, this is generally not apparent from the display, as evidenced by figures 4.3 and 4.4).

Many variants may occur on the haplotypic background of one parent and not the other. This is common in regions of the genome that are divergent between the two parents. Figure 4.5 depicts one such simulated event. A 41-bp tandem duplication has occurred on the background of the mother (evidenced by the presence of green edges), but not the father (thus the absence of blue edges). In the flanking tails, edges shared between all three samples are present until a blue edge separates from the graph and connects to different vertices. While not shown, these branches continue along the genome of the father.

Finally, it is possible to encounter variants where the path through the graph taken by the child can appear to follow both the variant and non-variant paths, as demonstrated

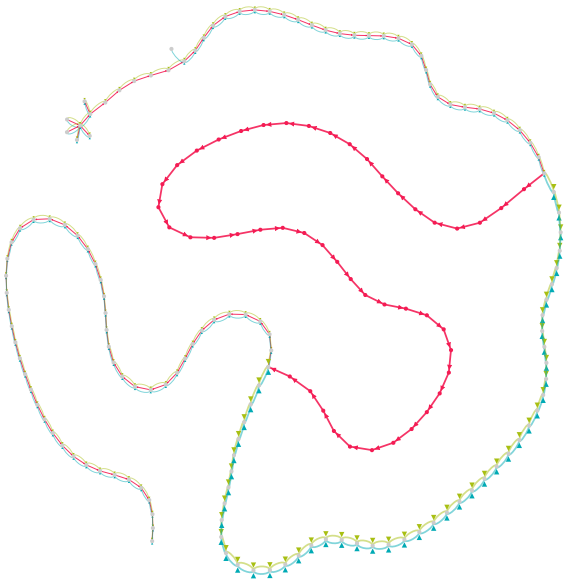

Figure 4.2: A multi-color de Bruijn graph at $k = 47$ for a haploid pedigree spanning a simulated *de novo* SNP. Vertex labels have been supressed for clarity. Spatial layout is arbitrary and for display purposes only.

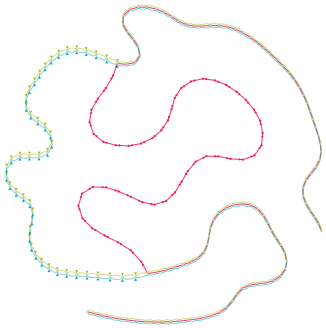

Figure 4.3: A 5 bp insertion in the child

by figure 4.6. Such a scenario may arise by a mutation on a sequence with copy number greater than 1: both the unaltered and altered sequences would then exist simultaneously in the child's genome.

4.1.2 Complex variant motifs

Recombination events (allelic crossovers or gene conversion events; NAHR events) will not necessarily appear as bubbles. Bubbles form in the graph when parental and progeny haplotypes diverge (at the site of a variant) and reconnect (at the flanking homologous regions). In a recombination event, the haplotypes do not necessarily reconnect. In a

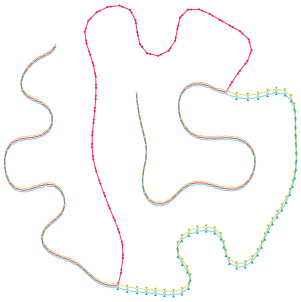

Figure 4.4: A 5 bp deletion in the child

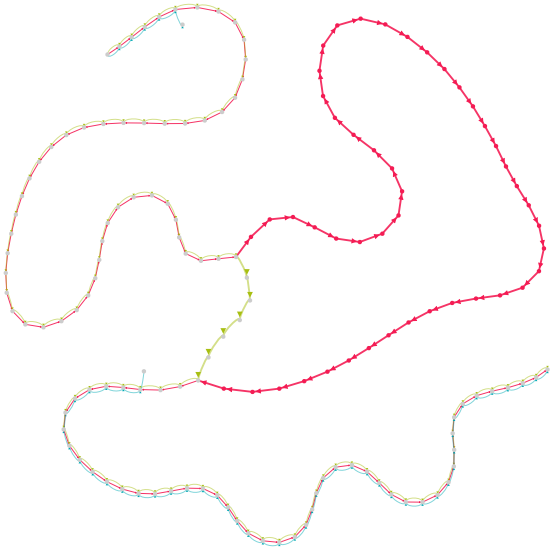

Figure 4.5: A tandem duplication on the haplotypic background of the mother.

crossover or gene conversion event, as in Figure **??**, the progeny's graph should follow one parent or the other, switching at the crossover site. Gene conversions and other multiple crossovers may switch back and forth several times. These events can be detected simply by keeping track of which parent's graph is apparently being followed. However, we caution the reader that it's quite likely that many events of this nature will likely go uncalled or improperly called. For such recombination events to be detected, a kmer spanning two proximal variants must be present. This is unlikely to occur for simple crossovers, particularly in genomes of reasonably low heterozygosity, as neighboring variants beyond a kmer's length away will not give rise to novel kmers necessary for the event's detection. It is perhaps slightly more likely for gene conversion events, where the multiple crossovers proximal to a variant exclusive to one of the parents will generate the

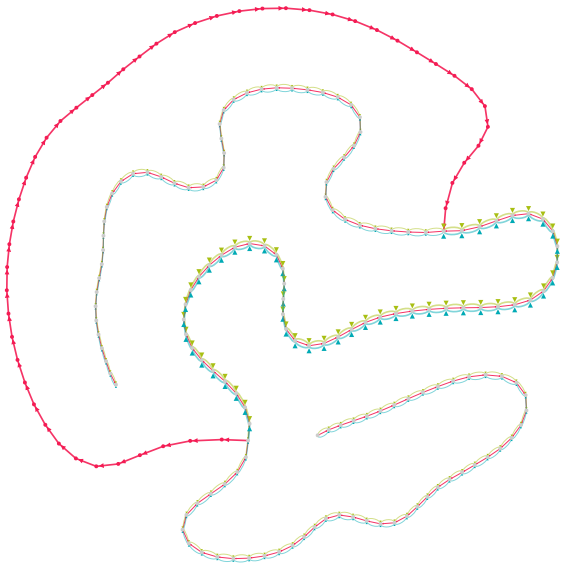

Figure 4.6: A variant wherein the child's path does not simply diverge from that of the parents, but rather navigates both.

sought-after novel kmer signal.

NAHR events are trickier; as these events are generally mitotic rather than meiotic events, the expected motif is that a child's graph should follow the same parent, but connect disjoint components of the graph (e.g. telomeres of different chromosomes) through novel kmers. In principle, one should be able to detect such an event by testing whether removing the child's contribution to the pedigree subgraph results in disrupting the otherwise connected components.[41] In practice, however, NAHR events are mediated by homology between low-complexity regions of the subtelomeric genomic regions. The homology will result in confounding connections between disparate regions of the graph. An easier solution is to track the parent apparently being copied from and determine the chromosome of origin for each kmer present in the flanking regions of the novel kmers. This is only feasible if one happens to have draft reference genomes for which a kmer's chromosome of origin can be approxiately determined.

4.1.3 Handling errors in sequencing

Bubble and non-bubble motifs are trivial to find and navigate if there are no errors in the sequence data, but much more difficult to navigate in the presence of errors. Errors manifest as extraneous branches in the graph, adding ambiguity to traversals. Without a guide as to which branch to explore at a junction, we are either forced to abandon the traversal, make a guess, or explore all possible branches. The former is overly conservative; many variants will go untyped. The second is hazardous; there's the very real potential

for choosing erroneously and typing the variant incorrectly. The latter is computationally expensive; errors explore the complexity of the graph, making it very difficult (if not impossible) to find the correct path through the graph.

To solve this, recall that DNMs do more than open up a bubble motif in the graph. Idaelly, each kmer along the variant path is novel. If we assume that branches following the novl branch at junctions are correct and linked with the current variant being explored, we can use these novel kmers as "sign posts" to mark successful traversals. Navigating a branch and not seeing a sign post gives us adqeuate cause to abandon a branch in favor of another.

4.2 Calling and classifing de novo variants

Armed with an intuition as to how graphs behave in regions of *de novo* variation, we can now describe the procedure for identifying and classifying a variant. The overview involves five big steps (and associated substeps):

1. Identify confident and trusted novel kmers
2. Construct multi-color de Bruijn "trio" graphs (child, mother, father)
3. Load subgraph local to a novel kmer
4. Identify and classify variants in the subgraph
5. Evaluate performance

We discuss these in details in the sections below.

4.2.1 Identify confident and trusted novel kmers

We first seek to build a list of novel kmers that are both *confident* (i.e. unlikely to be sequencing errors) and *trusted* (i.e. are unlikely to be the result of contamination). Identification of the novel kmers themselves is trivial; we simply build a list of kmers that appear in the child but are completely absent in the parents. The subsequent filtering steps are described below.

4.2.1.1 Remove low coverage kmers

For a deeply sequenced sample, we expect all kmers in the genome to be of similarly high coverage. While there will inevitably be regions of the genome where coverage is poor owing to failure to amplify regions with high GC content, the Lander-Waterman statistics

in Chapter 1 suggest we should easily see many copies of the entire *P. falciparum* genome at a coverage of $100x$. We can therefore assume that failure to reach a certain coverage threshold is indicative of sequencing error, and such kmers can be removed as candidates from the novel kmer list.

The problem remains as to where the threshold should be set. We plotted the histogram of kmer coverage, shown in Figure 4.7, smoothing the resulting distribution with a non-parametric LOESS fit. The smoothed histogram makes it easier to find the first local minimum using Algorithm 5.

Algorithm 5 Compute the local minimum of a kmer coverage distribution

1: **function** FIRSTLOCALMINIMUM(coverageHist)
2: $y = laggedDifferences(c(INT_MAX, x)) > 0L$
3: $y = cumSum(equalRunLengths(y).lengths)$
4: $y = y[seq(2L, length(y), 2L)]$
5: return(y[2])

4.2.1.2 *Remove possible contaminants*

Contamination can occur during sequencing library preparation due to handling from human operators (transferring either bacterial or human genetic material to the library) or from other samples (often from different species) being processed in the same laboratory. Contamination would result in kmers that appear novel - owing to their absense in the parents - but are irrelevant for our study. It is perhaps not a paramount concern when processing *P. falciparum* data, as the genome is very different than any other sample likely to be a contaminant. However, it may contribute a false-positive rate that we can easily mitigate.

To remove the effects of contamination, we run every confident novel kmer through BLAST.[42] Specifically, we used the `blastn` package and all available BLAST nucleotide databases to identify the likely species of every confident novel 47-mer in each sample. Any unidentified kmer or apparently *P. falciparum* kmer was retained; all others were removed. In practice, this removes anywhere from dozens to thousands of kmers from needing to be considered; as evidenced by Figure 4.8, the exact number varies as each sample is prepared independently, at different times and by different personnel.

4.2.2 *Construct multi-color de Bruijn "trio" graphs*

To construct the "trio" graphs, we perform assembly on each sample with the *Cortex de novo* assembly software[28] following the recommended workflow.[43] Briefly, each sample is

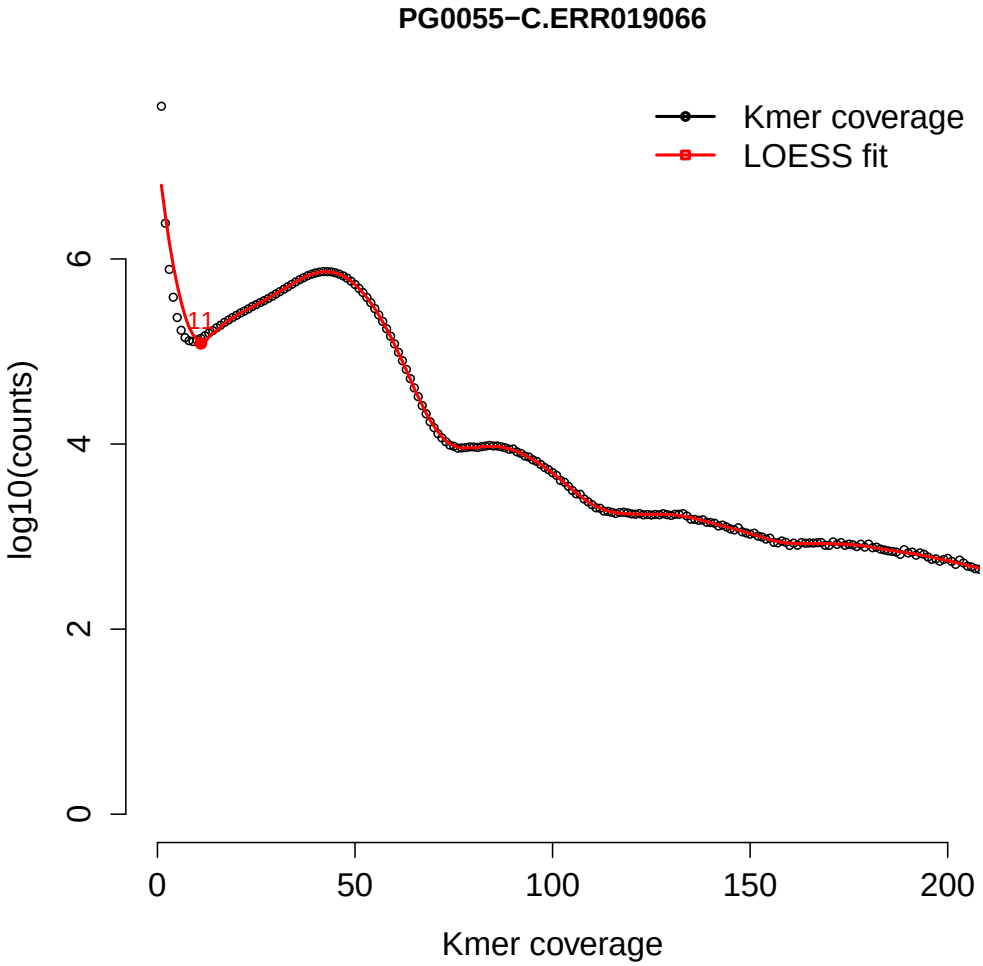

Figure 4.7: Kmer coverage histogram for a real sample, with LOESS fit

assembled using the `build` command at a kmer size of 47 bp[1] and ignoring nucleotides with an Illumina quality score less than 5. Each sample was then cleaned of likely sequencing errors with the `clean` command using automatically calculated coverage and supernode (unambiguous runs of kmers with in/out degree of 1) length thresholds. Finally, the graph for the child was merged into a *clean* trio graph, and separately, a *dirty* trio graph (one using the uncleaned graph data) using the `join` command. This multi-color graph consists of child, mother, and father. For our purposes with *P. falciparum*, "child" is assigned color 0, "mother" (the first parent of the cross) to color 1, and "father" to color 2. Read threading was *not* applied, as in our analyses, there is no need for contigs, just the graph data structure.

[1]This setting results from evaluations on optimal kmer size for maximizing contig length, despite not needing to produce contigs for our analyses

4.2.3 *Load subgraph local to a novel kmer*

To facilitate variant calling, we must process regions of the graph likely to harbor DNMs. While it is technically possible to load a *P. falciparum* graph into RAM (owing to its small 23 megabase genome), it is unlikely such a solution would scale to larger genomes. It is also cumbersome to do so when there are only on the order of dozens, perhaps hundreds, of variants to be discovered per genome. Therefore, we adopted a solution of fetching only relevant parts of the genome as necessary, operating on the local subgraph surrounding the putative variant, rather than on the entire graph at once.

4.2.3.1 *Depth-first search*

Graph exploration is an *online* problem, meaning the structure of the graph cannot be known until it is explored. As a non-linear data structure, it is not trivial to determine where to start and stop exploring a graph. During traversal, one may encounter junctions (due to errors or homology) without any additional information as to which branch to choose. Graphs sometimes loop back onto themselves, necessitating that we keep track of our walk so that we do not end up traversing endlessly in circles. Incorrect decisions in the traversal cannot always be detected, requiring appropriate stopping conditions to abort a traversal if no fruitful data is discovered.

Typically, graph exploration can be accomplished with a so-called *depth-first search*, or DFS. The basic premise of a DFS is summarized in Algorithm 6: start at a vertex, walk in a chosen direction until a junction is encountered, choose one branch and repeat the DFS from that point until it is deemed appropriate to stop, then jump back to the junction to choose the next branch, and repeat until completion. In practice, this approach has some drawbacks that manifest quickly in real data. First, the basic DFS algorithm terminates only when there are no more vertices left to traverse, which is overkill for our purposes. Second, all branches are treated equally. For our purposes, branches containing errors are not important and should be discarded lest they confuse later algorithms trying to find paths through bubbles in order to type variants. Third, this requires us to explore three graphs separately, which is inefficient.

The solution is to instead conduct a DFS with stopping conditions and recursively. Stopping conditions, specifying the conditions upon which a traversal is deemed "successful" or "failed" allow us to decide certain branches in the subgraph are unfruitful for analysis. The recursive traversal allows us to act on the result of the stopping condition, adding it to the subgraph if successful, discarding it if not. This is described in Algorithm 7.

Algorithm 6 A basic, iterative depth-first search

```
1:  function IDFS(graph, vertex)
2:      s = new Stack()
3:      visited = {}
4:      s.push(vertex)
5:      while !s.isEmpty() do
6:          current = s.pop()
7:          if visited.contains(current) then
8:              next
9:          visited.add(current)
10:         for v in graph.nextVertices(vertex) do
11:             s.push(v)
```

Stopping conditions are implemented as a callback object, permitting programmer-specified limits for different situations. To type DNMs, we begin traversal at a novel kmer, walking forward and backward in the graph until the stopping conditions are met. We then explore the parental graphs based on the subgraph we loaded for the child. As the parental traversals have some additional information (namely, the presence of the child's subgraph allows us to check if a parental traversal has diverged and rejoined from the graph), the stopping conditions are different.

The stopping condition callback object implements two boolean methods: `hasTraversalSucceeded` and `hasTraversalFailed`. Both of them may return `false`, in which case the traversal continues. If either returns `true`, traversal is halted and the branch is evaluated for retention or rejection.

4.2.3.2 Stopping conditions for child

The child's stopping conditions on traversal success or failure are provided in Algorithm 8 and 9, respectively. For the child, success is approximately determined by having explored a novel kmer stretch in the graph to the point that it has rejoined the parental graphs. However, we purposefully continue reading another 50 kmers before returning success. If more novel kmers are recovered in that span, we reset our counters and continue walking. This facilitates two things: the absence of novel kmers due to sequencing errors or overthresholding, and typing complex variants that may have short stretches of non-novel kmers. Failure is determined by a number of criteria, including the absence of novel kmers, low complexity regions, having no more branches to explore, having reached a maximum graph depth, or having reached a maximum graph size.

Algorithm 7 The recursive depth-first search with arbitrary stopping conditions

```
1: function RDFS(clean, dirty, kmer, color, g, stopper, depth, goForward, history)
2:     firstKmer = kmer
3:     sourceKmersAllColors = goForward ? getPrevKmers(clean, dirty, kmer) : getNextKmers(clean, dirty, kmer)
4:     sourceKmers = sourceKmersAllColors.get(color)
5:     dfs = new Graph<AnnotatedVertex, AnnotatedEdge>()
6:     stopper = instantiateStopper(stopperClass)
7:     repeat
8:         cv = kmer
9:         cr = clean.findRecord(cv)
10:        if (cr == null && dirty != null) then
11:            cr = dirty.findRecord(cv)
12:        prevKmers = CortexUtils.getPrevKmers(clean, dirty, cv);
13:        nextKmers = CortexUtils.getNextKmers(clean, dirty, cv));
14:        adjKmers = goForward ? nextKmers : prevKmers;
15:        numVerticesAdded = addVertexAndConnect(dfs, cv, prevKmers, nextKmers)
16:        if stopper.keepGoing(cr, g, depth, dfs.vertexSet().size(), adjKmers.get(color).size()) && !sourceKmers.contains(kmer) && !history.contains(kmer) then
17:            history.add(kmer);
18:            if adjKmers.get(color).size() == 1 then
19:                kmer = adjKmers.get(color).iterator().next();
20:            else if adjKmers.get(color).size() != 1 then
21:                childrenWereSuccessful = false;
22:                for ak in adjKmers.get(color) do
23:                    if (!ak.equals(firstKmer)) then
24:                        branch = dfs(clean, dirty, ak, color, g, stopperClass, depth + isNovelKmer(cr) ? 0 : 1, goForward, history)
25:                        if branch != null then
26:                            Graphs.addGraph(dfs, branch)
27:                            childrenWereSuccessful = true
28:                        else
29:                            for av in dfs.vertexSet() do
30:                                if av.getKmer().equals(ak) then
31:                                    av.setFlag("branchRejected")
32:                if childrenWereSuccessful || stopper.hasTraversalSucceeded(cr, g, depth, dfs.vertexSet().size(), 0) then
33:                    return dfs
34:                else
35:                    for av in dfs.vertexSet() do
36:                        if (av.getKmer().equals(kmer)) then
37:                            av.setFlag("branchRejected")
38:        else if stopper.traversalSucceeded() then
39:            return dfs
40:        else
41:            return null
42:    until (adjKmers.get(color).size() == 1)
43:    return null
```

Algorithm 8 Child's traversal success determination method

```
1: function hasTraversalSucceeded(cr, g, depth, size, edges)
2:     if goalSize == 0 && (cr.getCoverage(1) > 0 || cr.getCoverage(2) > 0) then
3:         goalSize = size
4:         goalDepth = depth
5:     if goalSize > 0 && isNovel(cr) then
6:         goalSize = size
7:         goalDepth = depth
8:     return (goalSize > 0 && (size >= goalSize + 50 || isLowComplexity(cr) || edges
   == 0))
```

Algorithm 9 Child's traversal failure determination method

```
1: function hasTraversalFailed(cr, g, depth, size, edges)
2:     return !isNovel(cr) && (isLowComplexity(cr) || edges == 0 || depth >= 5 || size
   > 5000)
```

4.2.3.3 Stopping conditions for parents

The parents' stopping conditions on traversal success or failure are provided in Algorithm 10 and 11, respectively. Success is determined by having diverged from the child's graph and rejoined it. Care is taken to ensure that we have rejoined the graph at a boundary of the novel kmer stretch, rather than some ther part of the graph obtained when trying to read some extra flanking data in Algorithm 8. Failure is determined by a number of criteria, including having reached a maximum graph size, having reached a maximum graph depth, or low complexity regions.

4.2.4 Identify and classify variants in the subgraph

With the relevant subgraph now loaded, we can now attempt to type variants. Typing involves three steps:

1. Annotate vertices in the graph that can be possible start and end points of the variant bubble
2. Find the shortest path from start to end that satisfies some conditions
3. From the haplotypes, remove the flanking homologous sequence to reveal the variant alleles

Let us first detail how we annotate the viable starts and ends of the variant, which simply amounts to iterating through each vertex in the subgraph and checking that it means various conditions. A variant start (end) vertex should have out degree (in degree)

Algorithm 10 Parents' traversal success determination method

```
1: function hasTraversalSucceeded(cr, g, depth, size, edges)
2:     fw = cr.getKmerAsString()
3:     rc = reverseComplement(fw)
4:     v = null
5:     rejoinedGraph = false
6:     if g.containsVertex(fw) || g.containsVertex(rc) then
7:         rejoinedGraph = true
8:         for av in g.vertexSet() do
9:             if (av.getKmer().equals(fw) || av.getKmer().equals(rc)) then
10:                if (!sawPredecessorFirst && !sawSuccessorFirst) then
11:                    if (av.flagIsSet("predecessor")) then sawPredecessorFirst = true
12:                    else if (av.flagIsSet("successor")) then sawSuccessorFirst = true
13:                if ((sawPredecessorFirst && av.flagIsSet("predecessor")) || (sawSuccessorFirst && av.flagIsSet("successor"))) then
14:                    rejoinedGraph = false
15:                    break
16:    return size > 1 && rejoinedGraph
```

Algorithm 11 Parents' traversal failure determination method

```
1: function hasTraversalFailed(cr, g, depth, size, edges)
2:     return size > 1000 || junctions >= 5 || isLowComplexity(cr)
```

greater than 1. The branches should be color-specific to reflect the opening of a bubble between the different samples in the graph. This is detailed in Algorithm 12.

4.2.4.1 Dijkstra's shortest path algorithm

With all of the start and end points now annotated, we now need to find a path from one end of the putative variant to the other. The number of possible paths could be very large. We shall make the assumption that the correct path is the shortest path. While this will often be the case, we note that the shortest path is not necessarily the biological path. This could be the case in highly repetitive regions longer than a kmer length. Since de Bruijn graphs tend to collapse long repeats, we may miss some events.

E. W. Dijkstra described his shortest path algorithm in 1959.[44] We describe it below in Algorithm 13. Briefly, we keep track of all vertices' distance from the source vertex, setting the source's distance to itself to 0 and all others to infinity (to denote as-of-yet unvisited vertices). We then select the vertex with the minimum distance to the source (in the first iteration, the source itself) and compute a new distance to all immediately adjacent vertices. This new distance is a sum of the distance traversed so far from the source to one of these vertices. For our purposes, we shall assume the distance between two

Algorithm 12 Annotating possible variant starts and ends of the subgraph

```
function annotateStartsAndEnds(b)
    for av in b.vertexSet() do
        if b.outDegreeOf(av) > 1 then
            aes = b.outgoingEdgesOf(av)
            childEdges = {}
            parentEdges = {}
            for AnnotatedEdge ae in aes do
                if ae.isPresent(0) && ae.isAbsent(1) && ae.isAbsent(2) then
                    childEdges.add(ae)
                if ae.isPresent(color) then
                    parentEdges.add(ae)
            if childEdges.size() > 0 && parentEdges.size() > 0 || beAggressive then
                av.setFlag("start")
                candidateStarts.add(av)
        if b.inDegreeOf(av) > 1 then
            aes = b.incomingEdgesOf(av)
            childEdges = {}
            parentEdges = {}
            for AnnotatedEdge ae in aes do
                if ae.isPresent(0) && ae.isAbsent(1) && ae.isAbsent(2) then
                    childEdges.add(ae)
                if ae.isPresent(color) then
                    parentEdges.add(ae)
            if (childEdges.size() > 0 && parentEdges.size() > 0) || beAggressive then
                av.setFlag("end")
                candidateEnds.add(av)
```

adjacent vertices is always 1. If the distance computed is less than the distance recorded, we replace the old value with the new. We remove this vertex from the processing queue and proceed to the next vertex with the lowest distance from the source.

Algorithm 13 Finding the shortest path in a graph

```
function DSPA(graph, source, destination, color)
    dist = {}
    prev = {}
    q = []
    for all v in g.vertexSet() do
        dist[v] = infinity
        prev[v] = null
        push(q, v)
    dist[source] = 1
    while !q.isEmpty() do
        u = vertexWithMinDistance(q, dist)
        if u == destination then
            break
        q.remove(u);
        if u != -1 then
            for all e in g.outgoingEdgesOf(u, color) do
                v = g.getEdgeTarget(e, color)
                alt = dist[u] + 1
                if alt < dist[v] then
                    dist[v] = alt
                    prev[v] = u
    s = []
    Integer u = destination
    while u != null && prev.containsKey(u) do
        push(s, u)
        u = prev[u]
    return s
```

To use this algorithm for allele identification, we iterate through all possible combinations of start and stop vertices, obtained as described in the previous section. We then iterate through the three graph colors (representing the child, mother, and father). For each color, we apply Algorithm 13. For the child, we place the additional constraint that the path accepted must contain at least one novel kmer.

This algorithm will return the child and parental haplotypes. To identify the precise alleles of the event, we simply trim back the homologous regions of the haplotypes with Algorithm 14.

Algorithm 14 Finding the shortest path in a graph

```
function trimHaplotypes(child, parent)
    e0 = child.length() - 1
    e1 = parent.length() - 1
    s = 0
    length = (child.length() < parent.length() ? child.length() : parent.length())
    for (s = 0; s < length && child[s] == parent[s]; s++) do
        (do nothing)
    while (e0 > s && e1 > s && child[e0] == parent[e1]) do
        e0–
        e1–
```

4.2.4.2 *Classify event*

For simple variants (those that conform to the bubble motif), event classification is reasonably straightforward. We simply inspect the recovered alleles and evaluate them against simple rules. For example, a SNP should be a single nucleotide in length. An insertion should have a parental allele length of 1 and a child allele length > 1.

To classify complex variants, we rely on other heuristics. GC events are classified by keeping track of which parent the child's genome appears to be exclusively copying from (simply by measuring when kmer coverage has dropped from one parent and returned in the other), detailed in Algorithm 15.

4.2.4.3 *Mark traversed novel kmers as used*

4.2.5 *Evaluate performance*

4.2.5.1 *Generate novel kmer to variant map*

4.2.5.2 *Load variant containing a novel kmer*

4.2.5.3 *Compare alleles*

4.2.6 *Summary*

4.3 *Results on simulated data*

Algorithm 15 Stretch has switches

```
function hasSwitches(graph, stretch)
    inherit = []
    for all 47-bp kmers k in stretch do
        cov0 = graph.getCoverage(k, 0)                    ▷ child
        cov1 = graph.getCoverage(k, 1)                    ▷ mother
        cov2 = graph.getCoverage(k, 2)                    ▷ father
        if (cov0 > 0 && cov1 == 0 && cov2 == 0) then
            append(inherit, "C")
        else if (cov0 > 0 && cov1 > 0 && cov2 == 0) then
            append(inherit, "M")
        else if (cov0 > 0 && cov1 == 0 && cov2 > 0) then
            append(inherit, "D")
        else if (cov0 > 0 && cov1 > 0 && cov2 > 0) then
            append(inherit, "B")
    for i in inherit.length() do
        for inherit[i] == 'B' do
            prevContext = '?'
            nextContext = '?'
            for (j = i - 1; j >= 0; j–) do
                if (inherit[j] == 'M' || inherit[j] == 'D') then
                    prevContext = inherit[j]
                    break
            for (j = i + 1; j < inherit.length(); j++) do
                if (inherit[j] == 'M' || inherit[j] == 'D') then
                    nextContext = inherit[j];
                    break;
            context = '?';
            if (prevContext == nextContext && prevContext != '?') then
                context = prevContext
            else if (prevContext != nextContext) then
                if (prevContext != '?') then
                    context = prevContext
                else
                    context = nextContext
            inherit[i] = context
    return inheritStr.matches(".*D+.C+.M+.*") || inheritStr.matches(".*M+.C+.D+.*")
```

Table 4.1: ROC metrics on simulated perfect data

sn	sim	fp	fn	tp	tn	rc	sens	spec	prec	npv	fpr	fnr	fdr	acc
5	perfect	1	5	126	22240910	9	0.9618	1	0.9921	1	0	0.0382	0.0079	1
3	perfect	2	3	154	22200660	4	0.9809	1	0.9872	1	0	0.0191	0.0128	1
0	perfect	0	2	106	22241343	7	0.9815	1	1.0000	1	0	0.0185	0.0000	1
12	perfect	0	2	113	22233659	3	0.9826	1	1.0000	1	0	0.0174	0.0000	1
13	perfect	0	2	114	22217297	5	0.9828	1	1.0000	1	0	0.0172	0.0000	1
17	perfect	0	2	117	22242890	6	0.9832	1	1.0000	1	0	0.0168	0.0000	1
1	perfect	1	2	137	22249556	8	0.9856	1	0.9928	1	0	0.0144	0.0072	1
14	perfect	2	2	162	22196410	7	0.9878	1	0.9878	1	0	0.0122	0.0122	1
11	perfect	1	1	114	22207761	2	0.9913	1	0.9913	1	0	0.0087	0.0087	1
6	perfect	1	1	118	22203683	2	0.9916	1	0.9916	1	0	0.0084	0.0084	1
8	perfect	1	1	129	22229379	5	0.9923	1	0.9923	1	0	0.0077	0.0077	1
10	perfect	0	1	130	22238480	2	0.9924	1	1.0000	1	0	0.0076	0.0000	1
15	perfect	1	1	131	22208991	10	0.9924	1	0.9924	1	0	0.0076	0.0076	1
16	perfect	1	1	166	22213928	4	0.9940	1	0.9940	1	0	0.0060	0.0060	1
2	perfect	1	0	105	22225262	3	1.0000	1	0.9906	1	0	0.0000	0.0094	1
4	perfect	2	0	72	22210167	2	1.0000	1	0.9730	1	0	0.0000	0.0270	1
7	perfect	1	0	57	22225612	1	1.0000	1	0.9828	1	0	0.0000	0.0172	1
9	perfect	0	0	107	22255460	2	1.0000	1	1.0000	1	0	0.0000	0.0000	1
18	perfect	2	0	133	22242522	0	1.0000	1	0.9852	1	0	0.0000	0.0148	1
19	perfect	1	0	98	22236313	6	1.0000	1	0.9899	1	0	0.0000	0.0101	1

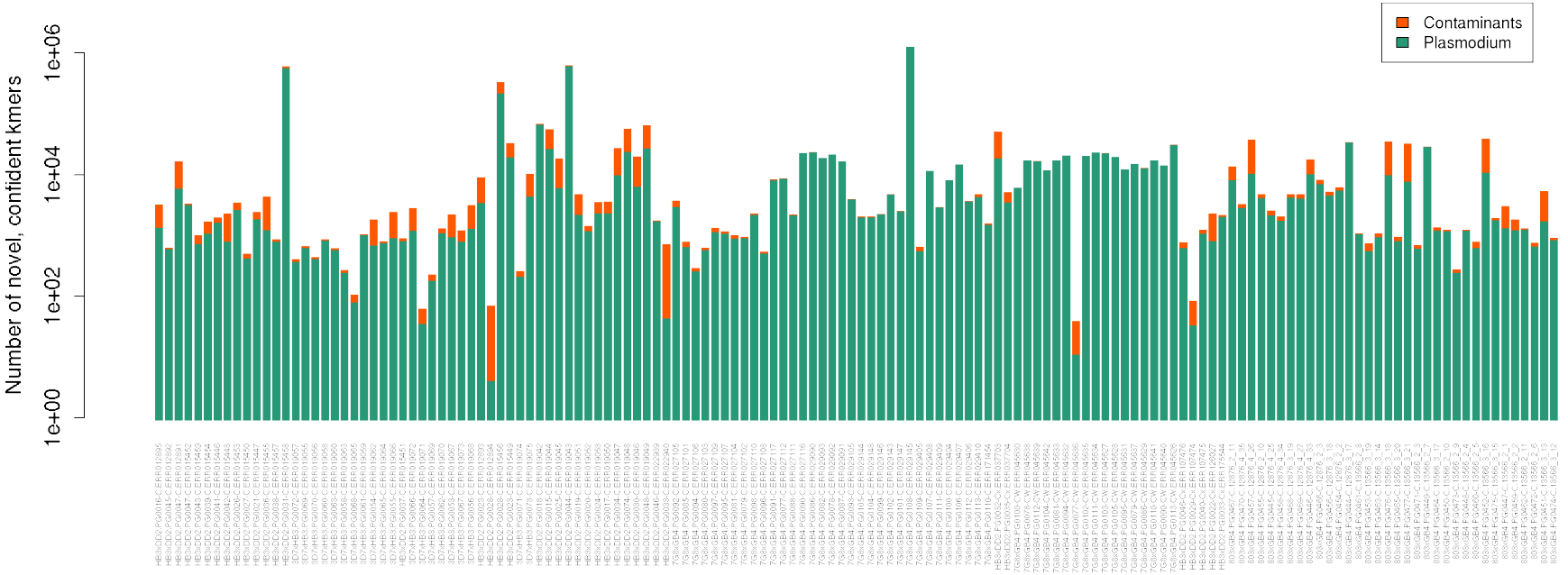

Figure 4.8: Removal of contaminating kmers; *P. falciparum* kmers are shown in green; the putative contaminants are shown in orange.

Table 4.2: ROC metrics on simulated realistic data

sn	sim	fp	fn	tp	tn	rc	sens	spec	prec	npv	fpr	fnr	fdr	acc
5	realistic	4	2	123	77566161	7	0.9840	1	0.9685	1	0	0.0160	0.0315	1
0	realistic	1	1	101	77516712	7	0.9902	1	0.9902	1	0	0.0098	0.0098	1
6	realistic	2	1	107	77289511	2	0.9907	1	0.9817	1	0	0.0093	0.0183	1
13	realistic	0	1	107	77362132	3	0.9907	1	1.0000	1	0	0.0093	0.0000	1
17	realistic	3	1	109	77390562	5	0.9909	1	0.9732	1	0	0.0091	0.0268	1
12	realistic	2	1	111	77440516	3	0.9911	1	0.9823	1	0	0.0089	0.0177	1
16	realistic	1	1	155	77369055	5	0.9936	1	0.9936	1	0	0.0064	0.0064	1
2	realistic	4	0	93	77425427	1	1.0000	1	0.9588	1	0	0.0000	0.0412	1
4	realistic	3	0	65	77312705	2	1.0000	1	0.9559	1	0	0.0000	0.0441	1
3	realistic	6	0	150	77296380	1	1.0000	1	0.9615	1	0	0.0000	0.0385	1
1	realistic	1	0	133	77509177	5	1.0000	1	0.9925	1	0	0.0000	0.0075	1
7	realistic	4	0	56	77445270	1	1.0000	1	0.9333	1	0	0.0000	0.0667	1
8	realistic	6	0	126	77469034	3	1.0000	1	0.9545	1	0	0.0000	0.0455	1
9	realistic	2	0	95	77503142	2	1.0000	1	0.9794	1	0	0.0000	0.0206	1
10	realistic	1	0	124	77398963	2	1.0000	1	0.9920	1	0	0.0000	0.0080	1
11	realistic	2	0	108	77422535	2	1.0000	1	0.9818	1	0	0.0000	0.0182	1
14	realistic	6	0	150	77355056	7	1.0000	1	0.9615	1	0	0.0000	0.0385	1
15	realistic	1	0	125	77435743	9	1.0000	1	0.9921	1	0	0.0000	0.0079	1
18	realistic	4	0	123	77355144	3	1.0000	1	0.9685	1	0	0.0000	0.0315	1
19	realistic	2	0	95	77480228	6	1.0000	1	0.9794	1	0	0.0000	0.0206	1

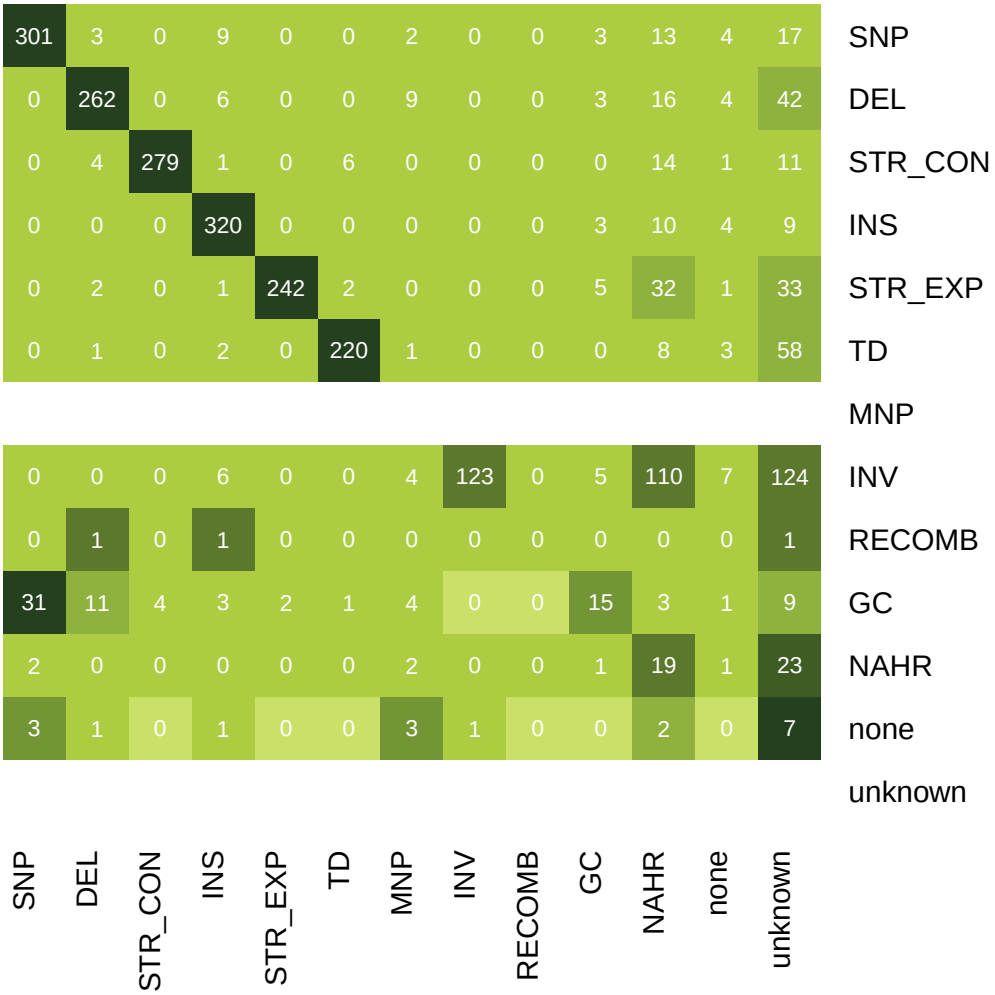

Figure 4.9: Confusion matrix for observed events (below) versus expected events (right), in simulated perfect data.

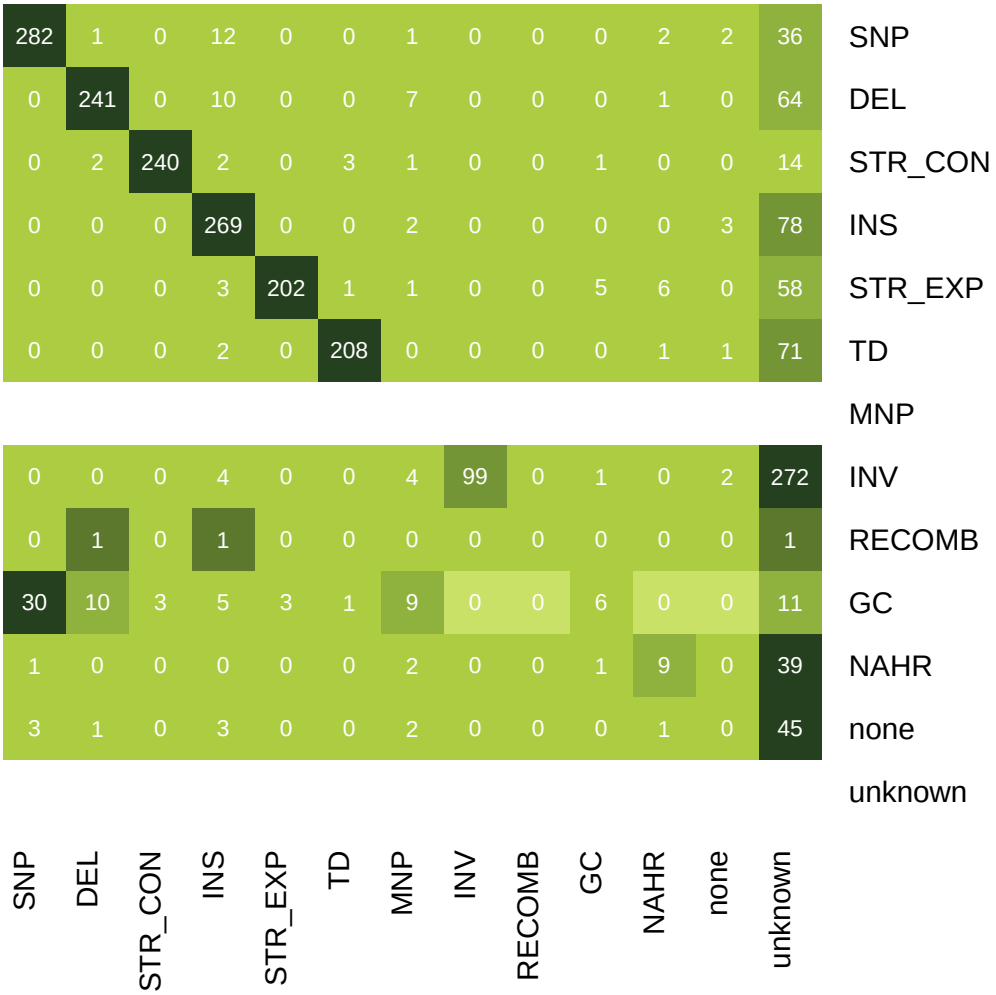

Figure 4.10: Confusion matrix for observed events (below) versus expected events (right), in simulated realistic data.

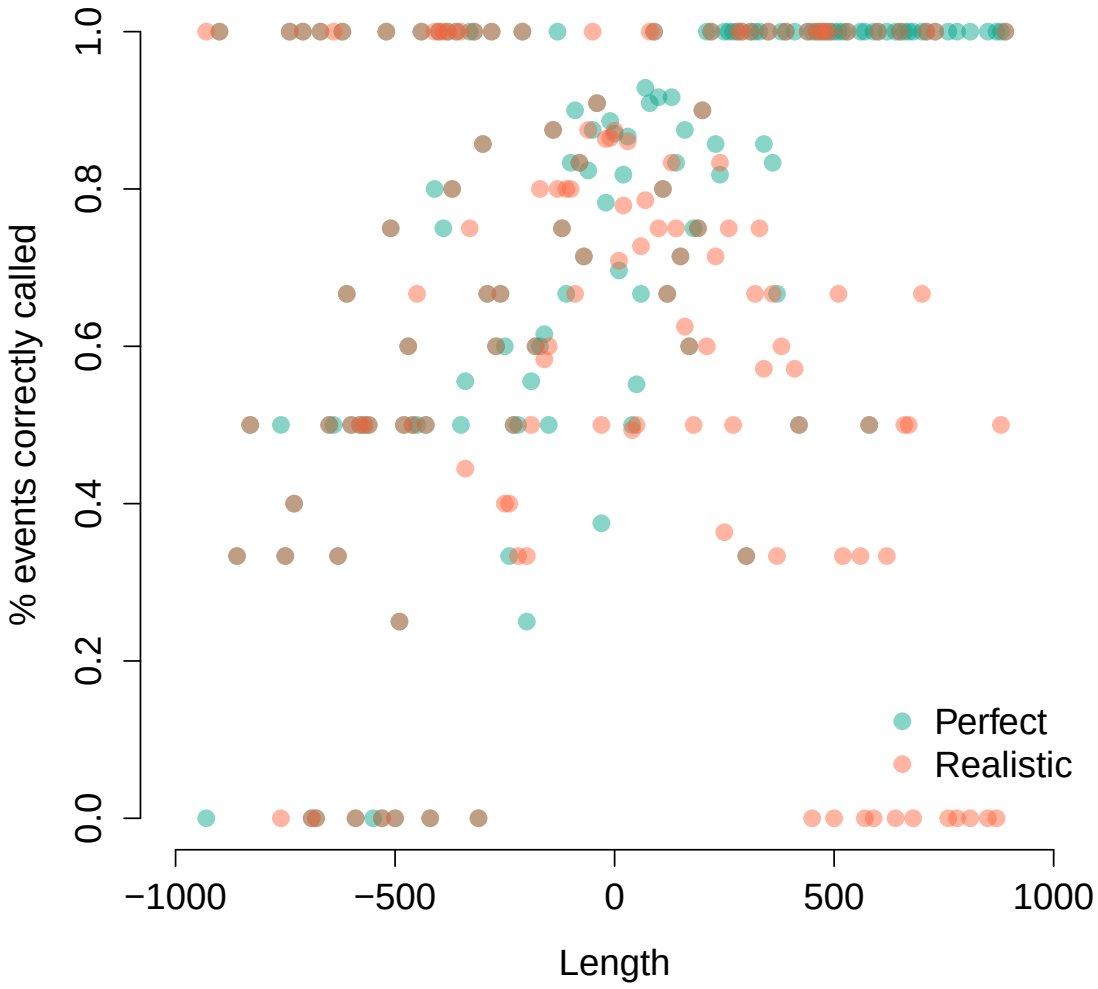

Figure 4.11: Event recovery as a function of event length

5 *Pf*

5.1 *Lit review*

5.1.1 *Review of Kong et al., 2002*

Augustine Kong et al. discuss a new genetic map of recombination rates using genotyping information from 869 individuals in 146 Icelandic families. This is the first such map made after the sequencing of the human genome, and is thus able to leverage the new reference sequence in order to correctly order the genotyped markers. It is a substantially higher-resolution map than provided by the former gold-standard, the Marshfield map. The Marshfield map contained data on only 188 meioses, whereas the Kong et al. map contained data on 1,257. The new map reveals marked differences in recombination rates between males and females (e.g. the recombination rate in female autosomes is a factor of 1.65 higher than that observed in males) for reasons beyond sequence features.

6 Chimp

6.1 Lit review

6.1.1 Review of Kong et al., 2002

Augustine Kong et al. discuss a new genetic map of recombination rates using genotyping information from 869 individuals in 146 Icelandic families. This is the first such map made after the sequencing of the human genome, and is thus able to leverage the new reference sequence in order to correctly order the genotyped markers. It is a substantially higher-resolution map than provided by the former gold-standard, the Marshfield map. The Marshfield map contained data on only 188 meioses, whereas the Kong et al. map contained data on 1,257. The new map reveals marked differences in recombination rates between males and females (e.g. the recombination rate in female autosomes is a factor of 1.65 higher than that observed in males) for reasons beyond sequence features.

7 Discussion

7.1 Lit review

7.1.1 Review of Kong et al., 2002

Augustine Kong et al. discuss a new genetic map of recombination rates using genotyping information from 869 individuals in 146 Icelandic families. This is the first such map made after the sequencing of the human genome, and is thus able to leverage the new reference sequence in order to correctly order the genotyped markers. It is a substantially higher-resolution map than provided by the former gold-standard, the Marshfield map. The Marshfield map contained data on only 188 meioses, whereas the Kong et al. map contained data on 1,257. The new map reveals marked differences in recombination rates between males and females (e.g. the recombination rate in female autosomes is a factor of 1.65 higher than that observed in males) for reasons beyond sequence features.

References

[1] Thomas S Rask, Daniel A Hansen, Thor G Theander, Anders Gorm Pedersen, and Thomas Lavstsen. Plasmodium falciparum Erythrocyte Membrane Protein 1 Diversity in Seven Genomes – Divide and Conquer. *PLoS computational biology*, 6(9):e1000933, September 2010.

[2] World Health Organization. Antimicrobial Resistance, 2014.

[3] C Wang, K Takeuchi, L H Pinto, and R A Lamb. Ion channel activity of influenza A virus M2 protein: characterization of the amantadine block. *Journal of virology*, 67(9):5585–5594, September 1993.

[4] Martha I Nelson, Lone Simonsen, Cécile Viboud, Mark A Miller, and Edward C Holmes. The origin and global emergence of adamantane resistant A/H3N2 influenza viruses. *Virology*, 388(2):270–278, June 2009.

[5] Vegard Eldholm, Gunnstein Norheim, Bent von der Lippe, Wibeke Kinander, Ulf R Dahle, Dominique A Caugant, Turid Mannsåker, Anne T Mengshoel, Anne M Dyrhol-Riise, and Francois Balloux. Evolution of extensively drug-resistant Mycobacterium tuberculosis from a susceptible ancestor in a single patient. *Genome Biology*, 15(11):490, November 2014.

[6] Matthew T G Holden, Edward J Feil, Jodi A Lindsay, Sharon J Peacock, Nicholas P J Day, Mark C Enright, Tim J Foster, Catrin E Moore, Laurence Hurst, Rebecca Atkin, Andrew Barron, Nathalie Bason, Stephen D Bentley, Carol Chillingworth, Tracey Chillingworth, Carol Churcher, Louise Clark, Craig Corton, Ann Cronin, Jon Doggett, Linda Dowd, Theresa Feltwell, Zahra Hance, Barbara Harris, Heidi Hauser, Simon Holroyd, Kay Jagels, Keith D James, Nicola Lennard, Alexandra Line, Rebecca Mayes, Sharon Moule, Karen Mungall, Douglas Ormond, Michael A Quail, Ester Rabbinowitsch, Kim Rutherford, Mandy Sanders, Sarah Sharp, Mark Simmonds, Kim Stevens, Sally Whitehead, Bart G Barrell, Brian G Spratt, and Julian Parkhill. Complete genomes of two clinical Staphylococcus aureus strains: evidence for the

rapid evolution of virulence and drug resistance. *Proceedings of the National Academy of Sciences of the United States of America*, 101(26):9786–9791, June 2004.

[7] D Walliker, I Quakyi, T Wellems, T McCutchan, A Szarfman, W London, L Corcoran, T Burkot, and R Carter. Genetic analysis of the human malaria parasite Plasmodium falciparum. *Science (New York, NY)*, 236(4809):1661–1666, June 1987.

[8] D S Peterson, D Walliker, and T E Wellems. Evidence that a point mutation in dihydrofolate reductase-thymidylate synthase confers resistance to pyrimethamine in falciparum malaria. *Proceedings of the National Academy of Sciences of the United States of America*, 85(23):9114–9118, December 1988.

[9] T E Wellems, L J Panton, I Y Gluzman, V E do Rosario, R W Gwadz, A Walker-Jonah, and D J Krogstad. Chloroquine resistance not linked to mdr-like genes in a Plasmodium falciparum cross. *Nature*, 345(6272):253–255, May 1990.

[10] A B Vaidya, O Muratova, F Guinet, D Keister, T E Wellems, and D C Kaslow. A genetic locus on Plasmodium falciparum chromosome 12 linked to a defect in mosquito-infectivity and male gametogenesis. *Molecular and biochemical parasitology*, 69(1):65–71, January 1995.

[11] T Furuya, J Mu, K Hayton, A Liu, J Duan, L Nkrumah, D A Joy, D A Fidock, H Fujioka, A B Vaidya, T E Wellems, and X z Su. Disruption of a Plasmodium falciparum gene linked to male sexual development causes early arrest in gametocytogenesis. *Proceedings of the National Academy of Sciences of the United States of America*, 102(46):16813–16818, November 2005.

[12] L H Freitas-Junior, E Bottius, L A Pirrit, K W Deitsch, C Scheidig, F Guinet, U Nehrbass, T E Wellems, and A Scherf. Frequent ectopic recombination of virulence factor genes in telomeric chromosome clusters of P. falciparum. *Nature*, 407(6807):1018–1022, October 2000.

[13] Michael F Duffy, Timothy J Byrne, Celine Carret, Alasdair Ivens, and Graham V Brown. Ectopic recombination of a malaria var gene during mitosis associated with an altered var switch rate. *Journal of molecular biology*, 389(3):453–469, June 2009.

[14] E W Myers. Toward simplifying and accurately formulating fragment assembly. *Journal of computational biology : a journal of computational molecular cell biology*, 2(2):275–290, 1995.

[15] Elaine R Mardis. A decade's perspective on DNA sequencing technology. *Nature*, 470(7333):198–203, February 2011.

[16] M C Schatz, A L Delcher, and S L Salzberg. Assembly of large genomes using second-generation sequencing. *Genome research*, 20(9):1165–1173, September 2010.

[17] Paul Flicek and Ewan Birney. Sense from sequence reads: methods for alignment and assembly. *Nature methods*, 6(11s):S6–S12, November 2009.

[18] Rasmus Nielsen, Joshua S Paul, Anders Albrechtsen, and Yun S Song. Genotype and SNP calling from next-generation sequencing data. *Nat Rev Genet*, 12(6):443–451, June 2011.

[19] N Woodford and M J Ellington. The emergence of antibiotic resistance by mutation. *Clinical Microbiology and Infection*, 13(1):5–18, 2007.

[20] Benjamin M Neale, Yan Kou, Li Liu, Avi Ma'ayan, Kaitlin E Samocha, Aniko Sabo, Chiao-Feng Lin, Christine Stevens, Li-San Wang, Vladimir Makarov, Paz Polak, Seungtai Yoon, Jared Maguire, Emily L Crawford, Nicholas G Campbell, Evan T Geller, Otto Valladares, Chad Schafer, Han Liu, Tuo Zhao, Guiqing Cai, Jayon Lihm, Ruth Dannenfelser, Omar Jabado, Zuleyma Peralta, Uma Nagaswamy, Donna Muzny, Jeffrey G Reid, Irene Newsham, Yuanqing Wu, Lora Lewis, Yi Han, Benjamin F Voight, Elaine Lim, Elizabeth Rossin, Andrew Kirby, Jason Flannick, Menachem Fromer, Khalid Shakir, Tim Fennell, Kiran Garimella, Eric Banks, Ryan Poplin, Stacey Gabriel, Mark Depristo, Jack R Wimbish, Braden E Boone, Shawn E Levy, Catalina Betancur, Shamil Sunyaev, Eric Boerwinkle, Joseph D Buxbaum, Edwin H Cook, Bernie Devlin, Richard A Gibbs, Kathryn Roeder, Gerard D Schellenberg, James S Sutcliffe, and Mark J Daly. Patterns and rates of exonic de novo mutations in autism spectrum disorders. *Nature*, 485(7397):242–245, May 2012.

[21] Donald F Conrad, Jonathan E M Keebler, Mark A DePristo, Sarah J Lindsay, Yujun Zhang, Ferran Casals, Youssef Idaghdour, Chris L Hartl, Carlos Torroja, Kiran V Garimella, Martine Zilversmit, Reed Cartwright, Guy A Rouleau, Mark Daly, Eric A Stone, Matthew E Hurles, Philip Awadalla, and 1000 Genomes Project. Variation in genome-wide mutation rates within and between human families. *Nature genetics*, 43(7):712–714, July 2011.

[22] Oliver Venn, Isaac Turner, Iain Mathieson, Natasja de Groot, Ronald Bontrop, and Gil McVean. Strong male bias drives germline mutation in chimpanzees. *Science (New York, NY)*, 344(6189):1272–1275, June 2014.

[23] Wigard P Kloosterman, Laurent C Francioli, Fereydoun Hormozdiari, Tobias Marschall, Jayne Y Hehir-Kwa, Abdel Abdellaoui, Eric-Wubbo Lameijer, Matthijs H Moed, Vyacheslav Koval, Ivo Renkens, Markus J van Roosmalen, Pascal Arp, Lennart C Karssen, Bradley P Coe, Robert E Handsaker, Eka D Suchiman, Edwin Cuppen, Djie T Thung, Mitch McVey, Michael C Wendl, Andre Uitterlinden, Cornelia M van Duijn, Morris Swertz, Cisca Wijmenga, Gertjan van Ommen, P Eline Slagboom, Dorret I Boomsma, Alexander Schönhuth, Evan E Eichler, Paul I W de Bakker, Kai Ye, and Victor Guryev. Characteristics of de novo structural changes in the human genome. *Genome research*, page gr.185041.114, 2015.

[24] Laurent C Francioli, Paz P Polak, Amnon Koren, Androniki Menelaou, Sung Chun, Ivo Renkens, Cornelia M van Duijn, Morris Swertz, Cisca Wijmenga, Gertjan van Ommen, P Eline Slagboom, Dorret I Boomsma, Kai Ye, Victor Guryev, Peter F Arndt, Wigard P Kloosterman, Paul I W de Bakker, and Shamil R Sunyaev. Genome-wide patterns and properties of de novo mutations in humans. *Nature genetics*, 47(7):822–826, May 2015.

[25] Mark A DePristo, Eric Banks, Ryan Poplin, Kiran V Garimella, Jared R Maguire, Christopher Hartl, Anthony A Philippakis, Guillermo del Angel, Manuel A Rivas, Matt Hanna, Aaron McKenna, Tim J Fennell, Andrew M Kernytsky, Andrey Y Sivachenko, Kristian Cibulskis, Stacey B Gabriel, David Altshuler, and Mark J Daly. A framework for variation discovery and genotyping using next-generation DNA sequencing data. *Nature genetics*, 43(5):491–498, May 2011.

[26] Andy Rimmer, Hang Phan, Iain Mathieson, Zamin Iqbal, Stephen R F Twigg, Andrew O M Wilkie, Gil McVean, and Gerton Lunter. Integrating mapping-, assembly- and haplotype-based approaches for calling variants in clinical sequencing applications. *Nature genetics*, 46(8):912–918, July 2014.

[27] Eric S Lander and Michael S Waterman. Genomic mapping by fingerprinting random clones: A mathematical analysis. *Genomics*, 2(3):231–239, April 1988.

[28] Zamin Iqbal, Mario Caccamo, Isaac Turner, Paul Flicek, and Gil McVean. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature genetics*, 44(2):226–232, February 2012.

[29] Malcolm J Gardner, Neil Hall, Eula Fung, Owen White, Matthew Berriman, Richard W Hyman, Jane M Carlton, Arnab Pain, Karen E Nelson, Sharen Bowman,

Ian T Paulsen, Keith James, Jonathan A Eisen, Kim Rutherford, Steven L Salzberg, Alister Craig, Sue Kyes, Man-Suen Chan, Vishvanath Nene, Shamira J Shallom, Bernard Suh, Jeremy Peterson, Sam Angiuoli, Mihaela Pertea, Jonathan Allen, Jeremy Selengut, Daniel Haft, Michael W Mather, Akhil B Vaidya, David M A Martin, Alan H Fairlamb, Martin J Fraunholz, David S Roos, Stuart A Ralph, Geoffrey I McFadden, Leda M Cummings, G Mani Subramanian, Chris Mungall, J Craig Venter, Daniel J Carucci, Stephen L Hoffman, Chris Newbold, Ronald W Davis, Claire M Fraser, and Bart Barrell. Genome sequence of the human malaria parasite Plasmodium falciparum. *Nature*, 419(6906):498–511, October 2002.

[30] C Aurrecoechea, J Brestelli, B P Brunk, J Dommer, S Fischer, B Gajria, X Gao, A Gingle, G Grant, O S Harb, M Heiges, F Innamorato, J Iodice, J C Kissinger, E Kraemer, W Li, J A Miller, V Nayak, C Pennington, D F Pinney, D S Roos, C Ross, C J Stoeckert, C Treatman, and H Wang. PlasmoDB: a functional genomic database for malaria parasites. *Nucleic acids research*, 37(Database):D539–D543, January 2009.

[31] Heng Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. March 2013.

[32] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, and Mark A DePristo. The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. *Genome research*, 20(9):1297–1303, September 2010.

[33] Antoine Claessens, William L Hamilton, Mihir Kekre, Thomas D Otto, Adnan Faizullabhoy, Julian C Rayner, and Dominic Kwiatkowski. Generation of Antigenic Diversity in Plasmodium falciparum by Structured Rearrangement of Var Genes During Mitosis. *PLoS genetics*, 10(12):e1004812, December 2014.

[34] Mark J P Chaisson, Richard K Wilson, and Evan E Eichler. Genetic variation and the de novo assembly of human genomes. *Nature Reviews Genetics*, 16(11):627–640, November 2015.

[35] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models*. Principles and Techniques. MIT Press, 2009.

[36] Petr Danecek, Adam Auton, Goncalo Abecasis, Cornelis A Albers, Eric Banks, Mark A DePristo, Robert Handsaker, Gerton Lunter, Gabor Marth, Stephen T Sherry,

Gilean McVean, Richard Durbin, and 1000 Genomes Project Analysis Group. The Variant Call Format and VCFtools. *Bioinformatics (Oxford, England)*, June 2011.

[37] Luc Devroye. *Non-Uniform Random Variate Generation*. Springer Science & Business Media, New York, NY, November 2013.

[38] Alistair Miles, Zamin Iqbal, Paul Vauterin, Richard Pearson, Susana Campino, Michel Theron, Kelda Gould, Daniel Mead, Eleanor Drury, John O'Brien, Valentin Ruano Rubio, Bronwyn MacInnis, Jonathan Mwangi, Upeka Samarakoon, Lisa Ranford-Cartwright, Michael Ferdig, Karen Hayton, Xinzhuan Su, Thomas Wellems, Julian Rayner, Gil McVean, and Dominic Kwiatkowski. Genome variation and meiotic recombination in Plasmodium falciparum: insights from deep sequencing of genetic crosses. *bioRxiv*, page 024182, August 2015.

[39] Thomas S Rask, Daniel A Hansen, Thor G Theander, Anders Gorm Pedersen, and Thomas Lavstsen. Plasmodium falciparum erythrocyte membrane protein 1 diversity in seven genomes–divide and conquer. *PLoS computational biology*, 6(9), 2010.

[40] Susan M Kraemer and Joseph D Smith. A family affair: var genes, PfEMP1 binding, and malaria disease. *Current Opinion in Microbiology*, 9(4):374–380, August 2006.

[41] John Hopcroft and Robert Tarjan. Efficient algorithms for graph manipulation, 1971.

[42] S F Altschul, W Gish, W Miller, E W Myers, and D J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, October 1990.

[43] Isaac Turner. McCortex Workflow Examples. November 2015.

[44] E W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.