UNIVERSITY OF CALIFORNIA
SANTA CRUZ

INFRASTRUCTURE FOR SCALABLE ANALYSIS OF GENOMIC VARIATION

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

BIOMOLECULAR ENGINEERING & BIOINFORMATICS

by

Adam M. Novak

June 2017

The Dissertation of Adam M. Novak
is approved:

Professor Joshua Stuart, Chair

Distinguished Professor David Haussler

Associate Professor Richard E. Green

Professor Beth Shapiro

Dean **Tyrus Miller**
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Infrastructure for Scalable Analysis of Genomic Variation

by

Adam M. Novak

The scale of the problems which human genomics is asked to solve necessitates that the field develop an ability to integrate and synthesize information across the entire human population. The abstraction of a single-copy human reference genome assembly, and the linear coordinate space that it induces, are more of a hindrance than a help at these scales. They can only ever represent one sample at any given place, and they make combining information about human variation across multiple studies and modalities difficult. To rectify these problems, I propose the construction and adoption of a graph-based alternative to the human reference genome assembly: a Human Genome Variation Map. I present here four research projects. The first is a theory of mapping to references that is extensible to graphs. The second describes a novel data structure for embedding individual haplotype sequences into a graph reference. The third surveys graph construction techniques to discover methods that produce graphs yielding read mapping and variant calling results superior to those obtained with linear, variation-free references. The fourth extends these improvement results to chromosome-scale graphs constructed from multiple sources and modalities of variation data. These four projects describe a research program aimed towards the eventual release of an official Human Genome Variation Map build, providing a piece of vital infrastructure for the analysis of human genomic variation at population scale.

Dedicated to all those who carry human genomes.

Acknowledgments

In addition to my advisor, Prof. David Haussler, and the other members of my committee, I would like to thank Dr. Benedict Paten for helping to direct my research efforts, Erik Garrison for his leadership of the `vg` team, Glenn Hickey for all the software plumbing he has written, Yohei Rosen for his amazing mathematics skills, Jordan Eizenga for his useful algorithms, Charles Markello for his data wrangling, Maciek Smuga-Otto and Sean Blum for their help on the GA4GH bake-off project, Mike Lin for his technical challenges, Jouni Sirén for his stringology expertise, Lynn Brazil, Kelly Sauder, and Tracie Tucker for their administrative assistance, and Microsoft Corporation for their provision of computing resources. I would also like to thank Anna Henderson for her contributions as an editor, a partner, and a friend, and my family for making all of this possible.

The text of this thesis includes reprints of the following previously published or preprint material:

A. M. Novak, Y. Rosen, D. Haussler, and B. Paten. Canonical, stable, general mapping using context schemes. *Bioinformatics*, page btv435, 2015.

In this work, I specifically helped develop the context scheme theory, wrote the majority of the custom software used in the analysis, produced the figures, and contributed substantially to the text of the manuscript.

A. M. Novak, E. Garrison, and B. Paten. A graph extension of the positional Burrows-Wheeler transform and its applications. In M. Frith and C. N. Storm Pedersen, editors, *Algorithms in Bioinformatics*, pages 246–256, Cham, 2016. Springer International Publishing.

In this work, I specifically developed the details of the eponymous graph extension and its algorithms, wrote most of the implementation code, ran the experiments, produced the figures, and contributed substantially to the text of the manuscript.

A. M. Novak, G. Hickey, E. Garrison, S. Blum, A. Connelly, A. Dilthey, J. Eizenga, M. S. Elmohamed, S. Guthrie, A. Kahles, et al. Genome graphs. *bioRxiv*, 2017. doi: 10.1101/101378. URL `http://biorxiv.org/content/early/2017/01/18/101378`.

In this work, I specifically distributed the source data to participants, produced the Camel graph, developed and ran the reference-free evaluation, ran and analyzed the low-coverage read alignments, produced high-coverage alignments for variant calling, developed about half of the variant calling code, made numerous improvements and bug fixes to the `vg` software to facilitate the analysis, and led the production of the manuscript.

The co-authors Benedict Paten and David Haussler listed in these publications directed and supervised the research which forms the basis for this thesis.

How to Read This Document

This document consists of several mostly-independent chapters, with an introduction at the beginning explaining how they fit together. For an introduction to genomics in general and the idea of graph-based genomic references in particular, read the introduction in Chapter 1. To jump directly into a description of context schemes and a formal treatment of mapping to linear references, skip ahead to Chapter 2. For a succinct, efficient method for storing haplotype databases embedded in a graph reference, read Chapter 3. For a demonstration of the utility of graph-based genomic references, including novel evaluation techniques, skip to Chapter 4. Finally, for a description of the construction and evaluation of a large-scale prototype graph reference, and for suggestions for future work, skip all the way to Chapter 5.

Chapter 1

Introduction and Background

1.1 Introduction

The human body is a four-billion-year-old piling-up of nanotechnological hacks, which exists because, for that four billion years, none of its ancestors died without issue; we have hijacked it and are now attempting to use it for our own ends. We don't know how it works, and we don't have the servicing equipment to fix it when it breaks. The goal of human genomics is to figure out how our bodies actually work, so that we can do more than change the nanotechnological oil.

Achieving this goal is relatively difficult, because of the scale of the problem. Each human genome is about 3.2 billion base pairs, and each human has two genomes (one from each gamete). Human brains struggle to think about a even a single billion of anything, a number so large that a one-in-a-million chance occurs on average a thousand times. Add to this the already difficult to comprehend lack of design throughout the system, where no part is *for* anything, and you can begin to get a sense of the difficulty of the problem.

Fortunately, despite our limitations, we are beginning to develop techniques and guiding principles for working with problems at these scales. One of those prin-

ciples is that large problems can be effectively addressed by integrating across large datasets [30]. With a population now exceeding 7.5 billion individuals, humans may constitute a sufficiently large dataset to begin to approach this problem [105]. However, effective tools to use genomic information at the scale of the human population will be required.

Human genomics as it exists today is organized around something referred to as "the human genome", obtained at great expense through the Human Genome Project [88]. There is a clear distinction in the field between the human genome, embodied by the **primary assembly** in reference genome assembly builds produced by the Genome Reference Consortium (GRC) [94], and databases of genomic variation, such as the variant sets produced by the 1000 Genomes Project [3] or the Simons Genome diversity Project [96, 97]. Such efforts generally distribute variation information in Variant Call Format (VCF) files, which present the data in the coordinate space defined by the GRC's assembly. Actual sequencing data is presented in Binary Alignment/Map (BAM) files, which give each sequencing read's alignment to the GRC's assembly. This linear coordinate system, which refers to locations in peoples' genomes by the chromosome and base index in the primary assembly, is the foundation of genomics.

Unfortunately, this approach will not scale, because as our view of the human population becomes wider, our view of human genomic variation is also broadening. The most recent release of the GRC's human reference genome assembly, GRCh38, contains 261 **alternate (alt) loci** [26], up from a mere nine in the previous assembly [25]. Alternate loci are sequences intended to provide alternative versions of parts of other sequences in the primary assembly, in order to be more representative of large-scale and structural variation in the human population [44]. When analyzing or describing a sample with a haplotype closer to one of the alt loci than to the corresponding primary assembly region, that alt locus can be used to fulfill the functions of a reference sequence for that region, replacing part of the primary assembly sequence. The num-

ber of genomes for which this replacement ought to be performed is relatively large. A recent study [42] found that some alternate loci are likely to be present in 90% or more of the individuals in some populations. The eight alternate loci for the Major Histocompatibility Complex (MHC) region were chosen to be representative of people of European ancestry [36]; many analyses of the genomes of people of European ancestry might benefit from using one of those alternate loci in place of what is in the primary assembly. However, available variation datasets [3], and indeed the VCF format itself [12, 66], do not allow for this, and instead cram all individuals into the space of the primary assembly, regardless of its appropriateness for the individuals or populations under study. On the software side, alternate loci support is still a novelty [42].

In genomic regions where these alternate loci apply, the traditional, primary-assembly-based linear coordinate system begins to break down. To properly reason about such genomic regions, we need to abandon either the idea that bases in peoples' genomes correspond to bases in a reference, or the idea that references are linear. Whereas in the previous assembly the problem was relatively contained, in GRCh38 this sudden proliferation of alternate loci is poking holes in that abstraction all over the genome, making the problem much more urgent.

The linear organization of the reference genome also frustrates attempts to study regions of the genome which are difficult to assemble, or which, due to sequence similarity, are very difficult to distinguish from similar regions at other locations in the genome. To facilitate analysis of the centromeres, for example, GRCh38 includes plausible synthetic linear centromere sequences [44]. We have more precise, graph-based models of what we actually know about the centromeres, but these models cannot be indexed by linear sequence coordinates or processed by tools that expect a linear reference sequence [72].

I propose a nonlinear, graph-based "Human Genome Variation Map", or HGVM. This new type of genomic reference will eliminate the artificial distinction

between the reference genome assembly—"the human genome"—and what we know about variation among the genomes of the human population. A graph-based reference can capture in a first-class way the sequence information which is currently relegated to alt loci, as well as additional variant information from sources like the 1000 Genomes Project [3]. If it managed to avoid giving preference to one version of a region with alternate loci over another, a human genome variation map could potentially combat **reference (allele mapping) bias**, an effect in which alleles that match a single linear reference are easier to detect than those which do not [5, 13]. The adoption of a unified representation of genomes will allow genomic analysis software to scale to much larger cross-dataset analyses, with a more representative view of individuals' genomes, allowing progress to be made in the understanding of human biology.

1.2 Background

1.2.1 How Genomics Works

The human reference genome assembly was built at great expense at the turn of the millennium and is maintained by the Genome Reference Consortium (GRC) [8]. This reference genome assembly was originally created by stitching together actual observed pieces of DNA sequence into a single-copy haploid **golden path** representing a complete genome [8]. Under this model, a hypothetical perfect assembly would have a single **contig**, or contiguous linear string of DNA bases, per chromosome. This naturally suggests a coordinate system: bases can be referred to by the contig they are on and their offset from the beginning of that contig.

This coordinate system is a critical piece of genomics infrastructure. It allows the reference genome to be annotated with genes and other elements. It provides the backbone to which descriptions of genomic variation are anchored. It defines the space in which genome sequencing happens, as short reads from sequencing machines are

mapped to positions in this space. The entire field depends on this coordinate system.

Unfortunately, whenever the official human reference genome is updated, and bases are inserted or removed, the old coordinates are no longer valid on the new reference, and a period of mass confusion ensues as everyone who studies human genomics translates everything they are working on over to the new coordinate system, and then wonders whether their colleagues have done the same. Resources that aren't converted to the new system are at best lost to the field, and at worst applied inappropriately to the wrong genomic locations.

The golden path model is inextricably bound to the concept of "the human genome"—the idea that one prototypical set of 24 chromosomes is a suitable foundation for the field of genomics. This idea has been central to human genomics, but it is not without its flaws. Putting aside the unfortunate normative implications of declaring the allele from whomever you sequenced first as "reference" and any alternatives from other populations as "variant", using a single reference genome when mapping sequencing reads leads to the well-known phenomenon of reference bias [5, 13]. Reads matching the reference genome at a variant site tend to map better and more often than those supporting differences from the reference. This reference bias affects many popular short-read aligners [61]. Additionally, in some genomic regions there are dramatically structurally distinct haplotypes present in the population [8]. One example of this phenomenon is the extremely variable Major Histocompatibility Complex (MHC) region on chromosome 6. Mapping reads only against the single haplotype actually included in the assembled golden path will almost certainly make it harder to map reads from other haplotypes.

1.2.2 The Release of GRCh38

A new version of the official human reference genome, GRCh38, was released in 2013 [26]. In addition to marking the transition to a unified version numbering scheme across major

genome browsers, this new release continues the GRC's gradual migration away from the golden path concept. Although GRCh38 is still constructed around a single (chimeric) haploid genome, the new reference assembly also provides sequences for hundreds of so-called **alt loci**—additional pieces of sequence with a specified alignment to that genome which describe some of the structurally distinct local genomic arrangements which have been observed in humans. The older GRCh37, by comparison, contained only three genomic regions with alt loci [8]. This means that the GRCh38 assembly, taken as a whole, is fundamentally nonlinear at more than just a few problematic locations. Unfortunately, popular tools like the Burrows-Wheeler Aligner (BWA), being originally designed for aligning to a single-copy primary assembly, need to apply complex heuristics to account for these alt loci [50, 51].

The new assembly also contains sequence for the centromeres—the central portions of the chromosomes, which contain extremely repetitive sequences that continue to defy conventional sequencing and assembly methods [44]. However, these new centromere sequences are not directly derived from actual sequence observations, but are instead plausible linearizations of a series of graph-based centromere models [72]. Unfortunately, the linear format discards much of the uncertainty information present in the graph models. Moreover, during testing, this additional sequence was found to cause trouble for traditional short-read alignment pipelines, so GRCh38 also comes as an "analysis set" with these sequences masked out [44]. The real problem, though, lies with the tools, which cannot handle either a full nonlinear description of what we know about the centromeres or even the placeholder linearization that GRCh38 includes.

In summary, GRCh38 marks the continuation of a trend towards nonlinearity in the human reference and provides an example of the shortcomings of the golden path approach. Until tools can be updated to account for alt loci and centromere sequences, GRCh38 cannot be used to its full potential.

1.2.3 The Genome Reference Consortium Assembly Model

Containing as it does both the golden-path-style assembled primary chromosomes and an increasing number of alternate loci, the GRC's human genome assembly needs to have a formally organized structure. The assembly is broken down into **units**, each of which contains a set of **sequences** [94]. The most important unit is the **primary assembly unit**, which is also referred to as the "primary assembly" or "primary path". This unit contains the full-chromosome sequences for all of the human chromosomes (1-22, X, and Y), as well as **unplaced scaffolds**, which are pieces of DNA that are thought to be chromosomal but have not yet been associated with a chromosome, and **unlocalized scaffolds**, which are associated with a chromosome but have not been inserted into that chromosome's sequence [94]. The point of the primary assembly unit is to be a complete haploid genome, containing exactly one version of every component [94]. (Note, however, that mitochondrial DNA is relegated to its own **non-nuclear assembly unit**.)

The alternate loci are layered on top of the primary assembly unit in a series of **alternate loci assembly units** [94], which are numbered (`ALT_REF_LOCI_1`, `ALT_REF_LOCI_2`, ...). Each alternate loci assembly unit contains at most one alternate locus for each genomic region having alternate loci. This means that the first alternate loci assembly unit will have the most alternate loci in it, and later units will have fewer, with the total number of units being the same as the number of alternate loci for the region that has the most alternate loci (which, in GRCh38, is the LRC_KIR region) [26, 94]. This data model is designed to be able to represent distinct, linked haplotypes, as is done in the GRC mouse assembly, but for human this capability is not used, and so no significance is assigned to two alternate loci being in the same assembly unit [94].

Finally, as an attempt to work around the disruptive impact of new assembly releases with changed coordinate systems, the GRC assembly model includes a **patches**

assembly unit, containing **patch scaffold** sequences [94]. The patches are divided into two types: **fix patches**, which are intended to replace an erroneous part of a sequence from another assembly unit with an improved, corrected sequence, and **novel patches**, which represent new alternate loci [94]. The GRC uses this patch model so that the coordinates of the other assembly units can be preserved (avoiding disruption and maintaining backward compatibility with existing annotations) while still allowing new alt loci or corrections to existing sequence to be rolled out in a timely fashion (i.e. once per quarter, rather than once every few years) [94].

The regions to which alternate loci belong, and official GRC alignments of the alternate loci to the portions of the primary assembly unit that they are intended to replace, are also part of the assembly model [94]. The region definition mechanism is also used to model the pseudoautosomal regions of the X and Y chromosomes [94].

1.2.4 Modeling Human Genomic Variants with VCF

Unlike in the assembly world, in the world of human variation data there is no master group or resource like the GRC and their assemblies. One particularly large database is dbSNP [95], which works at the level of single variants. There is also the variation data maintained by the GRC, in the form of alternate loci, which are restricted to certain genomic regions. However, the most influential institution in the study of human genomic variation so far has been the 1000 Genomes Project. The 1000 Genomes Project maintains and distributes variant data for over 2,500 people's genomes [104]. However, their greatest contribution to the field may be the development of the extremely popular **Variant Call Format** (VCF), a column-based text file format used to represent variation data, now maintained by the Global Alliance for Genomics and Health [12]. Samples are represented by columns, and polymorphic positions in the human genome by rows. VCF files can be supplemented by an index on genomic position, but no work appears to have yet been done to also provide an index by sample; conse-

quently, the scalability of VCF is currently limited to numbers of samples that can be scanned through efficiently [12]. Moreover, being primarily about a file format instead of a conceptual data model, the VCF specification [66] primarily discusses syntactic considerations, rather than semantic or pragmatic concerns.

VCF encodes individual samples' genomes by defining a series of variant sites along the length of the linear reference genome, defining a set of alternate alleles which have been observed at each site (in addition to the allele in the reference), and then indicating which alleles (in what phasing) are present in each sample at each site. This approach works extremely well for some types of variation, like single nucleotide polymorphisms (SNPs) and short indels in structurally quiet regions, but it also has shortcomings.

One problem with the VCF format is it does not define the semantics of the absence of a variant record. Does it mean that that position in the reference is known not to be variable in the population (or at least in the sampled portion of it)? Or does it mean that that location is not in the region covered by the VCF file? To solve this problem, the VCF format has been extended by Illumina to create the gVCF format, in which genotyped but nonvariant positions are also described [92].

Another potential problem with the VCF format, at least from the point of view of people who need to read it, is that it is very featureful. The format is extensible, through the inclusion of header lines defining various fields. However, different VCF processing tools need to have different sets of fields defined in order to work, and some tools or datasets [104] will use extra fields to modify the interpretation of standard fields defined in the VCF specification [66]. To avoid crashes and to ensure that variant calls are interpreted as the caller intended, it is vital to check the fields output by one tool against those accepted by another. This makes VCF a worse standard, because any two tools that both use VCF cannot necessarily communicate with each other, and because communication failures can be silent if the two tools agree mostly but not perfectly.

Furthermore, there are no fewer than three distinct syntaxes for specifying variants in VCF: the original syntax, in which alternate alleles are short stretches of sequence; a symbolic format, in which alternate alleles are mere specifications of inversion or duplication, or even references to named alleles defined elsewhere; and a breakend-based format, in which alternate alleles describe **breakends**, or points at which the reference would have had to have been cut and spliced to produce the sample [66]. Available VCF parsers do not help with integrating across or converting between these different internal formats, and some don't even support all of them. Tools written to directly extract information from VCFs without a parser library often support only one or maybe two of these formats. Between the three different formats and the fact that different alignment parameters can induce variant callers to describe the same observed sequence as different variants, it is very difficult to compare two VCF files.

Finally, the VCF format is tightly coupled to the linearity of the reference genome assembly. While VCF's breakend system allows the specification of complex rearrangement graphs for samples, there is no explicit support for even the alt loci of the current GRCh38 reference. For example, if one were to specify variant records on one of the MHC alt loci, there would be no way to specify phasing with respect to variants on the main chromosome 6, because VCF specifies records with different chromosome names to be unphased relative to each other [66]. Furthermore, there is no way to explicitly specify that a sample uses a certain alt locus; it would be necessary to infer this from the existence of called genotypes in the coordinates of that alt. It would certainly be possible to adopt certain conventions within the existing VCF format to work around this problem—for example, we could wire the alt loci into their parent chromosomes with breakends whenever they are present. However, no such conventions are standardized for data interchange.

A graph-based approach to the description of genomic variants could alleviate several of these problems, defining explicitly when an individual matches the primary

assembly, and expressing clearly and concisely the alt loci that an individual carries, and any variations on top of them, in a single sufficiently general syntax.

1.2.5 The 1000 Genomes Project

Besides the development of the VCF standard, the 1000 Genomes Project has also conducted one of the most useful publicly available surveys of human genomic variation to date [3]. The 1000 Genomes Project dataset is more consistently collected and analyzed than data from the Personal Genome Project (PGP) [9]. It is also easier to download than the Simons Genome Diversity Project dataset [97], requiring neither special software nor manual approval. For these reasons, the 1000 Genomes Project dataset is the standard human variation dataset to use in analyses.

The dataset contains genomes from 2,504 people across five **superpopulations** and 26 populations [3], as shown in Table 1.1. Sample collection goals and informed consent practices were broadly similar to those developed for the HapMap project, from which some of the initial samples were obtained [40]. In particular, the seductiveness of the three-letter-code abstraction notwithstanding, the goal is not to build a system or hierarchy of racial or ethnic categories for dividing up the human population [40]. Rather, the avowed intention is to describe where and from whom sample DNA was collected, although this necessarily relies on an implicit system of racial and ethnic categories.

1.2.6 Substring Search with the Suffix Array

The **suffix array** of a string is an array of indexes into the string, sorted in the lexicographical order of the suffixes that they point to [64]. For example, the string "dog" has suffixes "dog" at index 0, "og" at index 1, and "g" at index 2, so its suffix array would be $[0, 2, 1]$, corresponding to the suffix sort order ["dog", "g", "og"]. Another example suffix array for the string "GATTACA" is visible in the leftmost column of Figure 1.1.

Superpopulation	Population	Provided Description
AFR	ESN	"Esan in Nigeria"
	GWD	"Gambian in Western Division, Mandinka"
	LWK	"Luhya in Webuye, Kenya"
	MSL	"Mende in Sierra Leone"
	YRI	"Yoruba in Ibadan, Nigeria"
	ACB	"African Caribbean in Barbados"
	ASW	"People with African Ancestry in Southwest USA"
AMR	CLM	Colombians in Medellin, Colombia"
	MXL	"People with Mexican Ancestry in Los Angeles, CA, USA"
	PEL	"Peruvians in Lima, Peru"
	PUR	"Puerto Ricans in Puerto Rico"
EAS	CDX	Chinese Dai in Xishuangbanna, China"
	CHB	"Han Chinese in Beijing, China"
	CHS	"Southern Han Chinese"
	JPT	"Japanese in Tokyo, Japan"
	KHV	"Kinh in Ho Chi Minh City, Vietnam"
EUR	CEU	Utah residents (CEPH) with Northern and Western European ancestry"
	GBR	"British in England and Scotland"
	FIN	"Finnish in Finland"
	IBS	"Iberian Populations in Spain"
	TSI	"Toscani in Italia"
SAS	BEB	Bengali in Bangladesh"
	GIH	"Gujarati Indians in Houston, TX, USA"
	ITU	"Indian Telugu in the UK"
	PJL	"Punjabi in Lahore, Pakistan"
	STU	"Sri Lankan Tamil in the UK"

Table 1.1: Populations and superpopulations from the 1000 Genomes Project [3].

Suffix arrays have some useful properties. Most importantly, all of the suffixes that start with the same substring appear in a single contiguous block [22]. This block starts at the position corresponding to the number of occurrences of lexicographically smaller substrings of the same length [22]. This is particularly obvious in the case of single-character substrings: all the suffixes (and, thus, all the substrings) beginning with a certain character appear in one block, coming immediately after all suffixes beginning with lexicographically smaller characters.

Suffix arrays can be used as indexes to speed up substring search on the string they are derived from. Because of the block structure described above, and because every instance of a substring is at the beginning of some suffix, a simple binary search is sufficient to find any substring that is present, and a scanning up and down from one instance can pull out the entire corresponding block [64]. Supplementing the suffix array with a **longest common prefix (LCP)** array, holding the length of the prefix shared by each pair of adjacent suffixes, can further speed up the search, by requiring only a single-character comparison (instead of a string comparison) at each search step [64].

1.2.7 String Compression with the Burrows-Wheeler Transform

Human genomes, being extremely similar to each other and relatively similar to themselves in different places, lend themselves to compression. One particularly useful algorithm in string compression is the **Burrows-Wheeler Transform (BWT)**. The BWT takes strings and rearranges them for increased compressibility, by putting characters from similar contexts near each other [6]. (It is interesting to think of the BWT as defining a new, context-based coordinate system.)

The BWT operates by taking the string to be compressed (with a sentinel value "$" lexicographically smaller than all other characters appended to it) and imagining all possible rotations of it [6, 22]. Each rotation is derived from the previous one by taking the first character and moving it to the end [6]. The rotations are then sorted

lexicographically, and the last characters of all the rotations become the transformed string [6].

The BWT makes strings more compressible by grouping characters by the contexts they appear in (specifically, the strings they appear before). If two characters both appear before a suffix starting with "andy", they will appear near each other in the BWT. Assuming some letters are more likely, relative to the overall frequency distribution, to precede this string than others are (for example, "c" and "h" as opposed to "e" or "n"), this creates a region of the BWT which is enriched for those characters. This enrichment makes the region more compressible by move-to-front or even simple run-length encoding [6].

The implied BWT matrix, with all the sorted rotations as rows, is generally not kept, but it is often useful to consider the BWT string in its context as the last column of that matrix [6, 22]. It is also useful to think of this matrix as being made up of "character instances"; characters in the matrix that are derived from the same position in the original string are the same character instance. (Imagine uniquely numbering the character at each position on the original string before creating the matrix.) Such a matrix is visible in Figure 1.1.

We can show that, for each character in the alphabet, corresponding instances of that character will appear in the same order in the first column and in the last column. Consider just the rows where the character in question appears in the first column. When sorting these rows, the first column is uninformative (since it is constant across all rows), and the rows are sorted lexicographically by the remaining columns in order. Rotating all the strings so the uninformative column appears last will not change the order of the other columns, and thus will not change the relative sort order of the rows we are considering. Thus the instances of the character stay in the same relative order in the last column as in the first column [46].

$ 7	G 0	A 1	T 2	T 3	A 4	C 5	A 6
A 6	$ 7	G 0	A 1	T 2	T 3	A 4	C 5
A 4	C 5	A 6	$ 7	G 0	A 1	T 2	T 3
A 1	T 2	T 3	A 4	C 5	A 6	$ 7	G 0
C 5	A 6	$ 7	G 0	A 1	T 2	T 3	A 4
G 0	A 1	T 2	T 3	A 4	C 5	A 6	$ 7
T 3	A 4	C 5	A 6	$ 7	G 0	A 1	T 2
T 2	T 3	A 4	C 5	A 6	$ 7	G 0	A 1

Figure 1.1: An example BWT matrix for the string "GATTACA". The sentinel value "$" is appended to the end of the string, all rotations of the string are calculated, and the rotations are sorted. Bases are colored according to base identity and numbered according to position in the original string. The characters in the far right column are the BWT of the original string, while the numbers in the far left column are the suffix array (represented as indexes into the original string).

1.2.8 Searching in BWTs with the FM-index

Constructing the BWT matrix is essentially the same task as constructing the suffix array of the string being transformed. All the rotations of the string contain the "$" sentinel which is lexicographically less than all other characters. Thus the rotations are actually sorted by the portion before the "$" character—that is, by the corresponding suffixes of the original string. This is the same sort used to construct the suffix array.

A BWT can be augmented with a small amount of additional information to

create an **FM-index** (named after the initials of its inventors), which, like a suffix array, allows efficient substring search on the original string, but which also retains the compression afforded by the BWT [22]. The FM-index is based primarily on the idea of a **last-first (LF)** mapping. This mapping maps each character instance in the last column of the BWT matrix to the row in which that same character instance appears in the first column. Because each BWT matrix row is a rotation of the original string, the last column of the row will contain the character instance immediately preceding the one just looked up. Thus, following the LF-mapping around the BWT from any starting position allows the characters of the original string to be enumerated from there in reverse order [22].

Since only the last column of the BWT matrix (i.e. the actual BWT string) is used in the algorithm, only that string needs to be stored. Furthermore, the LF-mapping can easily be calculated from the BWT string. To LF-map the character instance at a certain index in the BWT, count up the number of characters in the BWT lexicographically less than the character, and add the character instance's rank among all instances of that character. This gives the index of the LF-mapping result in the BWT.

To see why this works, recall that in a suffix array, and thus also in the BWT matrix, all the suffixes (or here rotations) that start with a given character form a contiguous block, coming just after all those beginning with smaller characters. Thus, the first calculation is to find the start of this block. And since, as shown in Subsection 1.2.7, the relative order of character instances in the first column is the same as that in the last column, to find the offset of this particular character instance in that contiguous block, we merely need to find its rank among instances of the same character in the last column, which is the BWT string [46].

We can now define **backward search**, a search algorithm using the BWT which processes the characters in the query string from back to front. The algorithm

begins by selecting the entire BWT matrix, which is the range of suffixes that begin with the empty string. Then, for each character in the query string, from the last forwards, the algorithm extends the searched string at the front with that character. It takes the new character and finds the first and last instances of it in the BWT contained within the currently selected result range. It then LF-maps each of those instances, and takes the range between them as the new result range for the query string extended with that character. If there are no instances of the character to map, then the searched string is not found in the index [22].

Each row of the BWT matrix in the old range started with an instance of the old query string. Each of the rows that ended in the new query character corresponded to an instance of the old query string occurring after the new query character, and thus each implies an instance of the new, one-character-longer query string. The LF-mapping step finds the contiguous block of rows in the BWT matrix where those instances of the search string appear, the boundaries of which correspond to the first and last instances of the new character in the old BWT range (by the conservation of ordering mentioned at the end of Subsection 1.2.7). Thus, such an algorithm can be used to search for substrings in a string, using the BWT of the string [22].

By pre-calculating some auxiliary data structures, such as a table with the start index of each character's range in the BWT matrix, and by using succinct data structures for $O(1)$ rank queries, this algorithm can be made to run in time linear in the length of the query string, and constant in the length of the index [22]. Furthermore, using a downsampled copy of the suffix array, the location of each result in its source string can be calculated efficiently [101].

1.2.9 Bidirectional DNA Search with the FMD-Index

BWT-based indexes have found many applications in genomics, mostly due to their ability to efficiently search for and identify the locations of a substrings in very large

datasets—with a few modifications, this search can be extended to align reads to a reference [50]. The popular short read aligner BWA, for example, is built on an FM-index of the reference genome; indeed, the name stands for "Burrows–Wheeler Aligner" [50, 53]. The "String Graph Assembler" SGA also uses a BWT-based index to do its work, but in this case indexes reads themselves [99].

In genomics, the strings being indexed are DNA strings, consisting of As, Gs, Cs, and Ts. These DNA strings are usually excerpts from double-stranded DNA genomes, in which, for each chromosome, two strands of DNA form a double helix. One strand runs in one direction, and the other strand runs in the other direction, with bases complemented (As and Ts swapped, and Gs and Cs swapped). It's impossible to tell whether a DNA sequencing read came from the forward strand or the reverse-complement strand until a match is found for it in a reference somewhere. Thus, many analysis problems in genomics need to consider not only some set of DNA strings but also their reverse complements.

The existence of reverse complements is accounted for in SGA by creating two FM-indexes of the input data: one index of the forward strand, and one of the reverse-complement strand [99]. This construction requires DNA query strings to be searched against both indexes, and the results combined. However, there is a more elegant approach which allows the same search to be performed against a single index, and moreover allows bidirectional extension of the query string. This data structure, the **FMD-index** (the "D" is for "DNA"), is simply an FM-index of both the forward and reverse strands of all input sequences, concatenated into a single dataset [48].

The FMD-index provides for double-ended search; that is, an intermediate search result can be extended with a character on either the left or right end of the query string. This works by having the FMD-index store as its intermediate result not just the single range in the BWT corresponding to BWT matrix rows that start with the query string, but also the (equally long) range for the reverse complement

of the query string [48]. The first is the **forward range** and the second the **reverse range**. The fact that these two intervals will always be equally long is the key to the algorithm: because each string in the index is present as both itself and its reverse complement, any appearance of the query string has a corresponding appearance of its reverse complement. Extending the query string on the left causes the forward range to jump around in BWT coordinate space (to the regions of the BWT matrix that begin with the newly added character). However, extending on the left always causes the reverse range to cover a subrange of what it covered previously: the reverse complement of the query string gets extended on the right, and only BWT matrix rows which began with the original reverse-complement query string can possibly also begin with the longer reverse-complement query string.

The FMD-index search algorithm works as follows: When the query string is extended on the left, the forward range is updated as normal. The reverse range takes on the new interval length derived from the forward range, and a small dynamic programming problem is done over the alphabet to find its new start position. The dynamic programming problem is fairly simple because the reverse range can be partitioned into the ranges that would be selected upon left-extension with any character, ordered in lexicographic order by the reverse complement of the character. The dynamic programming simply consists of looping through the alphabet in lexicographic order by reverse complement, considering extending on each character up to the one actually being used, calculating how long the result set would be on the forward strand, and adding that in to the start of the reverse strand interval [48]. To extend a string on the right, the forward and reverse ranges are temporarily swapped, and the reverse complement of the query string is extended on its left with the reverse complement of the new base [48].

1.2.10 Sequence Graphs

There are many possible representations of a genomic reference as a graph [11], but one particularly useful model is a **bidirected graph**, or, when used to represent genomic data, a **sequence graph** [84]. In the sequence graph model, nodes in the graph are nucleic acid **sequences**, and each sequence has two **sides**: a "left" or "start" side and a "right" or "end" side. The two sides of a sequence are **opposites** of each other, and can be written as s and $\overline{s}$. The sequences are connected together by **edges**, each of which has two ends that are attached to sides of nodes. The model is called "bidirected" because, unlike in a directed graph where each edge consists of a set of nodes and a direction (from one node to another), in a bidirected graph each edge consists of a set of nodes and two directions. The edge can still be from one node to another (in which case it connects the end side of one node to the start side of the other), but it can also be "to" both nodes (in which case it connects their start sides together), or "from" both nodes (in which case it connects their end sides together).

In a graph such as this, traversals, walks, and other graph-theoretic concepts generalize from visiting just the nodes to visiting nodes in one of two orientations: **forward** (i.e. start side to end side) or **reverse** (i.e. end side to start side). A visit to a node in the forward orientation correspond to the node's sequence, whereas a visit to the node in the reverse orientation corresponds to the reverse complement of its sequence. When visiting multiple nodes, it is important that the visits' orientations be consistent: if two nodes are connected by an edge from the first to the second, and you visit the first node in its forward orientation, you may next visit the second node in its forward orientation (by leaving the first node's end and arriving at the second node's start), but you may not, traversing that edge, visit the second node in its reverse orientation. In other words, the forward orientation corresponds to arriving at the start side and leaving via the end side, whereas the reverse orientation corresponds to arriving at the

end side and leaving via the start side, and other combinations (such as both arriving and leaving via the start side) are not permitted [4].

In addition to the bidirected graph formulation, there exists an equivalent **biedged graph** formulation [85]. A sequence graph can be modeled as a graph with two types of edges: **sequence**, or "black", edges, and **join**, or "gray", edges. In this formulation, each side becomes a node, connected by a sequence edge labeled with the sequence that the side belongs to. When traversing the sequence edge in one direction, the sequence is read forward, while when traversing the sequence edge in the other direction, the sequence is read in reverse. Every node has exactly one sequence edge attached to it. The join edges connect nodes, just as the edges in the bidirected graph model connect sides; join edges are undirected.

1.2.11 Graph Substring indexes

A graph-based Human Genome Variation Map requires an efficient substring search algorithm, in order to allow sequencing reads to be efficiently aligned to the graph. Substring search in graphs is not a new idea. Many of the current approaches to this problem come at it from the perspective of trying to index a multiple sequence alignment [102]. Several such approaches are described below.

One approach, the Generalized Compressed Suffix Array (GCSA), extends the XBW transform (itself a generalization of the BWT to trees) to "prefix-range-sorted automata", which include de Bruijn graphs but not general directed labeled graphs [102]. However, the authors of that approach present only an implementation for acyclic multiple sequence alignments. The existence of nonlinear structures like polymorphic inversions, where a genomic region is forwards in some individuals but backwards in others, is not addressed, and no implementation for de Bruijn graphs is provided [102]. Moreover, the approach presented there provides search over all possible paths through the graph in question, which is a reasonable choice for indexes

derived from multiple alignments, but which might backfire for graphs with short cyclic structures that could provide pathological productions for many query strings [102].

Another, slightly newer approach uses the concept of a "population reference graph", also derived from a multiple sequence alignment [16]. In contrast to the BWT-based indexing methods described above, the population reference graph method turns its graph representation of genomes into a Hidden Markov Model (HMM), and identifies the most likely paths through it to match the k-mer spectrum of any particular sample [16]. Under this method, a pair of haploid genomes are then synthesized as sample-specific references, and existing read-to-genome mapping tools are used to map sequencing reads to these references [16]. Unfortunately, because of the way that k-mer counts from a sample are divided up to provide input for the HMM model in different genomic regions, this method is forced to divide its HMM states into "levels" that it proceeds through in a fixed, sequential order. The resulting graph model is constrained to closely resemble the multiple sequence alignment it was derived from. While this method can effectively model a wide range of alternative sequences in a region, it does not appear to be able to effectively model inversions, duplications, or other more complex structures [16].

A modern implementation of a more traditional substring index is presented in Maciuca et al. [63]. Known as the **vBWT**, with the "v" representing variation, it encodes a graph consisting of a reference backbone and a series of non-overlapping, potentially length-changing substitutions along that backbone, by demarcating the variable locations and the different alleles with special characters in a single long string, which is then compressed and indexed using the BWT [63]. These special characters are integrated into the standard backward search algorithm, splitting the selected range when required, to enable search over substrings in the graph using the BWT of the encoded string. This is an elegant and relatively simple approach, and its model is well-matched to that of the VCF format, which also deals with nonoverlapping replace-

ments along a linear reference. Unfortunately, it is even more restrictive than GCSA in terms of the requirements it imposes on the graph; one important shortcoming is that it cannot represent nested variation, such as SNPs inside of potential indels Maciuca et al. [63], Sirén et al. [102].

One final graph substring index is GCSA2, an improved version of GCSA that makes some different design decisions [100, 102]. Rather than restricting eligible graphs to prefix-range-sorted automata and indexing them in their entirety, GCSA2 accepts general directed labeled graphs (which can be generated from bidirected graphs by doubling out the nodes) [100]. The limitation instead is on the query length; GCSA2 works on an index mapping each k-mer to the positions in the graph at which copies of it start. The table of k-mers is compressed by representing it as a generalization of a de Bruijn graph (so that shared pieces of sequence between adjacent k-mers do not need to be duplicated), and then that graph is in turn represented using succinct self-index techniques designed for de Bruijn graphs. Search is accomplished by using BWT-based techniques on the index graph, and then following references from that graph to the relevant locations in the full graph being indexed [100].

1.2.12 Data Models with Protobuf

Representing human genomic variation as a graph reference requires a data model in which to represent that graph in software. Moreover, producing an effective Human Genome Variation Map that can actually be used by other researchers requires selecting a data model that can itself be easily communicated, that can have broad software support, and that people can be persuaded to agree on.

A simple way to describe a data model and get code generated in various languages for free is to use Google's Protocol Buffers (or "Protobuf") library [111]. The system provides a simple language to describe data structures, and a serialization system to allow multiple languages to read and write those data structures. The availability

of libraries such as "json2pb" (`https://github.com/shramov/json2pb`) permits easy interoperation with any language or tool that can consume or produce JSON. Additionally, the relative sparsity of features forces data models to be relatively simple, and the backing of a large, rich technology company helps convince people (for good or bad reasons) to agree on the format. Finally, the extensibility of the system allows new fields to be added to provide new features without invalidating older datasets.

An example Protobuf description of a data model for nodes in a genome graph is visible in Figure 1.2. Using a Protobuf-based data format for genome graphs allows for broad accessibility, without some of the disadvantages (such as large size and the necessity to write correct parsers in various languages) of bespoke text formats.

```
// *Nodes* store sequence data.
message Node {
    string sequence = 1;    // Sequence of DNA bases represented
       ↪ by the Node.
    string name = 2;  // A name provides an identifier.
    int64 id = 3;     // Each Node has a unique positive
       ↪ nonzero ID within its Graph.
}
```

Figure 1.2: Protobuf-format data model for a graph node, from `vg` [24]. Each kind of item in the data model is referred to as a "message", and each field is manually assigned a unique identifying number to allow its name to be changed later while retaining backward compatibility.

1.2.13 `vg`, the Variation Graph Toolkit

One collection of Protobuf data models for genome graphs comes from `vg`, a software suite created by Erik Garrison for working with genome graphs [24]. In `vg`, graphs are represented by a collection of nodes, each of which has an ID and a sequence, and a collection of edges, each of which connects one side (the start or end) of one node to one side of another node (which may actually be the same node, because the `vg` model

allows for cycles and self-loops). Each graph can also have a series of named paths associated with it, to represent how notable sequences, such as the primary path of a genome assembly, or the reference version of a particular gene, fit into the graph.

The `vg` suite is structured as a command-line `vg` command with a variety of subcommands (`vg construct`, `vg map`, `vg view`, etc.), which are designed to be chainable into pipelines and which communicate using streams of Protobuf-serialized graph data. The combination of the Protobuf-based serialization format, the modular architecture as a collection of subcommands, and the relatively comprehensive internal graph manipulation API make `vg` an attractive option as a framework in which new genome graph algorithms can be implemented, and a useful toolkit for performing graph-based analyses.

At the time when `vg` was selected as a basis for further development, it provided a data model supporting bidirected graphs, subcommand implementations for constructing, indexing, and mapping to directed acyclic graphs, and a succinct graph storage format. In part as a result of software development work undertaken for this thesis, `vg` now includes full support for working with bidirected graphs (with GCSA2-based indexing [100] and succinct graph storage), a variant calling implementation, and a suite of unit tests.

1.2.14 Copy-Number-Variable Alignments with Cactus

When building a Human Genome Variation Map, it would be desirable to be able to incorporate variation data in the form of observed sequences, such as the European-representative MHC sequences obtained in Horton et al. [36], or the many alt loci provided with GRCh38 [44]. Thus, it is necessary to have a mechanism to go from a collection of related sequences to a graph representation describing their commonalities. The `vg` suite includes a tool designed to do this, `vg msga` (which stands for "multiple sequence graph alignment"), but this tool remains under active development.

An alternative approach is to use a more mature multiple aligner tool called Cactus [82]. The Cactus aligner, which has been deployed in production for the production of community alignment resources [37], was designed for alignment problems involving large numbers of whole genomes from different species, and consequently understands the sorts of structural changes between species' genomes, such as large-scale deletions, duplications, and rearrangements, that need to be dealt with when working at those evolutionary time scales. In regions like the MHC, which are prone to **incomplete lineage sorting (ILS)** [89], interspecies evolutionary distances can exist within a single individual. Consequently, an aligner that is able to work well at such time scales might be expected to work well on the MHC alt loci and on other alt loci throughout the human genome assembly.

Cactus output typically takes the form of a Hierarchical Alignment Format (HAL) file [35, 37], which uses a phylogenetic tree with internal ancestor nodes to structure information about how blocks of different genomes relate to each other. The structure formed by relationships between corresponding blocks in different genomes is quite similar to a sequence graph, with the genomes being embedded in it as paths, and so Cactus-based alignments have a natural conversion to sequence graphs.

1.2.15 Finding Variable Sites in Graphs

The VCF format, at least in two of its three syntaxes, reifies the idea of a **variant**, which is a potential replacement of part of the linear reference genome assembly with alternate alleles form a collection of options [66]. A well-defined notion of a **variable site**, like the VCF variant, is a useful abstraction, but unfortunately it is difficult to carry over to the graph context. Sequence graphs intended to provide natural representations for nested variation—such as SNPs within potentially-deleted regions—but those structures are awkward to describe as single monolithic variable sites. Moreover, in a graph, it does not necessarily make sense to restrict the model to only accommodating differences

from the path taken by the reference genome assembly; novel material not present in the reference assembly's primary path also ought to be able to host variable sites.

One notion of a variable site in a graph context is the **ultrabubble** [85]. In terms of the biedged graph representation, an ultrabubble is a subgraph defined by two distinct **boundary sides** in it, such that:

1. The opposites of the boundary sides are not in the subgraph.
2. Removing the sequence edges of the boundary sides disconnects the subgraph from the rest of the graph, but leaves the subgraph connected.
3. Neither one of the boundary sides could be replaced with another side in the subgraph while still meeting the above conditions.
4. The subgraph is acyclic (i.e. when following valid walks, the same side cannot be reached twice).
5. Every side in the subgraph has an incident join edge.

These conditions define a subgraph, bounded at both ends by a single node, which can be traversed from one end to the other in one or more ways. Such a structure can be used to provide a graph-based notion of a variable site, with the boundaries giving its "location" in the graph, and the potential traversals from one boundary to the other replacing the VCF variant's reference and alternate alleles.

Note that ultrabubble can nest: both of the boundary sides can be replaced at once to define a smaller **child** ultrabubble [85]. If you consider child ultrabubbles to be opaque (i.e. equivalent to sequences) when working with their parents, they can produce a hierarchical model of nested variable sites that can accommodate things like SNPs within deletions.

Note also that not all graph structures are decomposable into ultrabubbles. Some graph-based descriptions of variable genomic regions (for example, descriptions of

inversions or duplications that use cycles or reversing edges) do not form ultrabubbles. Some of these more complex regions can be divided into **snarls**, which are like ultrabubbles but are allowed to contain cycles or unconnected sides. However, since they are more general than ultrabubbles, snarls are more challenging to work with [85].

1.2.16 Reliable, Portable Cloud Computing with Toil

The `vg` tool is architected as small components performing relatively simple tasks. In order to use it to produce a graph reference on the scale of the proposed Human Genome Variation Map, some sort of orchestration or workflow system is necessary. This is especially true if one desires to use more than one computer in the build process; tasks need to be scheduled and code and data moved around the cluster of systems in order to perform the build. Moreover, because a build process like this might require more computing resources than are required to work with the final product, it is desirable to be able to rent those resources from on-demand cloud providers, rather than being forced to purchase them outright. Unfortunately, cloud computing offerings lack standardization [80]; integrating cloud computing directly into a workflow can produce provider lock-in.

One potential solution to this problem is Toil, a Python-based workflow development library and execution engine which is capable of composing smaller tasks into larger workflows, and of executing those workflows either on a single computer or on a cloud-based virtual cluster from any supported cloud provider [112]. Toil workflows can be written as Python scripts, which, together with Python virtual environments housing their dependencies, can be dynamically distributed to worker machines in a cloud environment. Toil nodes communicate amongst themselves using a **job store**, which can be located on a shared filesystem or within a cloud-based distributed storage system such as Amazon's S3 or Microsoft's Azure Storage. The job store is used to keep track of which parts of the workflow have successfully completed, and which parts have

not yet executed or have failed, as well as to store files, arguments, and results that are communicated from job to job.

Toil jobs can dynamically create and string together additional jobs in a directed acyclic dependency graph, meaning that workflows can dynamically adapt their shape to the shape of the data they are working with. Moreover, because the information required to execute each job is stored in the job store, failed jobs can be retried, and jobs suffering from bugs can be restarted with a corrected version of the workflow code, allowing problems with a large workflow to be corrected without losing all of the work done so far.

To make running on cloud providers easier, Toil provides a system to control workflow input and output, by importing data from URLs at the beginning of the workflow, and exporting data to URLs at the end, to eliminate the need to manually copy data to and from ephemeral cloud instances. Additionally, Toil provides a Python API for running commands through the Docker container system, allowing workflows to call command-line tools without the user having to figure out a way to get them installed on large numbers of ephemeral cloud instances. Finally, Toil integrates with Amazon Web Services to allow clusters to be automatically scaled up and down as the resource requirements of a workflow change, or as the spot market price of computing-hours rises and falls, while for Microsoft Azure Toil provides a cluster template for easy deployment in a few clicks. Overall, Toil provides a much-needed cloud abstraction layer and puts power in the hands of the researcher by commoditizing cloud services.

1.3 Research Program Overview

In the pursuit of the Human Genome Variation Map goal, I have spearheaded a number of research projects, which I have brought together here as a relatively comprehensive sampling of the overall research program. The story begins with Chapter 2, adapted

from Novak et al. [77], wherein I describe a formalized alternative to traditional seed-and-extend mapping approaches, which suffer from a dependence on complex heuristics that are difficult to describe and reason about mathematically. This alternative system of **context schemes**, described in that chapter, has a natural extension to graph-based references, and indeed I created an extensive `sequence-graphs` software system in order to produce graphs by merging sequences according to context schemes and to characterize the performance of such reference structures, implementing some of the ideas described in the ultimately unpublished Paten et al. [83]. However, I was never satisfied with the software design I used in that implementation, where a graph was built by progressive alignment and merging of sequences and stored as a large in-memory FMD index with extra associated bit vectors. I was also not satisfied with the empirical mapping performance that I was able to achieve using context schemes; I never found a scheme that produced graphs I was really happy with. Although I did not continue with the context-scheme approach, that project taught me valuable lessons about the BWT, the power of succinct data structures, and how a graph-based genomics system ought to be architected.

The second project presented here, the graph Positional Burrows-Wheeler Transform, or gPBWT, was an inversion of the previous design. As described in Chapter 3, adapted from Novak et al. [78], rather than representing a graph as a merged collection of sequences, the gPBWT represents a collection of sequences embedded in a graph. This is an inversion of my previous design. The idea of modeling the graph as a first-class object was taken from the `vg` toolkit [24], within which the implementation of the gPBWT was constructed. I believe that the overall design of the `vg` toolkit is far superior to the design of my original `sequence-graph` codebase, specifically with regard to `vg`'s factoring into relatively small tools that interact, and its focus on serializability and a coherent data model. While some aspects of `vg`, such as its lack of full support for bidirected graphs and some issues with the correctness of some of its library functions,

had to be rectified in order to make it a sufficiently stable base for further development, effort that I put into solving those problems seems to have paid off. The use of the `vg` toolkit allowed me to implement the gPBWT much more quickly than would have otherwise been possible: the initial implementation was completed over the course of a week-long "BioHackathon" in Sendai, Japan. In addition to providing mechanisms to construct, manipulate, and store graphs, `vg` had already integrated a succinct data structure library, as part of its `xg` succinct graph storage system. Furthermore, implementation in the mainline `vg` project, rather than as a separate one-off piece of software, allowed the method to reach a broader audience.

The next phase of the research program was the Global Alliance for Genomics and Health (GA4GH) Graph Bake-off, a cross-institutional collaboration to compare different methods for constructing graph-based genomic references. As detailed in Chapter 4, adapted from Novak et al. [79], the UCSC team and I collected graph submissions from across five institutions (including ourselves), and compared them on a few graph-level statistics as well as on their performance as read mapping targets and variant calling references. Based in part on experience with `vg` obtained while creating the `vg`-based gPBWT implementation, we used `vg` as the read mapper for the bake-off, and developed a new variant calling component for `vg` to facilitate a downstream variant calling analysis. In comparison to the other bake-off analyses, which we conducted using a custom codebase, the `vg`-based analyses proved dramatically simpler to rerun over the course of the project.

While the project, being under the auspices of the GA4GH, originally was intended to showcase the GA4GH's data interchange API, this aspect did not go as planned. The graph-based aspects of the GA4GH API, originally developed as a separate prototype branch of the main system, did get incorporated into the main development line, but they were later removed again by the GA4GH, to avoid trying to prematurely standardize on an API for what was still a highly experimental kind of ge-

nomics. The lack of a clear definition of a variable site in a graph context, in particular, caused a lot of confusion within the GA4GH reference variation working group. Moreover, while the intention had been to have multiple institutions producing evaluations and creating API endpoints to expose their graphs, with the API being the common protocol to enable the graphs to be communicated between parts of the system, in actuality all of the evaluations were produced by UCSC, and only the one reference server API implementation was used. With our preferred practice being to import submitted graphs from API endpoints into `vg` format for further analysis, and with the GA4GH's decision to step back from incorporating graph-awareness into its APIs, the `vg` format played a significant role in the analysis as a de facto standard for graph interchange.

The results of the bake-off project were encouraging, and supported the idea that graph-based references can yield better genomics results. The best-performing graph submissions were the 1000 Genomes graph set and the Cactus graph set. Originally intended as a simple control, the 1000 Genomes graphs ignored the variation data supplied to graph submitters, and simply provided a graph representation of the variants present in the 1000 Genomes point variant VCFs. The surprisingly high performance of these graphs illustrates the principle that working from larger datasets can often be superior to working with cleverer algorithms [30]. The Cactus graphs, on the other hand, were produced by aligning the supplied variation data, which came in the form of long example sequences, using the Cactus aligner, and converting the result to a graph.

The discovery of high-performance graph reference construction techniques in the bake-off, and particularly the discovery of two high-performance techniques that were so different from each other, prompted the final project described in this document (Ch. 5). Building on the results of the bake-off, I attempted to combine the Cactus-based and 1000-Genomes-based graph construction methods into a single technique, and to scale up from the few-megabase-sized regions used in the bake-off to

chromosomal scale. Additionally, for this project, I decided to focus specifically on large-scale structural variation, which is simple to represent in a graph but disruptive to the abstractions used by traditional linear genomics. In response to the confusion among GA4GH members during the bake-off project, our lab developed a clean mathematical description of variable sites in a graph, allowing for nesting of sites within one another [85]. As part of this final project, I substantially revised the `vg` variant caller, allowing it to call complex and nested variants in accordance with this theory. The results achieved so far at chromosome scale with the improved `vg` implementation confirm and extend the results of the bake-off, showing superior variant-calling performance at chromosome scale when using a graph-based reference incorporating known variation than when using a linear reference (Subsec. 5.3.2-5.3.3).

However, although my results so far suggest that it should be possible to build and get improved variant calling results by using a whole-genome-scale HGVM, this research program is not yet complete. There remains, of course, the task of actually building and evaluating such a graph. Moreover, the results which I have obtained at chromosome scale, particularly with respect to structural variants, indicate that some of the design compromises that the team and I made during the bake-off, in order to get a working `vg` variant caller quickly, are now restricting our variant calling performance. One of the next steps in the research program will be a redesign of the variant caller to work more directly with individual reads, instead of operating on a pileup, which I anticipate will reduce the occurrence of some of the more obvious classes of mistakes that the caller is making. There is also work to be done on the workflow that I am using to build and evaluate the graph, in order to ensure that it can scale to whole genomes while making good use of the computing resources available to it. To facilitate this, it may be necessary or desirable to extend the `vg` data model to better allow graphs to be divided into chunks and later reintegrated.

Finally, since looking at some of the preliminary bake-off read mapping results

stratified by 1000 Genomes superpopulation, I have wanted to do a paper evaluating the impact of graph references on reference bias; in theory, including more material from more populations in the reference should reduce or eliminate any effect of sample source population on the performance of genomic analyses. In practice, this theory has proven difficult to test, in part because of understandable confounding between a sample's population and where and when it was sequenced [79]. A final release of an official Human Genome Variation Map intended for serious use as a replacement for the linear reference genome should be accompanied by a detailed investigation of its population biases.

Chapter 2

Canonical, Stable, General Mapping Using Context Schemes

This chapter has been adapted from the article Novak et al. [77][1], and contains material attributable to all authors of that work. Supplementary materials referenced here are available in the online version of that article.

2.1 Abstract

2.1.1 Motivation:

Sequence mapping is the cornerstone of modern genomics. However, most existing sequence mapping algorithms are insufficiently general.

2.1.2 Results:

We introduce context schemes: a method that allows the unambiguous recognition of a reference base in a query sequence by testing the query for substrings from an

[1] Adam M. Novak, Yohei Rosen, David Haussler, and Benedict Paten, Canonical, stable, general mapping using context schemes, Bioinformatics, 2015, 31, 22, 3569-3576, by permission of Oxford University Press.

algorithmically defined set. Context schemes only map when there is a unique best mapping, and define this criterion uniformly for all reference bases. Mappings under context schemes can also be made stable, so that extension of the query string (e.g. by increasing read length) will not alter the mapping of previously mapped positions. Context schemes are general in several senses. They natively support the detection of arbitrary complex, novel rearrangements relative to the reference. They can scale over orders of magnitude in query sequence length. Finally, they are trivially extensible to more complex reference structures, such as graphs, that incorporate additional variation. We demonstrate empirically the existence of high performance context schemes, and present efficient context scheme mapping algorithms.

2.1.3 Availability and Implementation:

The software test framework created for this work is available from `https://registry.hub.docker.com/u/adamnovak/sequence-graphs/`.

2.1.4 Supplementary Information:

Six supplementary figures and one supplementary section are available with the online version of the article Novak et al. [77].

2.2 Introduction

Many tools and algorithms exist for mapping reads to a reference genome [31, 47, 54]. These tools are based on the idea of scoring local alignments between a query string and a reference according to some set of match, mismatch, and gap scoring parameters, and then finding local alignments with maximal or near-maximal scores. Seed-and-extend approaches coupled with memory efficient substring indexes or hashing schemes have been highly successful in heuristically accelerating this search process [17, 47, 54].

The core problem with read mapping is ambiguity. There is often no single best place that a read maps, especially in the case of recent duplication within the reference genome. The precise base-level alignment of the read to a given location in the reference is also often ambiguous. To mitigate this, each mapped read is given a mapping quality, a per read score that indicates how likely the mapping was generated erroneously [56]. Quantifying this uncertainty is a reasonable approach for many applications, but even then the uncertainty can be difficult to accommodate downstream.

The difficulty of mapping a read to a reference motivates a consideration of its necessity. Recently, alignment-free methods of variant calling through substring detection have garnered significant interest [16]. The basic idea is not new; the dbSNP database has long provided, for each point variant in the database, a flanking nucleotide string that indicates the DNA context in which the variation was isolated [95]. In principle such a system of variant identification sidesteps the limitations of score based alignment, and can be used to canonically detect variations. However, in practice, insufficient rigor in defining the substrings to detect, and a failure to account for other variation near identified point mutations have limited the approach's usefulness. Here we formalize and extend this core idea; we propose using multiple, algorithmically defined context strings to canonically identify the presence of each base within a reference genome (potentially paving the way for high-specificity, alignment-free variant calling) and evaluate the performance of such a method in practice.

2.3 Methods

Throughout we make use of **DNA strings**, which are finite strings over the alphabet of **DNA bases** $\{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$. A DNA string x has a **reverse complement** x^*, which is the reverse of x with each DNA base replaced with its complement; A and T are complements of each other, as are G and C.

2.3.1 Mapping

A **reference (genome)** G is a set of DNA strings and an index set of the elements of these strings, each member of which is called a **position**. Each position p uniquely identifies an element $b(p)$ of a string in G. This allows us to unambiguously discuss the "positions" in that set of DNA strings, rather than "bases" or "characters", which could be interpreted as the four DNA bases themselves.

We define the problem of mapping a **query** DNA string $x = (x_i)_{i=1}^{n}$ to a reference G. A **mapping scheme** is a function that takes x and G and, for each query element i of x, either returns a position in G, declaring the query element i to be **mapped** to that position in G, or declares the query element to be **unmapped** in G. For the scheme to map a query element to a position p in G, $b(p)$ must either be x_i (in which case that query element is **forward mapped**), or ${x_i}^*$ (in which case that query element is **reverse mapped**).

2.3.2 Contexts

A **context** is a tuple (L, B, R), where L is a DNA string called the **left part**, B the base, and R is a DNA string called the **right part**. The string LBR is the **context string** of the context (L, B, R). The context distinguishes B from the rest of the context string, so that when the context is found to occur in a query string, it is clear which character in the query string (i.e. the one corresponding to B) has been recognized. For an element i in a DNA string x a context (L, B, R) is called a **natural context** if $B = x_i$, L is a (possibly empty) suffix of $(x_j)_{j=1}^{i-1}$ and R is a (possibly empty) prefix of $(x_j)_{j=i+1}^{n}$. Some example natural contexts are visible in Supplementary Figure S1.

2.3.2.1 Context Generality

A context $c_1 = (L_1, B_1, R_1)$ is **forward more general** than a context $c_2 = (L_2, B_2, R_2)$ if L_1 is a suffix of L_2, $B_1 = B_2$, and R_1 is a prefix of R_2. That is, if you turned the two contexts into strings with their bases marked as special characters, the more general context would be a substring of the **less general** context. Note that a context is forward more general than itself. A context c_1 is **reverse more general** than a context c_2 if c_1 is forward more general than the reverse complement of c_2, which is $c_2^* = (R_2^*, B_2^*, L_2^*)$. We define a context c_1 to be generically **more general** than context c_2 if it is either forward more general or reverse more general than c_2.

2.3.3 Context-Driven Mapping

It is possible to define a mapping scheme for a query string x to a reference G in terms of contexts for positions in the reference. Such a mapping scheme makes use of a context assignment.

2.3.3.1 Context Assignment

A **context assignment** assigns each position in a reference a nonempty **context set**, such that all contexts in the set have the same base as the position, and no context in one position's set is more general than any context in any other position's set (Figure 2.1). This second property of context assignments is called **nonredundancy**.

2.3.3.2 Matching

An element i in a query string x is said to **match** a context $c = (L, B, R)$ if the query, when partitioned into the context $((x_j)_{j=1}^{i-1}, x_i, (x_j)_{j=i+1}^{n})$, is less general than c. Note that this encompasses both forward less general (in which case element i **forward matches** the context) and reverse less general (in which case element i **reverse**

matches the context). When the context is in the context set of a reference position, the element **matches** the position **on** the context.

2.3.3.3 Context-Driven Mapping Schemes

A **context-driven mapping scheme** is a mapping scheme which, for query x and reference G with context assignment C, maps each element i in x to the unique position in G which it matches under C, or leaves i unmapped when no such position exists. An element remains unmapped when it does not match any context of a reference position, or when it matches contexts of two or more positions; in the latter case we say it **discordantly matches**, an example of which is visible in Supplementary Figure S2.

Under a (nonredundant) context assignment, each position p in the reference can be mapped to, because for each context (L, B, R) of p the context string LBR matches p on that context. The nonredundancy requirement ensures this matching is not discordant: no context more general than (L, B, R) can be in the context set of any other position in the reference.

2.3.3.4 Stability

An **extension** of a DNA string x is a DNA string that contains x as a substring. An element k in an extension x' of x is a **partner** of an element i in x if the context $((x_j)_{j=1}^{i-1}, x_i, (x_j)_{j=i+1}^{n})$ is more general than $((x'_j)_{j=1}^{k-1}, x'_k, (x'_j)_{j=k+1})$.

A mapping scheme is **weakly stable** if for each element i in each possible query string x, if i is mapped to a position p in the reference, its partners in all extensions of x will map to p or be unmapped. Weak stability is desirable because it guarantees that an element in a query cannot change its mapping to a different position under extension—the mapping scheme never has to admit that it mistook one reference position for another when presented with more information. Unlike score based mapping procedures, which are generally not weakly stable, all context-driven mapping schemes are weakly

Contexts for position P_1			Contexts for position P_2		
$(L,$	$b,$	$R)$	$(L,$	$b,$	$R)$
TGTCGC	C	CAAGCA	ACGAC	C	CCAG
TGGCGC	C	CAAGCA	CGAC	C	CT
TGTCGC	C	CACA	ACGAC	C	CCATG

Figure 2.1: Example of two nonredundant context sets. Substitutions relative to the first context in each set are in bold. If the context $(L, B, R) =$ C, C, C were added to either set, it would make the context assignment redundant, as it is more general than contexts that already occur in both sets.

stable, because for any mapped element i, the partners of i in an extension of the query string can only either map to the same position p, or be discordantly matched and therefore unmapped. This is because these partners have all the natural contexts of i, and therefore must match on a context in the context set of p, but may additionally match on the context of a different position in the reference and therefore discordantly match.

A mapping scheme is **stable** if for each element i in each possible query string x, if i is mapped to a position p in the reference, its partners in all extensions of x will map to p. Stability is naturally a more desirable property than weak stability, as it restricts mapping to individual positions aligned with high certainty. By the argument above, some context-driven mapping schemes are only weakly stable. A **stable context-driven mapping scheme** is equivalent to a context-driven mapping scheme that additionally makes an element of a query string unmapped if a partner element in any extension of the query would discordantly match.

2.3.4 The Natural Context-Driven Mapping Scheme

In our earlier paper [83] we discussed a number of different context assignments, including fixed k-mer approaches. Here we focus on a new scheme that is easy to reason about and which performed the best in our preliminary empirical tests (Supplementary Figure S3).

The **natural context assignment** assigns to each position in the reference

the subset of its natural contexts that are not natural contexts of any other position in the reference. It is trivially nonredundant. The **natural (context-driven mapping) scheme**, which uses the natural context assignment, has an intuitive interpretation: an element i of a query string is mapped to a position p of the reference when all natural contexts of i with context strings unique in the reference are are assigned to p.

2.3.4.1 Overview of Algorithms

The natural context scheme is also simple to implement. For a reference and query, a **maximum unique match (MUM)** is a maximum length substring of the query present once in the reference. Our definition of a MUM differs from that used by tools like MUMmer [14] in that it is nonsymmetric; we allow a MUM to have multiple **MUM instances** in the query, each of which is a MUM and an interval of the query corresponding to a location of the substring. For a query x of length n there are at most n MUM instances, since two cannot start at the same place. Each MUM instance that contains a given element i can be described as a natural context string of i: $(x_j)^{i-1}x_i(x_j)_{i+1}$. Under the natural context assignment, the context of each such MUM-derived context string matches exactly one reference position.

Using a suffix tree with suffix links of the strings in a reference (which can be constructed in time linear in the sum of the length of the reference strings), or a related substring index data structure, it is possible to find the set of MUM instances for a query string ordered by ascending start element in $O(n)$ time. These data structures all provide two $O(1)$ operations, **extend** and **retract**, which, given a search result set for some search string, can produce the result set for a search string one character longer or shorter, respectively. Employing these operations to find all MUMs in order by ascending query start position is straightforward. Starting with the empty string, extend the search string on its right end with successive characters from the query string until such an extension would produce a search result set with no results (or until the

query string is depleted). If at that point there is one result in the result set, and at least one extension has happened since the last retraction, then a MUM has been found. Next, retract a single character from the left end of the search string, and go back to extending with the remaining unused query string characters. Repeat the whole process until the query string is depleted.

Since each successful extend operation consumes a character from the query string, no more than $O(n)$ extend operations can ever be performed. Since each retract operation moves the left end of the search string to the right in the query, no more than $O(n)$ retract operations can be performed. And since each unsuccessful extend operation (which would produce an empty result set) is followed by a retract operation, no more than $O(n)$ of those can happen either. Thus the entire algorithm is $O(n)$.

Once the MUM instances have been found, it is necessary to identify the query elements that occur in exactly one MUM and therefore can be mapped under the natural scheme. (If an element is contained in two or more MUM instances then it must be discordantly mapped, because each must define a context that matches the element to a distinct position.) Given the MUM instances ordered by ascending query start element, it can be determined for all elements if each is in one, zero or multiple MUM instances, by a single traversal of the ordered MUM instances taking $O(n)$ time. We can therefore determine in $O(n)$ which elements in a query string are mapped. The combined time to map all the elements in a new query string given an existing reference substring index data structure of the type discussed is therefore $O(n)$.

2.3.5 The α-β-Natural Context-Driven Mapping Scheme

Under the natural context assignment, for each (by definition minimally unique) reference context string, there must exist another reference substring that is an edit distance of one from it. Therefore, while the natural context assignment ensures each context identifies a single position in the reference, a single substitution, insertion or deletion in

a query substring could result in a change in mapping. To avoid this, we now define a more robust scheme.

Throughout, we use the Levenshtein edit distance, in which a single character replacement, insertion, or deletion is worth one. This choice of edit distance metric makes reasoning about the behavior of our algorithms simpler, but they could potentially be extended to other metrics tailored to different sequence data sources.

For a pair of overlapping substrings (x_j), (x_k) of a string x, we call elements in either substring not contained within the intersection of their intervals on x **separate**. For two substrings within the reference (not necessarily overlapping or even in the same reference string) we can similarly talk about their number of separate elements. For a given reference substring, the **α-separation** is the minimum edit distance α between it and any other substring in the reference with a number of separate elements greater than its edit distance from the original substring. For a given natural context of a reference position, its α-separation is the α-separation of its context string.

Having a minimum α-separation for contexts in a natural context scheme makes mappings more stable in the face of edits to the query. Specifically, it ensures that the number of edits required to transform the context of one position into the context of another is at least α, for positions whose context strings have more than α separate elements. When two reference substrings with α edit distance have exactly α separate elements (it is easy to verify they can not have fewer than α) then there exists a minimum edit-distance alignment of the two substrings that only aligns together bases from each substring with the same reference positions, and the α edit distance is therefore trivially realizable as the removal of a prefix from one substring and a suffix from the other. However, it is also possible that two substrings with α edit distance and α separate elements could have other minimum edit distance alignments that would result in different mappings. Therefore, enforcing α-separation on a context assignment does not absolutely prevent mismappings produced by fewer than α edits—however,

such mismappings would have to be relatively localized on the reference.

Similarly, the natural context assignment is intolerant to edits between the query string and context strings of positions in the reference to which we might like query elements to map. To mitigate this issue, for a given context (L, B, R) and element i in DNA sequence x we define β**-tolerance**: if $x_i = B$, β is the minimum edit distance between the context string LBR and a natural context string of i. If $x_i \neq B$ then $\beta = \infty$. Hence for a position in the reference p, a β-tolerant context (L, B, R) is a context such that $b(p) = B$ and LBR is within β edits from a natural context string of p. The α-β**-natural context assignment** assigns each position in the reference a context set containing the minimal length contexts that are at least α-separated, and at most β-tolerant from it. It can be verified that as long as α is greater than or equal to one and β is less than $\alpha/2$ then the context assignment is nonredundant and therefore valid. The α-β-natural context assignment ensures all admitted contexts are both α-separated (and therefore unlikely to be matched by chance, requiring α misleading edits to coincide), and at most β-tolerant (and therefore tolerant of up to β edits obscuring the true reference position). The natural context-driven mapping scheme is a special case of the α-β**-natural (context-driven mapping) scheme** when both α and β equal 0. (A possible extension would be a context scheme in which the α-separation and β-tolerance required to admit contexts depended on the context length, but this would make the parametrization of the context scheme quite complex, and so is not explored here.)

2.3.5.1 Overview of Heuristic Algorithms for the α-β-Natural Context-Driven Mapping Scheme

Unfortunately, algorithms built on efficient substring indexes to implement the α-β-natural scheme require tracking a number of potential matches that is exponential in both α and β parameters. Instead we pursue an algorithm that heuristically approx-

imates this scheme. A full description of this algorithm is available in Supplementary Section S1; the basic idea, inspired by existing seed-and-extend hashtable methods and chaining methods like BWA-MEM, is to chain exact matches separated by mismatching gaps, until a sufficient α-separation is obtained [49, 55].

For a reference, a **minimal unique substring (MUS)** is a shortest length substring that appears once in that reference. Two MUSes are disjoint if they do not overlap. We define α' as the maximum number of disjoint MUSes within a context string. It is easy to verify that α' is a lower bound on α. Intuitively, each disjoint MUS would need to be disrupted by a distinct edit.

The heuristic algorithm attempts to chain together MUMs to accumulate at least α' disjoint MUSes, without requiring more than β' edits in the **interstitial substrings** between the MUMs. This creates **β'-synteny blocks**, as depicted in Figure 2.2, which are maximal sequences of MUMs that agree in order and orientation, and which have β' or fewer edits between the strings they mark out in the reference and the query. If a β'-synteny block can be created that has at least α' disjoint MUMs (and is thus α'-separated), the MUM instances it contains are used as in the natural mapping scheme above, to define contexts for the involved reference positions.

This heuristic algorithm, as demonstrated in Supplementary Section S1, finds contexts of reference positions in the query string that are at least α'-separated, and at most β'-tolerant, and takes $O(\beta'^2 n)$ time to map the query string, given the previously described substring index structure for the reference. Provided $\alpha' < \beta'$, this context scheme is nonredundant. The contexts found (and thus the matchings made) by this heuristic scheme are a subset of those that would be produced by the exact algorithm, although the same is not always true of the resulting mappings. A more thorough, empirical comparison of this heuristic scheme to an implementation of the exact scheme is left as future work, primarily due to the above-mentioned computational difficulty inherent in nontrivial exact β values.

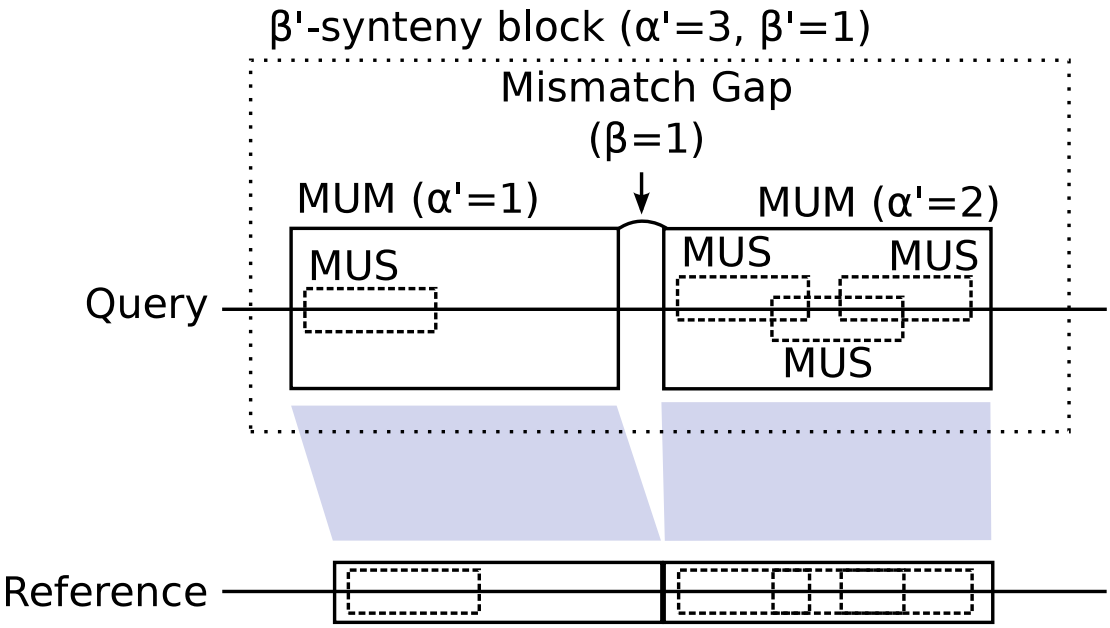


Figure 2.2: Diagram of a β'-synteny block for the α'-β'-natural context scheme, composed of two MUMs.

2.3.6 Credit

It is common to find some elements in a query string x which are unmapped, and cannot be mapped on any extension of x, yet are intuitively recognizable as corresponding to reference positions. This often happens if bases are between two MUMs but are not part of any MUM themselves, or if they were part of a MUM between two other MUMs that cannot join a sufficiently α'-separated β'-synteny block. In these cases, to create a scheme that maps more elements of the query, we can augment our context assignments with additional contexts that allow such bases to map **on credit**, based on the mappings of the bases on either side. The particular credit method used here looks at the nearest mapped base on either side of a gap in mapping, and matches up elements with the correct bases with respect to their implied positions in the reference, allowing at most one mismatch. Previously unmapped elements that are matched to exactly one reference position will be mapped on credit, while elements that are matched to zero or two positions will not map.

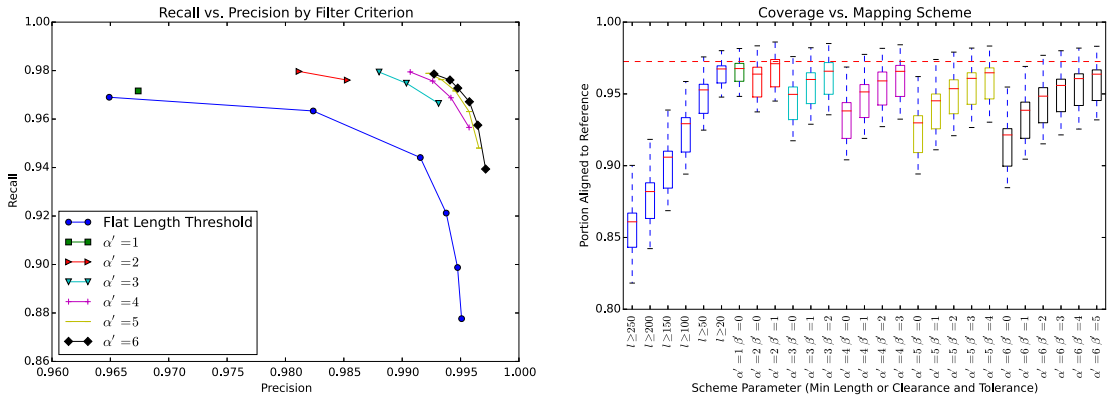
Since only elements that did not already match, and which could not possibly match on any extension of the query, are mapped in this way, the addition of credit does not interfere with the nonredundancy of a context assignment or the stability of a context-driven mapping scheme.

2.4 Results

In order to test the utility of the theoretical constructs described here, a series of software tests were created in order to evaluate the mappings produced by the α-β-natural scheme described above. Mapping accuracy was evaluated for both error-corrected long sequences and error-prone short sequences.

2.4.1 Mapping MHC Alt Loci

To evaluate the performance of the new mapping algorithms proposed here a long-sequence mapping task was defined. The human genome has, on chromosome 6, a region of approximately 5 megabases known as the **major histocompatibility complex (MHC)**, rich in antigen presentation genes vital to the function of the immune system [110]. The region is prone to complex rearrangement, is well-supplied with both coding and noncoding sequence, and exhibits extreme regional variation in the polymorphism rate along its span [110]. As one of the most complex and difficult regions of the genome, it provides a good testbed for methods designed to deal with difficult structural variation. To better represent the complexity of this region, the Genome Reference Consortium (GRC)'s current human reference assembly (GRCh38) contains seven full-length MHC **alt loci**, each of which serves as a different alternate reference for the region [8]. These alt loci come with official alignments to the chromosome 6 primary sequence, which are part of GRCh38 and were generated using the NGAligner tool and limited manual curation [93, 94].



(a) (b)

Figure 2.3: Results of MHC alignment. Points shown in 2.3a are averages of alt-loci alignments. Lines connect different β' levels for a given α'. The red dashed line in 2.3b is the average coverage of the GRC reference alignments.

Each mapping scheme under test took the MHC alt loci (GI568335879, GI568335954, GI568335976, GI568335986, GI568335989, GI568335992, and GI568335994), and mapped each to the GRCh38 **primary path** region, which actually appears in the "chr6" FASTA record. The resulting alignments were compared against the official GRC alignments distributed with GRCh38, with the caveat that aligned, mismatched bases in the GRC alignments were de-aligned to ensure a fair comparison, as the mapping schemes being evaluated were not permitted to align mismatching bases together. (Allowing mismatching bases in the GRC alignments to align made no perceptible difference in any figure, and was not pursued further.) The standard information retrieval metrics of precision and recall against the GRC alignments were calculated using `mafComparator`, and can be seen in Figure 2.3a [19]. Overall coverage (the fraction of bases of an alt locus aligned to the reference), and the frequency and size of rearrangement events implied by the alignments, were also calculated, and are visible in Figure 2.3b and Figure 2.4, respectively.

Mapping schemes using a wide range of α' and β' parameters were tried, with β' being restricted to values less than α'. Additionally, the natural mapping scheme

($\alpha' = 0, \beta' = 0$) was tested, with a parameter used to vary the minimum length of admissible unique substrings (i.e. defining a series of natural context scheme, each only considering unique reference substrings longer than a minimum threshold).

The strongest performing schemes, in terms of the harmonic mean of precision and recall (F-score), had a precision greater than 0.99 and a recall of around 0.98. Coverage was also remarkably close to that of the GRC reference alignments, suggesting that the conservative nature of the schemes did not result in undue underalignment (Figure 2.3b).

In all cases the natural length-thresholded context schemes performed substantially worse than the various α'/β' combinations in terms of recall of the GRC alignments at a given level of precision (Figure 2.3a), and in terms of coverage (Figure 2.3b). This suggests that α' and β' as defined here are effective heuristics.

Increasing α' for a given β' was found to increase precision and decrease recall, but increasing β' at a given α' could restore the lost recall (at the cost of precision). The $\alpha' = 5, \beta' = 4$ natural scheme was determined to strike the best balance between precision and recall, as there was a negligible increase in precision between it and the $\alpha' = 6, \beta' = 5$ scheme (Figure 2.3a). Both it and the $\alpha' = 3, \beta' = 2$ scheme—selected to provide a good balance between precision and recall while also optimizing for mapping shorter sequences—were chosen for the short sequence mapping tests in 2.4.2 below.

Two additional configuration options were available for the schemes under test: whether to map unmapped internal bases on credit, and whether to enforce stability over weak stability. Our tests, the results of which are visible in Supplementary Figure S1a and Supplementary Figure S4b, demonstrate that requiring stability had a negligible impact on recall for long sequences, while the use of credit produced a sizable gain in recall at a manageable cost in precision (note the scales of the axes in Supplementary Figure S4b). Consequently, credit was used for all analyses, and the stability requirement was used for the MHC mapping analysis.

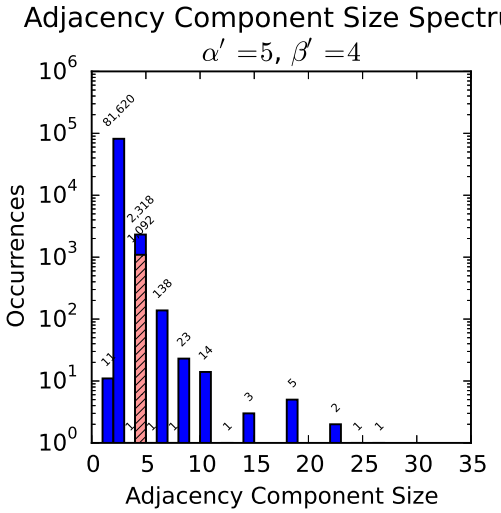


Figure 2.4: Frequency of rearrangements of different levels of complexity implied by the alignment of MHC alt loci to the primary path, under the $\alpha' = 5$, $\beta' = 4$ natural scheme, which was selected to give the best balance between precision and recall. The X axis shows the number of nodes involved in the rearrangement, while the Y axis shows the number of rearrangements of that size. The red bar shows the number of 4-node rearrangements that are automatically identifiable as tandem duplications.

The $\alpha' = 5$, $\beta' = 4$ natural scheme, which provided the best trade-off between precision and recall, was also evaluated in terms of the number and complexity of rearrangements it invoked to relate the MHC alt loci back to the primary path. Figure 2.4 depicts a "spectrum" plot of rearrangement frequency versus size, where a rearrangement is defined as a connected component in a multi-breakpoint/adjacency graph representing the alignment between the primary reference sequence and an alt-loci sequence [70, 81]. Briefly, the nodes of the graph are the ends (sides) of aligned sets of two or more bases and the edges the adjacencies, possibly containing interstitial unaligned sequence, that connect these ends [70, 81]. The spectrum plot shows that the vast majority of rearrangements involve only two nodes (which is the structure of SNPs and simple indels), and of the rearrangements involving 4 nodes, slightly under half of them are recognizable as simple tandem duplications. The tandem duplications, which frequently involve just a handful of bases, are discoverable because of the non-linear

nature of context-driven mapping. The remaining, more complex rearrangements have not been identified or named. Supplementary Figure S5 shows UCSC Genome Browser renderings of some of the rearrangements described in the spectrum plot.

2.4.2 Mapping Simulated Short Reads

Perhaps the most important current application of traditional alignment methods is mapping reads from short read sequencing. To test this scenario a second mapping task was created. Each of the MHC alt loci sequences was broken into overlapping 200bp fragments at 100bp intervals. The read length was chosen to align with that of current or expected near future sequencing technologies, and is near the low end of what the mapping schemes presented here can accommodate [90]. Each of these fragments had substitution errors introduced with an independent probability of 1% per base (comparable to current sequencing technologies) [90]. We used this simulated scenario, rather than actual reads, because it allowed us to assess the reads' origins to easily determine if mappings were correct or aberrant.

Two variants of the α'-β'-natural scheme ($\alpha' = 3, \beta' = 2$, and $\alpha' = 5, \beta' = 4$), in both stable and weakly stable versions, were used to map each read to the primary path MHC from GRCh38. The results of the popular aligners BWA (using default parameters) and LASTZ (using an empirically-determined restrictive score cut-off), were also included [31, 54]. BWA in particular functioned as a gold standard: we did not expect to outperform BWA, but rather sought to recapitulate some of its results in a context-driven framework.

Mapping accuracy was assessed in two ways. First, the number of reads that each mapper could place anywhere in the reference, and portion of bases mapped regardless of correctness, were measured. These results are visible in Figures 2.5a and 2.5b, respectively. Second, the number of genes and portion of gene bases with incorrect mappings to other genes, as annotated by the UCSC Genome Browser's "Known Genes"

database, were also measured, and are visible in Figures 2.5c and 2.5d [71].

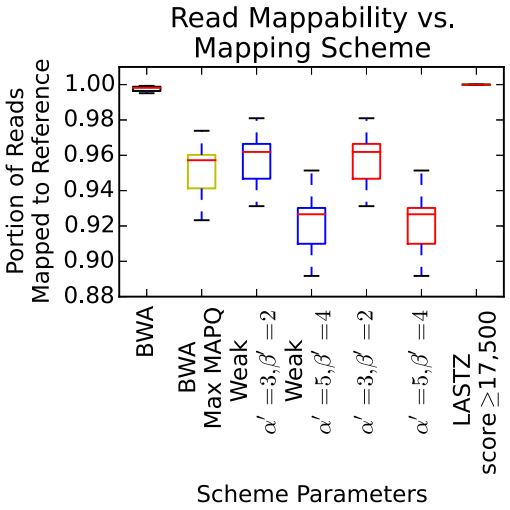
BWA and LASTZ both mapped more of the reads and covered more of the read bases than the context-driven mapping schemes, though the difference was relatively small: less than 10% in terms of mapped reads and, for the weakly stable context schemes, less than 15% in terms of coverage. These results were unsurprising, given that a context-driven mapping scheme is a function that can not multi-map any position, while the other two aligners freely produced multi-mappings.

The context-driven mapping schemes examined broadly matched BWA's performance in terms of avoiding mapping genes to their paralogs (Figures 2.5c and 2.5d). All four context-driven schemes tested outperformed BWA's raw output. However, if BWA's output was filtered to only include reads mapped with maximum mapping quality (which was observed to be 60), only the $\alpha' = 5$, $\beta' = 4$ natural schemes managed to outperform it in terms of the portion of genes with any mappings to paralogs—and that at a very substantial drop in coverage (Figure 2.5b). LASTZ, on the other hand, did not report mapping qualities; even with what was intended to be a stringent score threshold applied, it produced the most mappings to paralogs of any aligner tested (Figures 2.5c and 2.5d).

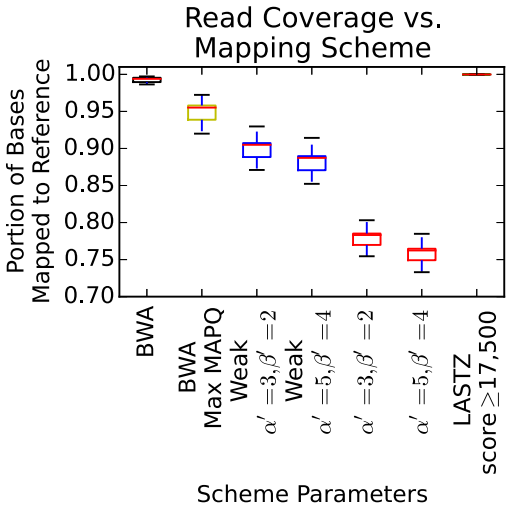
While the difference between stable and weakly stable mapping schemes was insignificant for long-read mapping, the coverage difference for these shorter reads was much greater. Thus stability, rather than weak stability, might seem an impractical restriction for short reads, albeit one that still admits the mapping of the majority of query sequence elements.

A final experiment characterized the minimum context lengths with which it was possible for a base to map in the GRCh38 primary path MHC; the results are shown in Figure 2.6. The vast majority of bases were found to be mappable with contexts of 100bp or less, and all but about 2% of bases at $\alpha' = 5$ were found to be mappable with contexts of 200bp or less.

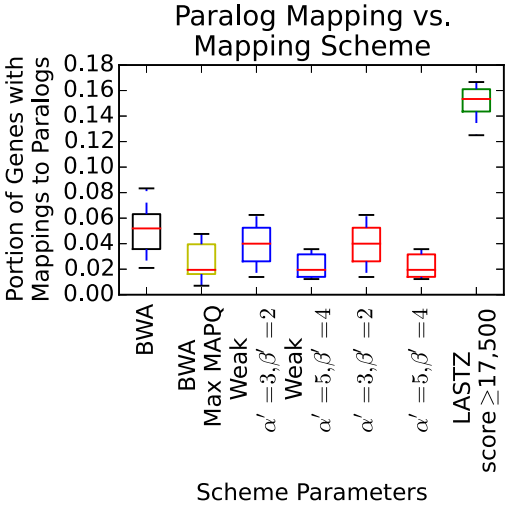
Figure 2.5: Results of read alignments. Reads were generated from MHC alt loci by taking tiling 200bp windows at 100bp intervals, and randomly introducing substitution errors at a frequency of 1%. Reads were aligned to the GRCh38 primary path MHC region.


Read Mappability vs.
Mapping Scheme
Portion of Reads
Mapped to Reference
1.00
0.98
0.96
0.94
0.92
0.90
0.88
BWA
BWA
Max MAPQ
Weak
$\alpha' = 3, \beta' = 2$
Weak
$\alpha' = 5, \beta' = 4$
$\alpha' = 3, \beta' = 2$
$\alpha' = 5, \beta' = 4$
LASTZ
score ≥ 17,500
Scheme Parameters

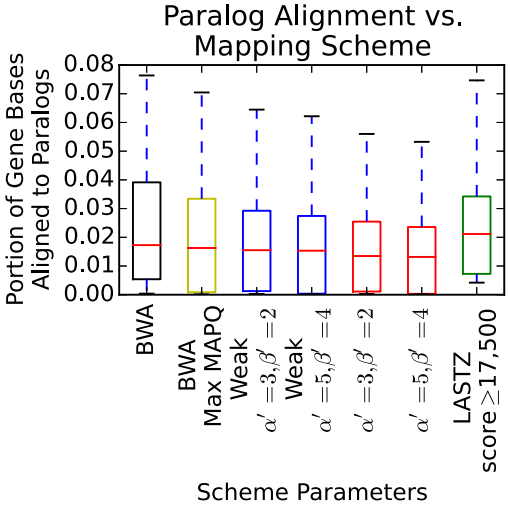
(a)


Read Coverage vs.
Mapping Scheme
Portion of Bases
Mapped to Reference
1.00
0.95
0.90
0.85
0.80
0.75
0.70
BWA
BWA
Max MAPQ
Weak
$\alpha' = 3, \beta' = 2$
Weak
$\alpha' = 5, \beta' = 4$
$\alpha' = 3, \beta' = 2$
$\alpha' = 5, \beta' = 4$
LASTZ
score ≥ 17,500
Scheme Parameters

(b)

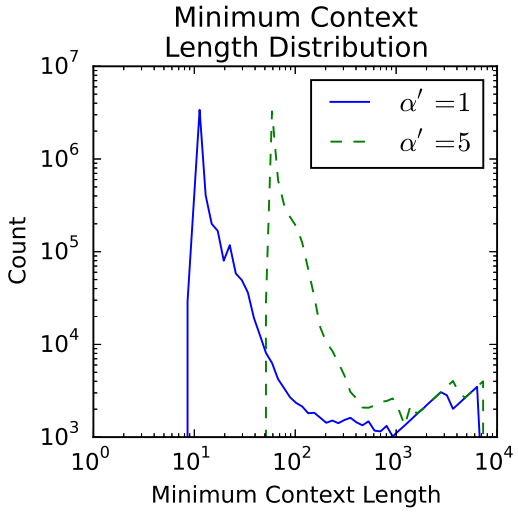

Paralog Mapping vs.
Mapping Scheme
Portion of Genes with
Mappings to Paralogs
0.18
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
BWA
BWA
Max MAPQ
Weak
$\alpha' = 3, \beta' = 2$
Weak
$\alpha' = 5, \beta' = 4$
$\alpha' = 3, \beta' = 2$
$\alpha' = 5, \beta' = 4$
LASTZ
score ≥ 17,500
Scheme Parameters

(c)

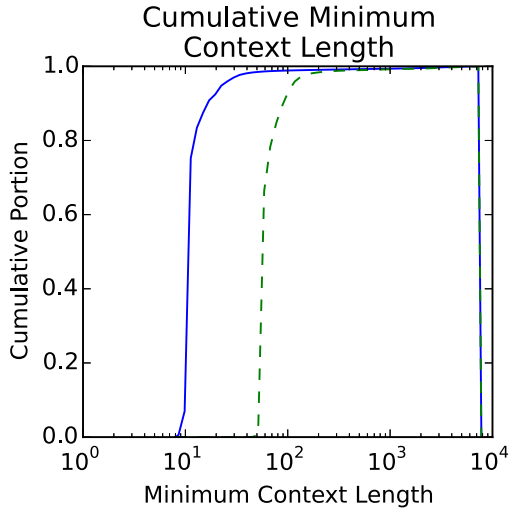

Paralog Alignment vs.
Mapping Scheme
Portion of Gene Bases
Aligned to Paralogs
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
BWA
BWA
Max MAPQ
Weak
$\alpha' = 3, \beta' = 2$
Weak
$\alpha' = 5, \beta' = 4$
$\alpha' = 3, \beta' = 2$
$\alpha' = 5, \beta' = 4$
LASTZ
score ≥ 17,500
Scheme Parameters

(d)

Figure 2.6: Minimum $\beta' = 0$ context lengths required to map uniquely in a reference derived from the GRCh38 primary path MHC, for different α' values. At an α' of 1, 1.16% of minimal contexts are longer than 100bp, and 0.97% are longer than 200bp. At an α' of 5, 8.85% of minimal contexts are longer than 100bp, and 1.74% are longer than 200bp


Minimum Context
Length Distribution
Count
Minimum Context Length
$\alpha' = 1$
$\alpha' = 5$
10^0
10^1
10^2
10^3
10^4
10^3
10^4
10^5
10^6
10^7

(a)


Cumulative Minimum
Context Length
Cumulative Portion
Minimum Context Length
0.0
0.2
0.4
0.6
0.8
1.0
10^0
10^1
10^2
10^3
10^4

(b)

2.5 Discussion

The new mapping scheme proposed here—both radically different and more conservative than existing methods—has some important benefits. The first is that it is versatile: it can be used to map multi-megabase MHC sequences while accounting for complex rearrangements, but also does reasonably well with 200bp simulated reads. The second major benefit is stability: although requiring stability reduces coverage when mapping short reads, it reveals a majority subset of mapped positions that are aligned globally with high certainty. This is a useful per-base quality assessment somewhat orthogonal and complementary to the widely used read-level mapping quality scores [56]. The third major benefit—being able to define variants in terms of canonical contexts which can diagnose their presence—is related to the second: having a stable mapping scheme enables the articulation of sequences which, when observed, always indicate the presence of a particular variant. This could ultimately pave the way for a high-specificity reference-free method of variant calling, building on the dbSNP concept of flanking strings [95].

Our results show that the context-driven, stable mapping approach can be more conservative than existing mappers like BWA and LASTZ, at the cost of coverage. If there is any possibility of later having to admit that it was wrong in mapping a base, a stable scheme will not map that base. A weakly stable scheme is only slightly more permissive, willing to map bases only if it knows they cannot possibly map elsewhere. We show that the α'-β'-natural schemes can be much more selective than LASTZ, and can in certain circumstances outperform BWA in avoiding mappings to paralogs, and in the general case are no worse. This specificity comes at the cost of a reduced ability to contend with high sequencing error rates. However, it is particularly important when analyzing regions like the MHC, where some genes present in a query may not be present themselves in the reference to which reads are being mapped, but may nonetheless have

close and misleading paralogs in the reference.

The α'-β'-natural scheme presented here is more useful for mapping longer sequences, where the costs of stability (incurred only near the sequence ends) are lower, and the chances of finding longer and more distinctive contexts are higher. Longer reads are also more likely to directly exhibit some of the linearity-breaking structural rearrangements that our scheme is designed to deal with. The scheme presented here largely recapitulates the GRC's official alignments. The $\alpha' = 5, \beta' = 4$ instantiation, for example, has approximately 99% precision and 98% recall when compared to the GRC alignments, as depicted in Figure 2.3a. Given that the GRC alignments for the MHC alt loci do not contain any duplications, translocations, or inversions, some of the missing precision is almost certainly due to the correct detection of events that the GRC alignments did not completely describe. Judging by our manual analysis (illustrated in Supplementary Figure S5), such calls are generally plausible.

Finally, the context-based mapping scheme method is abstracted from its reference and query inputs, and thus easy to generalize. In addition to being very general in the types of queries it can accept, from short reads to entire alt loci, it is also very general in the types of references it can map to. As long as context sets can be defined for each position, this method can be extended to map to nonlinear, graph-based reference structures (as in Supplementary Figure S6). Such graph structures, containing common variation in addition to the primary reference, would help to alleviate the reference allele bias inherent in current approaches to variant detection. The mapping scheme presented here provides a concrete approach to mapping to such a structure, something we explored in our earlier paper [83] and that we are actively pursuing.

Future work to be done on this project includes the creation of a full alignment tool based on the algorithms described here, and an extension of those algorithms to graph-structured references. The software test framework created for this work is available from `https://registry.hub.docker.com/u/adamnovak/sequence-graphs/`.

Acknowledgements

Funding This work was supported by a grant from the Simons Foundation (SFLIFE #351901). AN was supported by research gift from Agilent Technologies, a fellowship from Edward Schulak, and an award from the ARCS Foundation. Research reported in this publication was also supported by the National Human Genome Research Institute of the National Institutes of Health under Award Number U54HG007990. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

Chapter 3

A Graph Extension of the Positional Burrows-Wheeler Transform and its Applications

This chapter has been adapted from the article Novak et al. [78][1], and contains material attributable to all authors of that work.

3.1 Abstract

We present a generalization of the Positional Burrows-Wheeler Transform, or PBWT, to genome graphs, which we call the gPBWT. A genome graph is a collapsed representation of a set of genomes described as a graph. In a genome graph, a haplotype corresponds to a restricted form of walk. The gPBWT is a compressible representation of a set of these graph-encoded haplotypes that allows for efficient subhaplotype match queries. We give efficient algorithms for gPBWT construction and query operations. As a demonstration,

[1]WABI 2016: Algorithms in Bioinformatics, A Graph Extension of the Positional Burrows-Wheeler Transform and Its Applications, Lecture Notes in Computer Science 9838, 2016, 246-256, Adam M. Novak, Erik Garrison, Benedict Paten,

we use the gPBWT to quickly count the number of haplotypes consistent with random walks in a genome graph, and with the paths taken by mapped reads; results suggest that haplotype consistency information can be practically incorporated into graph-based read mappers. We estimate that with the gPBWT of the order of 100,000 diploid genomes, including all forms structural variation, could be stored and made searchable for haplotype queries using a single large compute node.

3.2 Introduction

The PBWT is a compressible data structure for storing haplotypes that provides an efficient search operation for subhaplotype matches [18]. The PBWT is itself an extension of the ordinary Burrows-Wheeler Transform (BWT), a method for compressing string data [6], with some concepts borrowed from the FM-index, an extension of the BWT that makes it searchable [22]. Implementations of the PBWT, such as BGT [52], can be used to compactly store and query the haplotypes of thousands of samples. The PBWT can also allow existing haplotype-based algorithms to work on much larger collections of haplotypes than would otherwise be practical [60]. The Haplotype Reference Consortium dataset, for example, contains 64,976 haplotypes [67], and PBWT-based software allows data at this scale to efficiently inform phasing calls on newly sequenced samples, with significant speedups over other methods [59].

In the PBWT each site (corresponding to a genetic variant) is a binary feature and the sites are totally ordered. The input haplotypes to the PBWT are binary strings, with each element in the string indicating the state of a site. In the generalization we present, each input haplotype is a walk in a general bidirected graph, or genome graph. Graph-based approaches to genomics problems like mapping and variant calling have been shown to produce better results than linear-reference-based methods [16, 79], so adapting the PBWT to a graph context is expected to be useful. Other generalizations

of BWT-based technologies to the graph context have been published [38, 63, 100], but they deal primarily with the substring search problem, rather than the problem of storing and querying haplotypes.

The PBWT generalization presented here allows haplotypes to be partial (they can start and end at arbitrary nodes) and to traverse arbitrary structural variation. It does not require the sites (nodes in the graph) to have a biologically relevant ordering to provide compression. However, despite these generalizations, essential features of the PBWT are preserved. The core data structures are similar, the compression still exploits genetic linkage, and the haplotype matching algorithm is essentially the same. It is expected that this generalization of the PBWT will allow large embedded haplotype panels to inform read-to-graph alignment, graph-based variant calling, and graph-based genomic data visualization, bringing the benefits of the PBWT to the world of genome graphs.

3.3 Definitions

We define $G = (V, E)$ as a **genome graph** in a bidirected formulation [69, 83]. Each node in V has a DNA-sequence label; a left, or $5'$, **side**; and a right, or $3'$, side. Each edge in E is a pairset of sides. The graph is not a multigraph: only one edge may connect a given pair of sides and thus only one **self-loop**, or edge from a side to itself, can be present on any given side.

While more powerful algorithms are generally used in practice, a simple genome graph can be constructed relatively easily from a reference sequence and a set of nonoverlapping variants (defined as replacements of a nonempty substring of the reference with a nonempty alternate string). Start with a single node containing the entire reference sequence. For each variant to be added, break the nodes in the graph so that the reference allele of the variant is represented by a single node. Then create a node to

represent the alternate allele, and attach the left and right sides of the alternate allele to everything that is attached to the left and right, respectively, of the reference allele.

We consider all the sides in the graph to be (arbitrarily) ordered relative to one another. We define the **null side**, 0, as a value which corresponds to no actual side in the graph, but which compares less than any actual side. We also define the idea of the **opposite** of a side s, with the notation $\overline{s}$, meaning the side of s's node which is not s (i.e. the left side of the node if s is the right side, and the right side of the node if s is the left side). Finally, we use the notation $n(s)$ to denote the node to which a side s belongs.

To better connect the world of bidirected graphs, in which no orientation is better than any other, and the world of algebra, in which integer subscripts are incredibly convenient, we introduce the concept of an **ambisequence**. An ambisequence is like a sequence, but the orientation in which the sequence is presented is insignificant; a sequence and its reverse are both equal and opposite **orientations** of the same underlying ambisequence. An ambisequence is isomorphic to a stick-shaped undirected graph, and the orientations can be thought of as traversals of that graph from one end to the other. For every ambisequence s, a canonical orientation is chosen arbitrarily, and the subscripted items s_i are the items in that arbitrarily selected sequence. This orientation is also used for defining concepts like "previous" and "next" in the context of an ambisequence.

Within the graph G, we define the concept of a **thread**, which can be used to represent a haplotype or haplotype fragment. A thread t on G is a nonempty ambisequence of sides, such that for $0 \leq i < N$ sides t_{2i} and t_{2i+1} are opposites of each other, and such that G contains an edge connecting every pair of sides t_{2i} and t_{2i+1}. In other words, a thread is the ambisequence version of a walk through the sides of the graph that alternates traversing nodes and traversing edges and which starts and ends with nodes. Note that, since a thread is an ambisequence, it is impossible to reverse.

Instead, the "reverse" of a thread is one of its two orientations.

We consider G to have associated with it a collection of **embedded** threads, denoted as T. We propose an efficient storage and query mechanism for T given G.

The Graph Positional Burrows-Wheeler Transform

Our high-level strategy is to store T by grouping together threads that have recently visited the same sequences of sides, and storing in one place the next sides that those threads will visit. As with the Positional Burrows-Wheeler Transform, used to store haplotypes against a linear reference, and the ordinary Burrows-Wheeler Transform, we consider the recent history of a thread to be a strong predictor of where the thread is likely to go next [18]. By grouping together the next side data such that nearby entries are likely to share values, we can use efficient encodings (such as run-length encodings) and achieve high compression.

More concretely, our approach is as follows. Within an orientation, we call an instance of side in an even-numbered position $2i$ a **visit**; a thread may visit a given side multiple times, in one or both of its orientations. (We define it this way because, while a thread contains both the left and right sides of each node it touches, we only want one visit to stand for the both of them.) Consider all visits of orientations of threads in T to a side s. For each visit, take the sequence of sides coming before this arrival at s in the thread and reverse it, and then sort the visits lexicographically by these (possibly empty) sequences of sides, breaking ties by an arbitrary global ordering of the threads. Then, for each visit, look two steps ahead in its thread (past s and $\overline{s}$) to the side representing the next visit, and append it (or the null side if there is no next visit) to a list. After repeating for all the sorted visits to s, take that list and produce the array $B_s[]$ for side s. An example $B[]$ array and its interpretation are shown in Figure 3.1. (Note that, throughout, arrays are indexed from 0 and can produce their

lengths trivially upon demand.)

Each unoriented edge $\{s, s'\}$ in E has two orientations (s, s') and (s', s). Let $c()$ be a function of these oriented edges, such that for an oriented edge (s, s'), $c(s, s')$ is the smallest index in $B_{s'}[]$ of a visit of s' that arrives at s' by traversing $\{s, s'\}$. Note that, because of the global ordering of sides and the sorting rules defined for $B_{s'}[]$ above, $c(s_0, s') \leq c(s_1, s')$ for $s_0 < s_1$ both adjacent to s'. Figure 3.2 and Table 3.1 give a worked example of a collection of $B[]$ arrays and the corresponding $c()$ function values.

For a given G and T, we call the combination of the $c()$ function and the $B[]$ arrays a **graph Positional Burrows Wheeler Transform (gPBWT)**. We submit that a gPBWT is sufficient to represent T, and, moreover, that it allows efficient counting of the number of threads in T that contain a given new thread as a subthread.

Table 3.1: $B_s[]$ and $c()$ values for the embedding of threads illustrated in Figure 3.2.

Side	$B_s[]$ Array	Edge	$c(s,t)$ count
1	[5]	$\{2, 5\}$	0
2	[0]	$\{4, 5\}$	1
3	[5]	$\{6, 7\}$	1
4	[0]	$\{6, 9\}$	0
5	[9, 7]	$\{8, 8\}$	0
6	[4, 2]	$\{10, 9\}$	1
7	[8, 8]	$\{5, 2\}$	0
8	[6, 0]	$\{5, 4\}$	0
9	[9, 0]	$\{7, 6\}$	0
10	[10, 6]	$\{9, 6\}$	1

3.4 Extracting Threads

To reproduce T from G, and the gPBWT, consider each side s in G in turn. Establish how many threads begin (or, equivalently, end) at s by taking the minimum of $c(x, s)$ for all sides x adjacent to s. If s has no incident edges, take the length of $B_s[]$ instead. Call

Figure 3.1: An illustration of the $B_1[]$ array for a single side numbered 1. (Note that a similar, reverse view could be constructed for the $B_2[]$ array and the opposite orientations of all the thread orientations shown here, but it is omitted for clarity.) The central rectangle represents a node, and the pairs of solid lines on either side delimit edges attached to either the left or right side of the node, respectively. These edges connect the node to other parts of the graph, which have been elided for clarity. The dashed lines within the edges represent thread orientations traveling along each edge in a conserved order, while the solid lines with triangles at the ends within the displayed node represent thread orientations as they cross over one another within the node. The triangles themselves represent "terminals", which connect to the corresponding dashed lines within the edges, and which are wired together within the node in a configuration determined by the $B_1[]$ array. Thread orientations entering this node by visiting side 1 may enter their next nodes on sides 3, 5, or 7, and these labels are displayed near the edges leaving the right side of the diagram. (Note that we are following a convention where nodes' left sides are assigned odd numbers, and nodes' right sides are assigned even numbers.) The $B_1[]$ array records, for each thread orientation entering through side 1, the side on which it enters its next node. This determines through which of the available edges it should leave the current node. Because threads tend to be similar to each other, their orientations are likely to run in "ribbons" of multiple thread orientations that both enter and leave together. These ribbons cause the $B_s[]$ arrays to contain runs of identical values, which may be compressed.

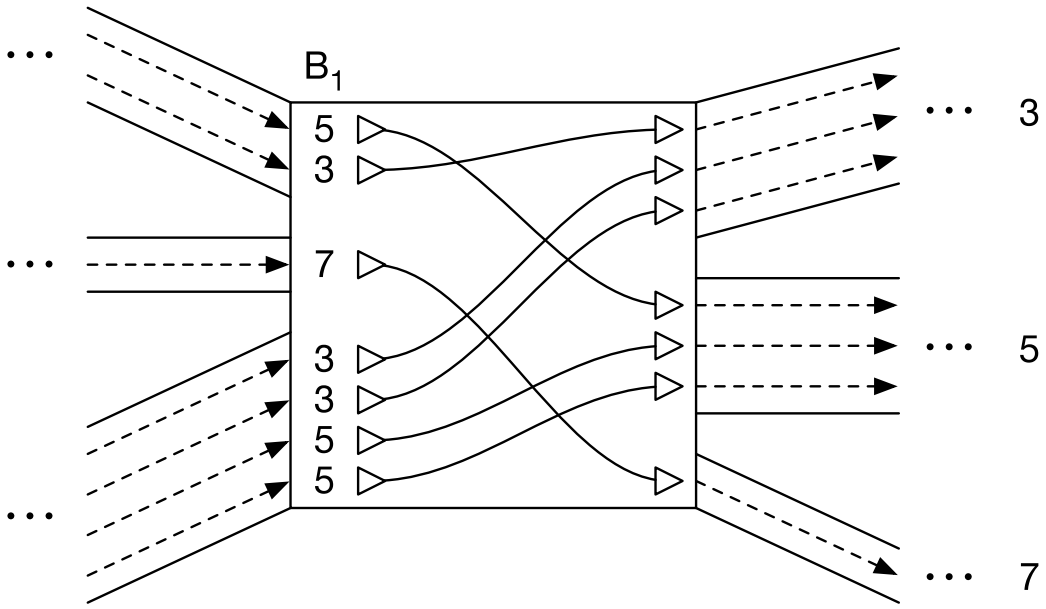
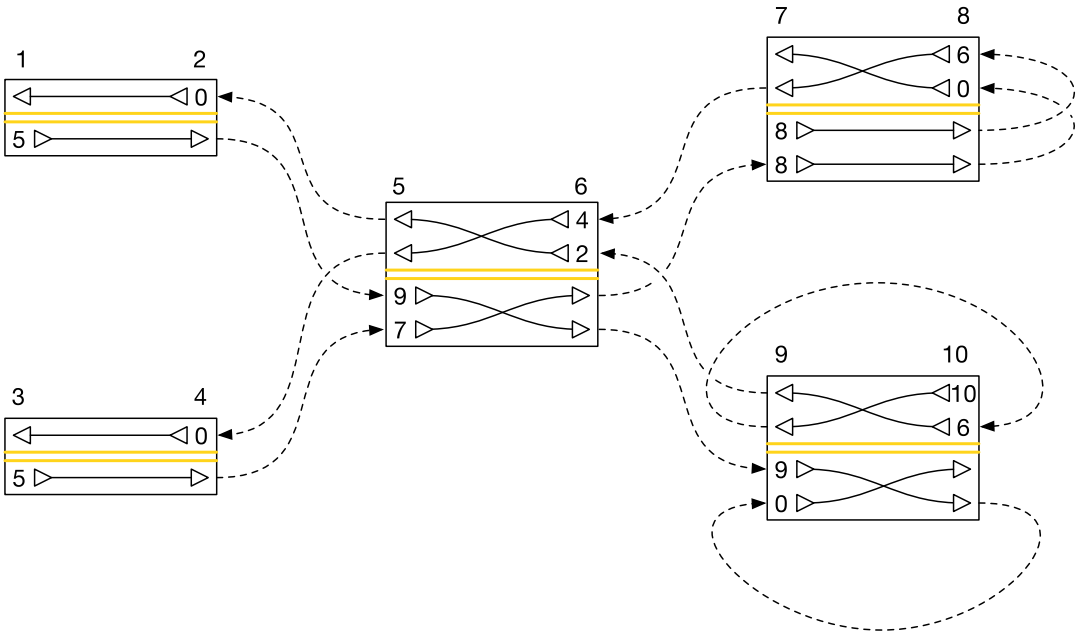

B1
5
3
7
3
3
5
5
3
5
7

Figure 3.2: A diagram of a graph containing two embedded threads. The graph consists of nodes with sides $\{1, 2, 3, \ldots, 10\}$, connected by edges $\{2, 5\}$, $\{4, 5\}$, $\{6, 7\}$, $\{6, 9\}$, $\{8, 8\}$, and $\{10, 9\}$. Note that, once again, odd numbers are used for left sides and even numbers are used for right sides. As in Figure 3.1, nodes are represented by rectangles, and thread orientations running from node to node are represented by dashed lines. The actual edges connecting the nodes are omitted for clarity; only the thread orientations are shown. Because each side's $B[]$ array defines a separate permutation, each node is divided into two parts by a central double yellow line (like on a road). The top half of each node shows visits to the node's right side, while the bottom half shows visits to the node's left side. Within the appropriate half of each node, the $B[]$ array entries for the entry side are shown. The special 0 value is used to indicate that a thread stops and does not continue on to another node. When moving from the entry side to the exit side of a node, threads cross over each other so that they become sorted, stably, by the side of their next visit. Threads' order of arrival at a node is determined by the relative order of the edges incident on the side they arrive at, which is in turn determined by the ordering of the sides on the other ends of the edges. The threads shown here are $[1, 2, 5, 6, 9, 10, 9, 10]$ and $[3, 4, 5, 6, 7, 8, 8, 7]$. See Table 3.1 for a tabular representation of this example.


1
2
0
5
3
4
0
5
5
6
4
2
9
7
7
8
6
0
8
8
9
10
10
6
9
0

this number b. Then, for i running from 0 to b, exclusive, begin a new thread orientation at $n(s)$ with the sides $[s, \overline{s}]$. Next, we traverse from $n(s)$ to the next node. Consult the $B_s[i]$ entry. If it is the null side, stop traversing, yield the thread orientation, and start again from the original node s with the next i value less than b. Otherwise, traverse to side $s' = B_s[i]$. Calculate the arrival index i' as $c(\overline{s}, s')$ plus the number of entries in $B_s[]$ before entry i that are also equal to s' (i.e. the s'-**rank** of i in $B_s[]$). This arrival index, computed by the WHERE_TO function in Algorithm 3.1, gives the index in $B_{\overline{s'}}[]$ of the next visit in the thread orientation being extracted. Then append s' and $\overline{s'}$ to the growing thread orientation, and repeat the traversal process with $i \leftarrow i'$ and $s \leftarrow s'$, until the terminating null side is reached.

This process will enumerate both orientations of each thread in the graph. The collection of observed orientations can trivially be converted to the collection of underlying ambisequence threads T, accounting for the fact that T may contain duplicate threads. Pseudocode for thread extraction is shown in Algorithm 3.1. The algorithm checks each side for threads, and traces each thread one at a time, doing a constant amount of work at each step (assuming a constant maximum degree for the graph). Therefore, the algorithm runs in $O(M \cdot N + S)$ time for extracting M threads of length N from a graph with S sides. Beyond the space used by the gPBWT itself, the algorithm uses $O(M \cdot N)$ memory, assuming the results are stored.

This algorithm works because the thread orientations embedded in the graph run through it in "ribbons" of several thread orientations with identical local history and a conserved relative ordering. The reverse prefix sort specified in the $B[]$ array definition causes thread orientations' visits to a side s that come after the same sequence of immediately prior visits to co-occur in a block in $B_s[]$. For any given next side s', or, equivalently, any edge $(\overline{s}, s')$, the visits to s' that come after visits in that block in $B_s[]$ will again occur together and in the same relative order in a block in $B_{s'}[]$. This is because the visits at side s' will share all the same history that the previous visits

shared at side s, plus a new previous visit to s that no other visits to s' can share. By finding a visit's index among the visits to s that next take the edge from $\overline{s}$ to s', and by using the $c()$ function to find where in $B_{s'}[]$ the block of visits that just came from s starts, one can find the entry in $B_{s'}[]$ corresponding to the next visit, and thus trace out the whole thread orientation from beginning to end.

Algorithm 3.1 Algorithm for extracting threads from a graph.

function STARTING_AT($Side$, G, $B[]$, $c()$)
  ▷ Count instances of threads starting at $Side$.
  ▷ Replace by an access to a partial sum data structure if appropriate.
  if $Side$ has incident edges **then**
    return $c(s, Side)$ for minimum s over all sides adjacent to $Side$.
  else
    return LENGTH($B_{Side}[]$)
function RANK($b[]$, $Index$, $Item$)
  ▷ Count instances of $Item$ before $Index$ in $b[]$.
  ▷ Replace by RANK of a rank-select data structure if appropriate.
  $Rank \leftarrow 0$
  for all index i in $b[]$ **do**
    if $b[i] = Item$ **then**
      $Rank \leftarrow Rank + 1$
  return $Rank$
function WHERE_TO($Side$, $Index$, $B[]$, $c()$)
  ▷ For a thread orientation visiting $Side$ with $Index$ in the reverse prefix sort order, get the corresponding sort index of the next visit in that thread orientation in the side it visits.
  ▷ Works by accounting for all thread orientations starting at the next side or entering the next side via edges before the edge being traversed, and then accounting for the thread orientation's rank among all thread orientations that similarly go from $Side$ to the same next side.
  return $c(\overline{Side}, B_{Side}[Index]) + \text{RANK}(B_{Side}[], Index, B_{Side}[Index])$
function EXTRACT(G, $c()$, $B[]$)
  ▷ Extract all oriented threads from graph G.
  for all Side s in G **do**
    $TotalStarting \leftarrow$ STARTING_AT$(s, G, B[], c())$
    for all i in $[0, TotalStarting)$ **do**
      $Side \leftarrow s$
      $Index \leftarrow i$
      $Orientation \leftarrow [s, \overline{s}]$
      $NextSide \leftarrow B_{Side}[Index]$
      while $NextSide \neq 0$ **do**
        $Orientation \leftarrow Orientation + [NextSide, \overline{NextSide}]$
        $Index \leftarrow$ WHERE_TO$(Side, Index, B[], c())$
        $Side \leftarrow NextSide$
        $NextSide \leftarrow B_{Side}[Index]$
      yield $Orientation$

3.5 Succinct Storage

For the case of storing haplotype threads specifically, we can assume that, because of linkage, many threads in T are identical local haplotypes for long runs, diverging from each other only at relatively rare crossovers or mutations. Because of the reverse prefix sorting of the visits to each side, successive entries in the $B[]$ arrays are thus quite likely to refer to locally identical haplotypes, and thus to contain the same value for the side to enter the next node on. Thus, the $B[]$ arrays should benefit from run-length compression. Moreover, since (as will be seen below) one of the most common operations on the $B[]$ arrays will be expected to be rank queries, a succinct representation, such as a collection of bit vectors or a wavelet tree [29], would be appropriate. To keep the alphabet of symbols in the $B[]$ arrays small, which is beneficial for such representations, it is possible to replace the stored sides for each $B_s[]$ with numbers referring to the edges traversed to access them, out of the edges incident to s.

We note that, for contemporary variant collections (e.g. the 1000 Genomes Project), the underlying graph G may be very large, while there may be relatively few threads (of the order of thousands) [3]. Implementers should thus consider combining multiple $B[]$ arrays into a single data structure to minimize overhead.

3.6 Embedding Threads

A trivial construction algorithm for the gPBWT is to independently construct $B_s[]$ and $c(s, s')$ for all sides s and oriented edges (s, s') according to their definitions above. However, this would be very inefficient. Here we present an efficient algorithm for gPBWT construction, in which the problem of constructing the gPBWT is reduced to the problem of embedding an additional thread.

Each thread is embedded by embedding its two orientations, one after the

other. To embed a thread orientation $t = [t_0, t_1, \ldots t_{2N}, t_{2N+1}]$, we first look at node $n(t_0)$, entering by t_0. We insert a new entry for this visit into $B_{t_0}[]$, lengthening the array by one. The location of the new entry is near the beginning, before all the entries for visits arriving by edges, with the exact location determined by the arbitrary order imposed on thread orientations. If no other order of thread orientations suggests itself, the order created by their addition to the graph will suffice, in which case the new entry can be placed at the beginning of $B_{t_0}[]$. The addition of this entry necessitates incrementing $c(s, t_0)$ by one for all oriented edges (s, t_0) incident on t_0 from sides s in G. We call the location of this entry k. The value of the entry will be t_2, or, if t is not sufficiently long, the null side, in which case we have finished the orientation.

If we have not finished the orientation, we first increment $c(s, t_2)$ by one for each side s adjacent to t_2 and after t_1 in the global ordering of sides. This updates the $c()$ function to account for the insertion into $B_{t_2}[]$ we are about to make. We then find the index at which the next visit in t ought to have its entry in $B_{t_2}[]$, given that the entry of the current visit in t falls at index k in $B_{t_0}[]$. This is given by the same procedure used to calculate the arrival index when extracting threads, denoted as WHERE_TO(t_1, k) (see Alg. 3.1). Setting k to this value, we can then repeat the preceding steps to embed t_2, t_3, etc. until t is exhausted and its embedding terminated with a null-side entry. Pseudocode for this process is shown in Algorithm 3.2.

As this algorithm proceeds, the $B[]$ arrays are always maintained in the correctly sorted order, because each insertion occurs at the correct location in the array. After each $B[]$ array insertion, the appropriate updates are made to the $c()$ function to keep it in sync with what is actually in the array. Thus, after each thread's insertion, the data structure correctly contains that thread, and so after the insertions of all the relevant threads, a properly constructed gPBWT is produced.

Assuming a dynamic succinct representation, where the $B[]$ array information is both indexed for $O(\log(n))$ rank queries and stored in such a way as to allow $O(\log(n))$

insertion and update (in the length of the array n)[2], this insertion algorithm is $O(N \cdot \log(N + E))$ in the length of the thread to be inserted (N) and the total length of existing threads (E). Inserting M threads of length N will take $O(M \cdot N \cdot \log(M \cdot N))$ time, and inserting each thread will take $O(N)$ memory in addition to the size of the gPBWT.

Algorithm 3.2 Algorithm for embedding a thread in a graph.

```
procedure INSERT(b[], Index, Item)
    ▷ Insert Item at Index in b[].
    ▷ Replace by INSERT of a rank-select-insert data structure if appropriate.
    LENGTH(b[]) ← LENGTH(b[]) + 1 ▷ Increase length of the array by 1
    for all i in (Index, LENGTH(b[]) − 1], descending do
        b[i] ← b[i − 1]
    b[Index] = Item
procedure INCREMENT_C(Side, NextSide, c())
    ▷ Modify c() to reflect the addition of a visit to the edge (Side, NextSide).
    for all side s adjacent to NextSide in G do
        if s > Side in side ordering then
            c(s, NextSide) ← c(s, NextSide) + 1
procedure EMBED(t, G, B[], c())
    ▷ Embed a thread orientation t in graph G.
    ▷ Call this twice to embed a thread for search in both directions.
    k ← 0 ▷ Index we are at in B_{t_2i}[]
    INCREMENT_C(0, t_0, c())
    ▷ Increment c() for all edges to t0, to note a thread start.
    for all i in [0, LENGTH(t)/2) do
        if 2i + 2 < LENGTH(t) then
            ▷ The thread has somewhere to go next.
            INSERT(B_{t_2i}[], k, t_{2i+2}) ▷ Fill in the B[] array slot for this visit.
            INCREMENT_C(t_{2i+1}, t_{2i+2}, c()) ▷ Record the traversal of the edge to the
next visit.
            k ← WHERE_TO(t_2i, k, B[], c())
        else
            INSERT(B_{t_2i}[], k, 0) ▷ End the thread.
```

[2]Dynamic data structures at least this good are available as part of the `DYNAMIC` library, from `https://github.com/xxsds/DYNAMIC`.

3.7 Batch Embedding Threads

The embedding algorithm described above, Algorithm 3.2, requires a dynamic implementation for the succinct data structure holding the $B[]$ array information, which can make it quite slow in practice due to the large constant factors involved. In order to produce a more practical implementation, it may be preferable to use a batch construction algorithm, which handles all threads together, instead of one at a time. For the case of directed acyclic graphs (DAGs), such an algorithm is presented here as Algorithm 3.3.

This algorithm works essentially like the naïve trivial algorithm of independently constructing every $B_s[]$ for every side s and every $c(s, s')$ for every oriented edge (s, s') from the definitions. However, because of the directed, acyclic structure of the graph, it is able to save redundant work on the sorting steps. Rather than sorting all the threads at each side, it sorts them where they start, and simply combines pre-sorted lists at each side to produce the $B[]$ array ordering, and then stably buckets threads into new sorted lists to pass along to subsequent nodes. The directed, acyclic structure allows us to impose a full ordering on the sides in the graph, so that the sorted lists required by a side all come from "previous" sides and are always available when the side is to be processed.

Although this algorithm requires that all threads be loaded into memory at once in a difficult-to-compress representation (giving it a memory usage of $O(M \cdot N)$ on M threads of length N), and although it requires that the graph be a directed acyclic graph, it allows the $B[]$ arrays to be generated for each side in order, with no need to query or insert into any of them. This means that no dynamic succinct data structure is required. Since the graph is acyclic, each thread can visit a side only once, and so the worst case is that a side is visited by every thread. Assuming a constant maximum degree for the graph, since the algorithm visits each side only once, the worst-case running time is $O(M \cdot S)$ for inserting M threads into a graph with S sides.

This algorithm produces the same gPBWT, in the form of the $B[]$ arrays and the $c()$ function, as the single-thread embedding algorithm would.

Algorithm 3.3 Algorithm for embedding all threads at once into a directed acyclic graph.

function BATCH_EMBED_INTO_DAG(T, G)
    ▷ Construct the gPBWT for threads T embedded in directed acyclic graph G.
    ▷ The forward orientation of each t must flow forwards through the forward orientation of G.
    Create empty $B_s[]$ for each side s in G
    Create empty $c()$
    for all o in [`FORWARD`, `REVERSE`] **do**
        $Messages \leftarrow []$
        $ThreadsByStart \leftarrow []$
        for all t in T **do**
            $t' \leftarrow t$ in orientation o
            $ThreadsByStart[t'_0] \leftarrow t'$
            INCREMENT_C$(0, t'_0, c())$
            ▷ Increment $c()$ for all edges to t'_0, to note a thread start.
        for all leading side s in G traversed in orientation o **do**
            $ThreadsHere \leftarrow []$
            for all t' in $ThreadsByStart[s]$ **do**
                $ThreadsHere \leftarrow ThreadsHere + [(t', 0)]$
            for all edge (s', s) in G, in order **do**
                ▷ Collect messages coming along edges to s.
                $ThreadsHere \leftarrow ThreadsHere + Messages[(s', s)]$
                $Messages[(s', s)] \leftarrow []$
            for all (t', n) at index i in $ThreadsHere$ **do**
                $n \leftarrow n + 1$
                if LENGTH$(t') > n * 2$ **then**
                    $NextSide \leftarrow t'[n * 2]$
                    $Messages[(\overline{s}, NextSide)] \leftarrow Messages[(\overline{s}, NextSide)] + [(t', n)]$
                    INCREMENT_C$(\overline{s}, NextSide, c())$
                else
                    $NextSide \leftarrow 0$
                $B_s[i] \leftarrow NextSide$
    return $B[], c()$

3.8 Counting Occurrences of Subthreads

The generalized PBWT data structure presented here preserves some of the original PBWT's efficient haplotype search properties [18]. The algorithm for counting all occurrences of a new thread orientation t as a subthread of the threads in T runs as follows.

We define f_i and g_i as the first and past-the-last indexes for the range of visits of orientations of threads in T to side t_{2i}, ordered as in $B_{t_{2i}}[]$.

For the first step of the algorithm, f_0 and g_0 are initialized to 0 and the length of $B_{t_0}[]$, respectively, so that they select all visits to node $n(t_0)$, seen as entering through t_0. On subsequent steps, f_{i+1} and g_{i+1}, are calculated from f_i and g_i merely by applying the WHERE_TO function (see Alg. 3.1). We calculate $f_{i+1} = \text{WHERE\_TO}(t_{2i}, f_i)$ and $g_{i+1} = \text{WHERE\_TO}(t_{2i}, g_i)$.

This process can be repeated until either $f_{i+1} \geq g_{i+1}$, in which case we can conclude that the threads in the graph have no matches to t in its entirety, or until t_{2N}, the last entry in t, has its range f_N and g_N calculated, in which case $g_N - f_N$ gives the number of occurrences of t as a subthread in threads in T. Moreover, given the final range from counting the occurrences for a thread t, we can count the occurrences of any longer thread that begins (in its forward orientation) with t, merely by continuing the algorithm with the additional entries in the longer thread.

This algorithm works because the sorting of the $B[]$ array entries by their history groups entries for thread orientations with identical local histories together into contiguous blocks. On the first step, the block for just the orientations visiting the first side is selected, and on subsequent steps, the selected block is narrowed to just the orientations that visit the current side and which share the sequence of sides we have previously used in their history. The WHERE_TO function essentially traces where the first and last possible consistent thread orientations would be inserted in the next $B[]$

array, and so produces the new bounds at every step.

Assuming that the $B[]$ arrays have been indexed for $O(1)$ rank queries (which is possible using available succinct data structure libraries such as [28], when insert operations are not required), the algorithm is $O(N)$ in the length of the subthread t to be searched for, and has a runtime independent of the number of occurrences of t. It can be performed in a constant amount of memory ($O(1)$) in addition to that used for the gPBWT. Pseudocode is shown in Algorithm 3.4.

Algorithm 3.4 Algorithm for searching for a subthread in the graph.

```
function COUNT(t, G, B[], c())
    ▷ Count occurrences of subthread t in graph G.
    f ← 0
    g ← LENGTH(B_{t_0}[])
    for all i in [0, LENGTH(t)/2 − 1) do
        f ← WHERE_TO(t_{2i}, f, B[], c())
        g ← WHERE_TO(t_{2i}, g, B[], c())
        if f ≥ g then
            return 0
    return g − f
```

3.9 Results

The gPBWT was implemented within `xg`, the succinct graph indexing component of the `vg` variation graph toolkit [24]. The primary succinct self-indexed data structure used, which compressed the gPBWT's $B[]$ arrays, was a run-length-compressed wavelet tree, backed by sparse bit vectors and a Huffman-shaped wavelet tree, all provided by the `sdsl-lite` library used by `xg` [28]. The $B[]$ arrays, in this implementation, were stored as small integers referring to edges leaving each node, rather than as full next-side IDs. The $c()$ function was implemented using two ordinary integer vectors, one storing the number of threads starting at each side, and one storing the number of threads using each side and each oriented edge. Due to the use of `sdsl-lite`, and the poor constant-factor performance of dynamic alternatives, efficient integer vector

insert operations into the $B[]$ arrays were not possible, and so the batch construction algorithm (Alg. 3.3), applicable only to directed acyclic graphs, was implemented. A modified release of `vg`, which can be used to replicate the results shown here, is available from `https://github.com/adamnovak/vg/releases/tag/gpbwt2`.

The modified `vg` was used to create a genome graph for human chromosome 22, using the 1000 Genomes Phase 3 VCF on the hg19 assembly, embedding information about the correspondence between VCF variants and graph elements [3]. Note that the graph constructed from the VCF was directed and acyclic; it described only substitutions and indels, with no structural variants, and thus was amenable to the batch gPBWT construction algorithm. Next, haplotype information for the 5,008 haplotypes stored in the VCF was imported and stored in a gPBWT-enabled `xg` index for the graph, using the batch construction algorithm mentioned above. In some cases, the VCF could not be directly translated into self-consistent haplotypes. For example, a `G` to `C` SNP and a `G` to `GAT` insertion might be called at the same position, and a haplotype might claim to contain the alt alleles of both variants. A naïve interpretation might have the haplotype visit the `C` and then the `GAT`, which would be invalid, because the graph would not contain the `C` to `G` edge. In cases like this, an attempt was made to semantically reconcile the variants automatically (in this case, as a `C` followed by an `AT`), but this was only possible for some cases. In other cases, invalid candidate haplotype threads were still generated. These were then split into valid pieces to be inserted into the gPBWT. Threads were also split to handle other exceptional cases, such as haploid calls in the input. Overall, splitting for causes other than loss of phasing occurred 203,145 times across the 5,008 haplotypes, or about 41 times per haplotype.

The `xg` indexing and gPBWT construction process took 9 hours and 19 minutes using a single indexing thread on an Intel Xeon X7560 running at 2.27 GHz, and consumed 278 GB of memory. The high memory usage was a result of the decision to retain the entire data set in memory in an uncompressed format during construction.

However, the resulting `xg` index was 436 MB on disk, of which 321 MB was used by the gPBWT. Information on the 5,008 haplotypes across the 1,103,547 variants was thus stored in about 0.93 bits per diploid genotype in the succinct self-indexed representation, or 0.010 bits per haplotype base. [3] Extrapolating linearly from the 51 megabases of chromosome 22 to the entire 3.2 gigabase human reference genome, a similar index of the entire 1000 Genomes dataset would take 27 GB, with 20 GB devoted to the gPBWT. This is well within the storage and memory capacities of modern computer systems.

3.9.1 Random Walks

The query performance of the gPBWT implementation was evaluated using random walk query paths. 1 million random walks of 100 bp each were simulated from the graph. To remove walks covering ambiguous regions, walks that contained two or more `N` bases in a row were eliminated, leaving 686,590 random walks. The number of haplotypes in the gPBWT index consistent with each walk was then determined, taking 61.29 seconds in total using a single query thread on the above-mentioned Xeon system. The entire operation took a maximum of 458 MB of memory, indicating that the on-disk index did not require significant expansion during loading to be usable. Overall, the gPBWT index required 89.3 microseconds per count operation on the 100 bp random walks. It was found that 316,078 walks, or 46%, were not consistent with any haplotype in the graph. The distribution of of the number of haplotypes consistent with each random walk is visible in Figure 3.3.

3.9.1.1 Read Alignments

To further evaluate the performance of the query implementation, we evaluated read alignments to measure their consistency with stored haplotypes. 1000 Genomes Low

[3] The improved size results here relative to the results in our conference paper are related to the use of a new run-length-compressed storage backend for the $B[]$ arrays, replacing one that was previously merely succinct [78].

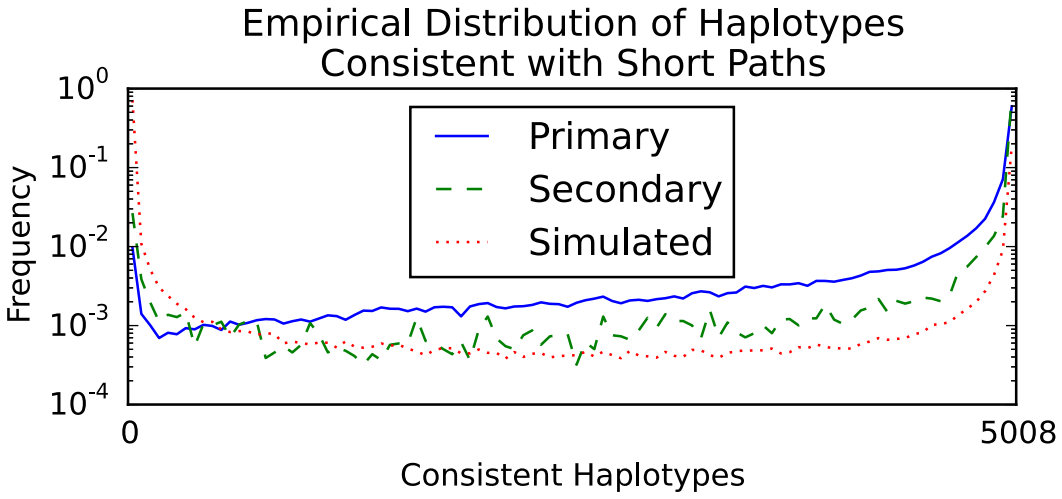
Coverage Phase 3 reads for NA12878 that were mapped in the official alignment to chromosome 22 were downloaded and re-mapped to the chromosome 22 graph, using the `xg`/GCSA2-based mapper in `vg`, allowing for up to a single secondary mapping per read. (The `vg` aligner was chosen because of its ease of integration with our `xg`-based gPBWT implementation, but in principle any aligner that supports aligning to a graph could be used.) The mappings with scores of at least 90 points out of a maximum of 101 points (for a perfectly-mapped 101 bp read) were selected (thus filtering out alignments highly like to be erroneous) and broken down into primary and secondary mappings. The number of haplotypes in the gPBWT index consistent with each read's path through the graph was calculated (Fig. 3.3). For 1,500,271 primary mappings, the count operation took 150.49 seconds in total, or 100 microseconds per mapping, using 461 MB of memory. (Note that any approach that depended on visiting each haplotype in turn, such as aligning each read to each haplotype, would have to do its work for each read/haplotype combination in less than 20 microseconds, or about 45 clock cycles, in order to beat this time.) It was found that 2,521 of these primary mappings, or 0.17%, and 320 of 43,791 secondary mappings, or 0.73%, were not consistent with any haplotype path in the graph. [4] These read mappings, despite having reasonable edit based scores, may represent rare recombinations, but the set is also likely to be enriched for spurious mappings.

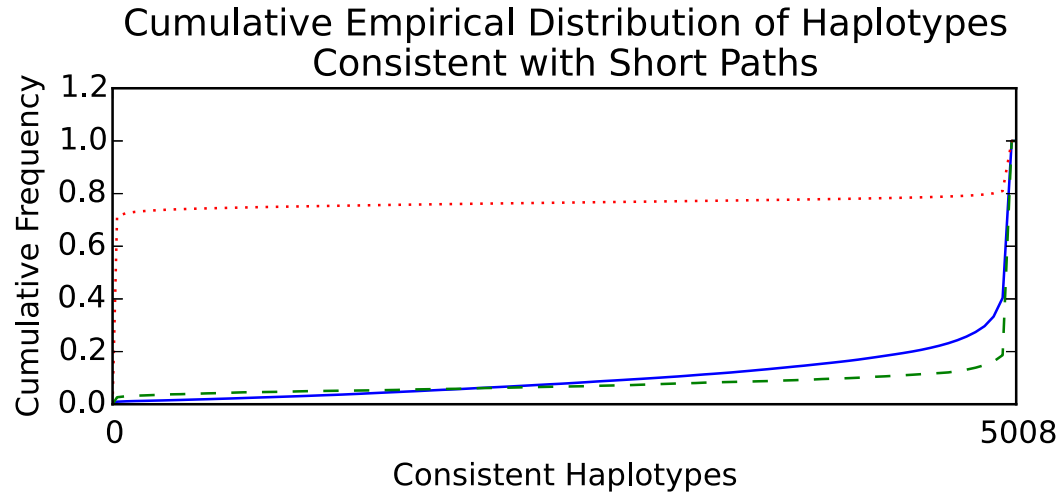
3.9.2 Scaling Characteristics

To evaluate the empirical space usage scaling characteristics of our gPBWT implementation, a scaling experiment was undertaken. The 1000 Genomes VCFs Phase 3 VCFs for the GRCh38 assembly were downloaded, modified to express all variants on the forward strand in the GRCh38 assembly, and used together with the assembly data to

[4] These numbers are expected to differ from those reported in our conference paper due to improvements to the `vg` mapping algorithms since the conference paper was prepared [78].

Figure 3.3: Distribution (top) and cumulative distribution (bottom) of the number of 1000 Genomes Phase 3 haplotypes consistent with short paths in the hg19 chromosome 22 graph. Primary mappings of 101 bp reads with scores of 90 out of 101 or above ($n = 1,500,271$) are the solid blue line. Secondary mappings meeting the same score criteria ($n = 43,791$) are the dashed green line. Simulated 100 bp random walks in the graph without consecutive `N` characters ($n = 686,590$) are the dotted red line. Consistent haplotypes were counted using the gPBWT support added to `vg` [24].


Empirical Distribution of Haplotypes
Consistent with Short Paths
Frequency
10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
Primary
Secondary
Simulated
0
5008
Consistent Haplotypes


Cumulative Empirical Distribution of Haplotypes
Consistent with Short Paths
Cumulative Frequency
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
5008
Consistent Haplotypes

produce a graph for chromosome 22 based on the newer assembly. This graph was then used to construct a gPBWT with progressively larger subsets of the available samples. Samples were selected in the order they appear in the VCF file. For each subset, an `xg` serialization report was generated using the `xg` tool, and the number of bytes attributed to "threads" was recorded. The number of bytes used versus the number of samples stored is displayed in Figure 3.4.

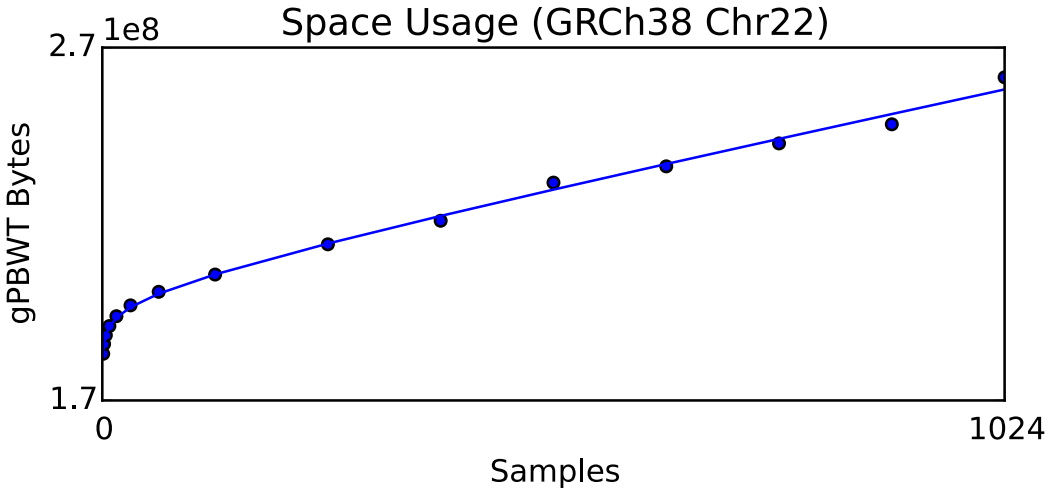


Figure 3.4: Disk space usage for the gPBWT versus sample count for GRCh38 chromosome 22. Points are sampled at powers of two up to 128, and intervals of 128 thereafter up to 1024. The trend line shown corresponds to the function $y(x) = 3.16 \times 10^6 \text{ bytes} \cdot \ln(x/\text{samples}) + 5.12 \times 10^4 \frac{\text{bytes}}{\text{sample}} \cdot x + 1.84 \times 10^8 \text{ bytes}$.

After empirical size data was obtained, a log-plus-linear curve, consisting of a log component and a linear component, was fit to the data. This curve was used to extrapolate an estimated size of 5.34 GB on disk for the storage of 100,000 samples' worth of data on chromosome 22. We choose 100,000 because it is representative of the scale of large contemporary sequencing projects, such as Genomics England's 100,000 Genomes Project (`https://www.genomicsengland.co.uk/the-100000-genomes-project/`) [76] and the NHLBI's TOPMed program (`https://www.nhlbi.nih.gov/rese`

arch/resources/nhlbi-precision-medicine-initiative/topmed). Linear extrapolation from the 51 megabase chromosome 22 to the 3.2 gigabase human genome yields a size estimate of 336 GB for the storage of 100,000 diploid genomes, in addition to the space usage of the underlying graph. Although this extrapolation does not account for the dependence of graph complexity on the number of samples sequenced, it suggests that the gPBWT is capable of scaling to the anticipated size of future sequencing data sets, while using currently available computing resources.

3.10 Discussion

We have introduced the gPBWT, a graph based generalization of the PBWT. We have demonstrated that a gPBWT can be built for a substantial genome graph (all of human chromosome 22 and the associated chromosome 22 substitutions and indels in 1000 Genomes). Using this data structure, we have been able to quickly determine that the haplotype consistency rates of random walks and primary and secondary read mappings differ substantially from each other, and based on the observed distributions we hypothesize that consistency with very few haplotypes can be a symptom of a poor alignment.

Such poor alignments could arise by a variety of means, including similarity between low complexity sequence, or paralogy; the latter representing true sequence homology but not true sequence orthology. Paralogous alignments are often difficult to distinguish from truly orthologous alignments, and can lead to the reporting of false or misplaced variants. Using haplotype consistency information is one way we might better detect paralogy, because paralogous sequence is not expected to be consistent with linkage relationships at a paralogous site. A more sophisticated analysis of haplotype consistency rate distributions could thus improve alignment scoring.

In the present experiment, we have examined only relatively simple variation:

substitutions and short indels. Instances of more complex variation, like large inversions and translocations, which would have induced cycles in our genome graphs, were both absent from the 1000 Genomes data set we used and unsupported by the optimized DAG-based construction algorithm which we implemented. We expect that complex structural variation is well suited to representation as a genome graph, so supporting it efficiently should be a priority for a serious practical gPBWT construction implementation.

Extrapolating from our results on chromosome 22, we predict that a whole-genome gPBWT could be constructed for all 5,008 haplotypes of the 1000 Genomes data on GRCh37 and stored in the main memory of a reasonably apportioned computer, using about 27 GB of memory for the final product. On the GRCh38 data set, we extrapolate a space usage of 21 GB for the 2,504 samples of the 1,000 Genomes Project; A whole-genome gPBWT for 100,000 samples on GRCh38, we predict, could be stored in about 336 GB. Computers with this amount of memory, though expensive, are readily available from major cloud providers. (The wasteful all-threads-in-memory construction implementation we present here, however, would not be practical at such a scale, requiring on the order of 50 TB of memory to handle 100,000 samples when constructing chromosome 1; a disk-backed implementation or other low-memory construction algorithm would be required.) The relatively modest growth from 5,008 haplotypes (2,504 samples) to 200,000 haplotypes (100,000 samples) is mostly attributable to the run-length compression used to store the B arrays in our implementation. Each additional sample is representable as a mere increase in run lengths where it agrees with previous samples, and contributes an exponentially diminishing number of new variants and novel linkage patterns. While further empirical experimentation will be necessary to reasonably extrapolate further, it does not escape our notice that the observed scaling patterns imply the practicality of storing cohorts of a million or more individuals, such as those envisaged by the Precision Medicine Initiative [39] and other similar national

efforts, within an individual powerful computer. Looking forward, this combination of genome graph and gPBWT could potentially enable efficient mapping not just to one reference genome or collapsed genome graph, but simultaneously to an extremely large set of genomes related by a genome graph.

3.11 List of Abbreviations

- **BWT**: Burrows-Wheeler Transform
- **PBWT**: Positional Burrows-Wheeler Transform
- **gPBWT**: Graph Positional Burrows-Wheeler Transform
- **GRC**: Genome Reference Consortium
- **GRCh37**: GRC human genome assembly, build 37
- **GRCh37**: GRC human genome assembly, build 38
- **DAG**: Directed Acyclic Graph

3.12 Declarations

3.12.1 Ethics approval and consent to participate

All human data used in this study comes from already published, fully public sources, namely the 1000 Genomes Project and the human reference assembly. We believe that the work performed in this study is consistent with the purpose for which these data resources were created, and that the original ethical reviews of the creation and publication of these data resources, and the consent assertions given to the original projects, are sufficient to cover this new work.

3.12.2 Consent for publication

Not applicable

3.12.3 Availability of data and material

The datasets analyzed during the current study are available in the 1000 Genomes repository, at ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20130502/ALL.chr22.phase3_shapeit2_mvncall_integrated_v5a.20130502.genotypes.vcf.gz (md5 ad7d6e0c05edafd7faed7601f7f3eaba), ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20130502/ALL.chr22.phase3_shapeit2_mvncall_integrated_v5a.20130502.genotypes.vcf.gz.tbi (md5 4202e9a481aa8103b471531a96665047), ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/technical/reference/phase2_reference_assembly_sequence/hs37d5.fa.gz (md5 a07c7647c4f2e78977068e9a4a31af15), and ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20130502/supporting/GRCh38_positions/ALL.chr22.phase3_shapeit2_mvncall_integrated_v3plus_nounphased.rsID.genotypes.GRCh38_dbSNP_no_SVs.vcf.gz (md5 cf7254ef5bb6f850e3ae0b48741268b0), and in the GRCh38 assembly repository, at ftp://ftp.ncbi.nlm.nih.gov/genomes/all/GCA/000/001/405/GCA_000001405.15_GRCh38/GCA_000001405.15_GRCh38_assembly_structure/Primary_Assembly/assembled_chromosomes/FASTA/chr22.fna.gz (md5 915610f5fb9edfcc9ce477726b9e72c6).

3.13 Competing interests

The authors declare that they have no competing interests.

3.13.1 Funding

This work was supported by the National Human Genome Research Institute of the National Institutes of Health under Award Number 5U54HG007990, the W.M. Keck foundation under DT06172015, the Simons Foundation under SFLIFE# 351901, the ARCS Foundation, and Edward Schulak. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health or any other funder.

3.14 Author's contributions

A.M.N. wrote most of the gPBWT implementation presented here, conducted the experiments, and composed the majority of the manuscript. E.G. managed the `vg` project, wrote the read simulation and mapping code used here, and collaborated on the gPBWT implementation. B.P. developed the mathematics of the gPBWT and collaborated on the manuscript.

3.15 Acknowledgements

We would like to thank Richard Durbin for inspiration, David Haussler for his extremely helpful comments on the manuscript, and Jordan Eizenga for additional helpful comments on manuscript revisions.

Chapter 4

Genome Graphs

This chapter has been adapted from the article Novak et al. [79], and contains material attributable to all authors of that work. Supplementary materials referenced here are available in the online version of that article.

4.1 Abstract

There is increasing recognition that a single, monoploid reference genome is a poor universal reference structure for human genetics, because it represents only a tiny fraction of human variation. Adding this missing variation results in a structure that can be described as a mathematical graph: a genome graph. We demonstrate that, in comparison to the existing reference genome (GRCh38), genome graphs can substantially improve the fractions of reads that map uniquely and perfectly. Furthermore, we show that this fundamental simplification of read mapping transforms the variant calling problem from one in which many non-reference variants must be discovered de-novo to one in which the vast majority of variants are simply re-identified within the graph. Using standard benchmarks as well as a novel reference-free evaluation, we show that a simplistic variant calling procedure on a genome graph can already call variants at least as well as,

and in many cases better than, a state-of-the-art method on the linear human reference genome. We anticipate that graph-based references will supplant linear references in humans and in other applications where cohorts of sequenced individuals are available.

4.2 Introduction

The human reference genome, completed in draft form in 2001 and revised several times subsequently [8, 41], is the single most important resource used in human genetics today. It acts as a universal coordinate system and as such is the space in which annotations (genes, promoters, etc.) and genetic variants are described [3, 32, 108]. It is also the target for read mapping, and, downstream of this mapping, is used for functional assays and variant calling pipelines [15, 53].

The contemporary definition of a reference genome is completely linear: a single monoploid assembly of the genome of a species. A key limitation of the linear human reference genome (the set of chromosome scaffolds) is that it is but a single genome. As such, it is an imperfect lens through which to study our population's variation; there exist variants and annotations that can not be easily described with respect to the reference genome [36, 86]. Furthermore, as a target for mapping and interpretation it introduces a reference allele bias: a tendency to over-report alleles present in the reference genome and under-report other alleles [5, 13]. To mitigate these issues, recent versions of the reference genome assembly, such as GRCh38, have contained "alternate locus" sequences ("alts"): extra sequence representations of regions of the human genome considered to be highly polymorphic, anchored at their ends to locations within the "primary" (monoploid) reference assembly. Such a structure, which contains multiple partially-overlapping sequence paths, can be considered a form of mathematical graph. The explicit use of graphs in biological sequence analysis has a long history, notably for sequence alignment [83], sequence assembly [75, 87], assembly

representation (as in FASTG and now GFA)[27, 109], substring indexes (which are often thought of in terms of suffix trees or similar data structures) [53, 98], and transcript splice graphs [34]. Recently the notion of graphs for representing genomes has been considered explicitly [16, 63, 83], and work has been done towards using these graphs as references [58]. The alternate loci currently used are just one way to extend the linear reference genome into a genome graph; many other ways are possible. In this work, conducted by a task team of the Global Alliance for Genomics and Health, we experiment with different methods for graph construction and testing the utility of different graphs for read mapping and variant calling. This work is the first study of its kind that we are aware of. We attempt to test the simple hypothesis that adding data into the reference structure—in effect, adding to the "reference prior" on variation extant in the population—will result in improved genome inferences.

4.3 Results

There are many possible types of genome graph; here we use *sequence graphs*. The nodes of a sequence graph are a set of DNA sequences. Each node is therefore a string of nucleotide characters, called positions, giving the sequence of the node's forward strand. We call the terminal 5' and 3' ends of this strand the *sides* of the node. Each edge in the graph is an unordered pair of sides, representing a (potential) bond between two sides of a pair of nodes. This is a bidirected graph representation, because features of the edge indicate to which side of a node (sequence), 5' or 3', each end of the edge connects (Fig. 4.1)[69]. Other representations of genome graphs, such as the directed acyclic representation, can be useful; see Supplementary Section 1. A longer DNA sequence can be represented as a *thread* within a sequence graph, beginning in one oriented node, ending in the same node or another, and in between walking from node to node, with the rule that if the walk enters a node on one side it exits through the

other side.

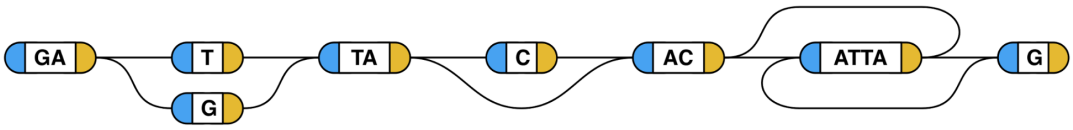


Figure 4.1: Example sequence graphs. Each node holds a string of bases. An edge can connect, at each of its ends, to a base on either the left (5', blue) or the right (3', yellow) side of the node. When reading through a thread to form a DNA sequence, a valid walk must leave each node via the opposite side from that through which it was entered; a node's sequence is read as reverse-complemented if the node is entered on the 3' side. One thread that this graph spells out (reading from the left side of the leftmost sequence to the right side of the rightmost sequence, along the nodes drawn in the middle) is the sequence "GATTACACATTAG". Straying from this path, there are three variants available: a substitution of "G" for "T", a deletion of a "C", and an inversion of "ATTA". If all of these detours are taken, the sequence produced is "GAGTAACTAATG". All 8 possible threads from the leading G to the trailing G are allowed.

To evaluate the utility of sequence graphs we invited teams to construct and evaluate graphs for five test regions of the human genome: the major histocompatibility complex (MHC), the killer cell immunoglobulin-like receptors (LRC_KIR) region, the spinal muscular atrophy (SMA) locus, and the BRCA1 and BRCA2 genes. MHC, SMA and LRC_KIR are all regions with alternate loci represented in GRCh38, while BRCA1 and BRCA2 represent more typical human genes. Regions ranged from 81 kilobases in size with a single gene (BRCA1) to 5.0 megabases in size with 172 genes (MHC). We considered graphs from five teams built with eight different pipelines (Table 4.2). For each region we provided a set of long, high quality input sequences from which to construct graphs (Table 4.1), but also encouraged the creation of graphs using additional data of the builder's choice. Some graphs were built based upon existing variant calls, such as the 1000 Genomes calls used to construct the 1KG graph [3]. Graphs were also built with a wide variety of different algorithmic approaches (Table 4.2). Three control graphs were constructed for each region as points of comparison. The Primary graphs

Region	Chromosome	Length in Primary Reference (bp)	GRCh38 Coordinates	Number of Genes	Alt Haplotypes in pilot data
BRCA1	17	81,189	43044293-43125482	1	2
BRCA2	13	84,989	32314860-32399849	1	2
LRC_KIR	19	1,058,685	54025633-55084318	47	35
MHC	6	4,970,458	28510119-33480577	172	8
SMA	5	2,397,625	69216818-71614443	21	2

Table 4.1: Pilot Regions. Selected test cases represent a sampling of both typical and challenging genomic regions.

contain just the single, linear reference path from GRCh38. The Unmerged graphs consist of just the set of provided sequences, each represented as a disjoint path. The Scrambled graphs (see Online Methods) are essentially identical topologically to the 1KG graphs, but with structures shifted to create false variants. These graphs acted as a negative control for the effects of adding nonsense variation to the graphs.

4.3.1 Graph Read Mapping

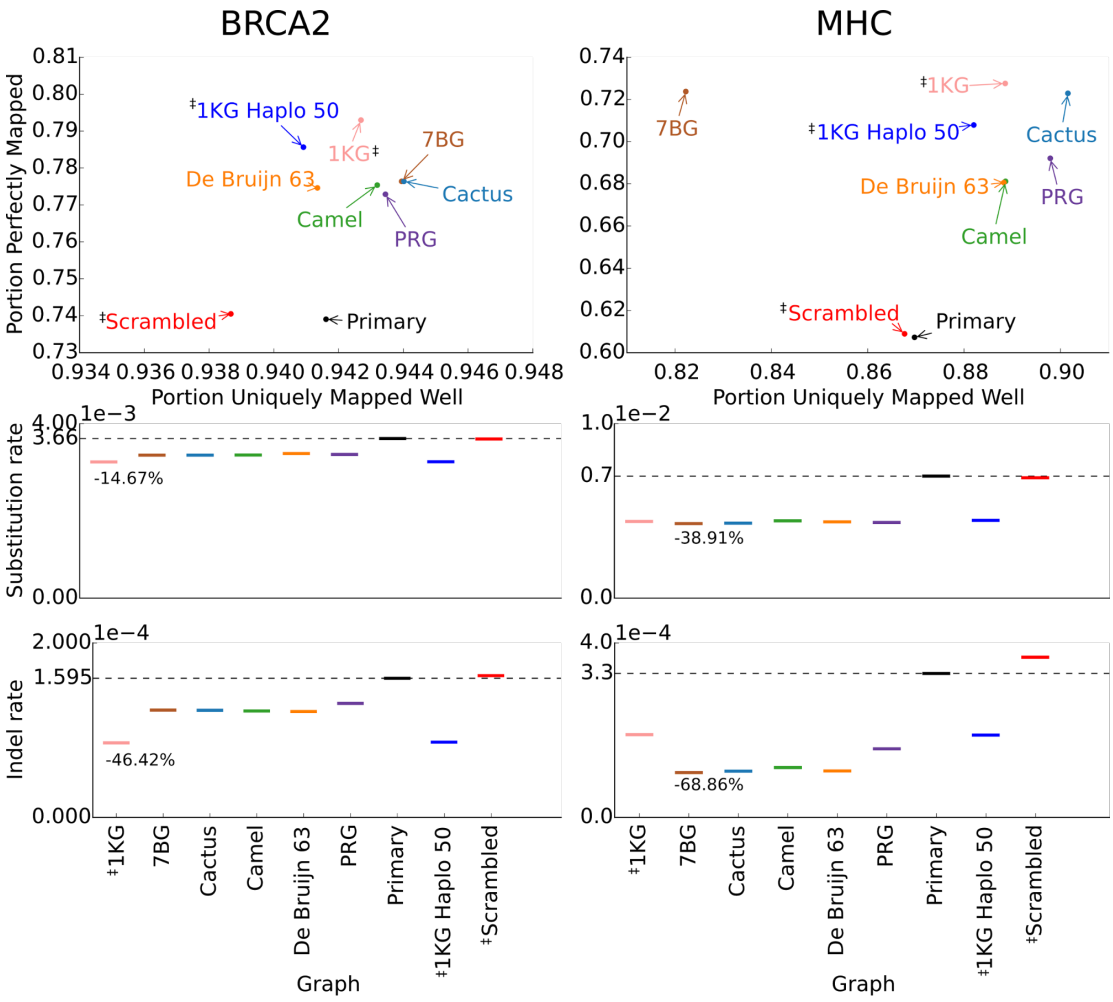
To evaluate the utility of sequence graphs for read mapping we used the software program vg[107], which contains a mapping algorithm capable of mapping to a fully general and potentially cyclic sequence graph (see Supplementary Section 2). We mapped all relevant reads (see Online Methods) from 1000 Genomes Phase 3 low coverage samples to each graph. We found that vg was able to map almost all such reads to the graphs (Supplementary Fig. 3).

Relative to the primary graph, a graph containing more of the variants should produce an increase in the fraction of reads that map perfectly (without either substitutions or indels) to at least one place. For BRCA2 we see a relative increase of 7.3%

Table 4.2: Genome Graph Submissions. Submissions were collected from a variety of institutions, and showcase a variety of graph construction methods.

Submissions using pilot data			
Submission	**Team**	**Short Name**	**Description of Algorithm**
Cactus	UCSC	Cactus	Graph-based multiple sequence aligner [82].
Camel	UCSC	Camel	Creates graphs progressively by mapping using context schemes [77].
De Bruijn Graph (k=63)	MSKCC	De Bruijn 63	Forms a De Bruijn graph of input data with k=63, then converts to a sequence graph.
Population Reference Graph	Oxford	PRG	Creates a graph from a K-mer-based HMM description of a region [16].
Seven Bridges	Seven Bridges	7BG	Multiple genome alignment.
Submissions using other data			
Submission	**Team**	**Short Name**	**Description of Algorithm**
1000 Genomes SNP Graph	Sanger/ UCSC	1KG	Generated using vg construct on a VCF containing variants identified in the 1000 genomes project. Platinum genome samples were not included, to avoid circularity in variant evaluation.
1000 Genomes Haplotype 50	Sanger/ UCSC	1KG Haplo 50	Adapted form of 1KG graph in which phasing information is used to reduce the number of unobserved recombinations represented by the graph. 50 is the number of bases two variants can be apart to be considered for this phasing.
Scrambled 1000 Genomes	Sanger/ UCSC	Scrambled	Generated by shifting all the variants in the standard 1KG graph 200 bp downstream.

Figure 4.2: Mapping reads to sequence graphs. Results for the 1000 Genomes Phase 3 low coverage samples against the BRCA2 and MHC graphs. The median per-sample portion of reads that are mapped perfectly (Y axis), and the median per-sample portion of reads that are mapped with a unique, obviously-best alignment (X axis) are both visible in the top row. The median per-sample substitution rate for a primary mapping, computed per aligned base, is shown in the second row. The median per sample frequency of indels in primary mapped reads, computed per read base, is given in the third row. The horizontal black line represents the result for the primary reference graph in the region. The symbol marks graphs generated using additional data beyond the provided reference and alternate sequences. The unmerged graphs are excluded because very few reads mapped uniquely to them.


BRCA2
MHC
Portion Perfectly Mapped
Portion Uniquely Mapped Well
‡1KG Haplo 50
1KG‡
7BG
De Bruijn 63
Cactus
Camel
PRG
‡Scrambled
Primary
Substitution rate
1e−3
1e−2
-14.67%
-38.91%
Indel rate
1e−4
-46.42%
-68.86%
‡1KG
7BG
Cactus
Camel
De Bruijn 63
PRG
Primary
‡1KG Haplo 50
‡Scrambled
Graph

in the median number of reads mapping perfectly to the 1KG graph vs. the Primary graph, but for MHC the increase is 20% (Fig. 4.2 top row, Supplementary Section 3, Supplementary Fig. 1). The increase for BRCA2 is close to what would be expected if the sequence graph contained the majority of polymorphisms for a typical region of the genome (Supplementary Section 3), while the larger increase for MHC is likely due to a greater degree of polymorphism [5]. Similar, slightly smaller increases are seen for graphs built from other, smaller collections of variants. The scrambled graphs do not show significant gains, thus indicating that the effect is specific to graphs containing known variation. Furthermore, the overall substitution rate between reads and the experimental graphs was observed to decrease, relative to the rate between the reads and the Primary control graph. In the highest-performing graphs the decline is slightly below the bounds of previous read substitution error rate estimates of 0.7-1.6% [1–3, 74] (Fig. 4.2 second row; see Supplementary Section 4 and Supplementary Fig. 4). The decrease in indel rate (Fig. 4.2 third row) moving from the Primary graph to the 1KG graph is comparable to estimates of the human indel polymorphism rate[73] (Supplementary Section 5).

The median fraction of reads that uniquely map increases for many of the graphs, relative to the primary and scrambled graphs. For example, in the Cactus graph, an increase of 0.26% is observed in BRCA2, and an increase of 3.7% is observed in the MHC. No such increase in unique mapping is seen for the comparably complex scrambled graph. Unique mapping is defined as having a good primary mapping and no reasonable secondary mapping (see Supplementary Section 3 and Supplementary Fig. 2).

To test if the choice of sequence graph reference affected population level reference allele bias, we binned samples by 1KG super-population and looked at the difference in perfect mapping between the 1KG graph and the Primary graph for each subpopulation. We find a small but significant difference in perfect mapping increase

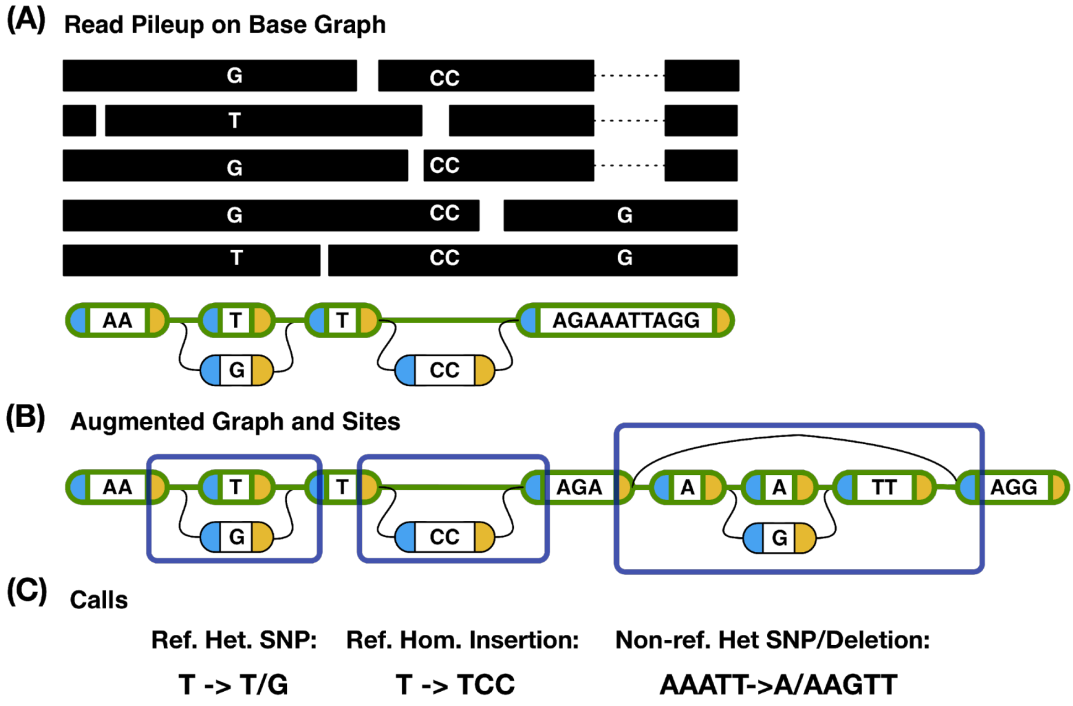
between super-populations for most regions (Supplementary Section 6, Supplementary Fig. 5), but we also find relatively large differences in absolute rates of perfect mapping (Supplementary Fig. 6). These latter differences suggest that super-population may be confounded with sequencing batch, making drawing conclusions from this analysis quite difficult.

4.3.2 Graph Variant Calling

We implemented a comprehensive, albeit basic, variant calling pipeline based on the Samtools pileup approach [57], modified to work with sequence graphs (see Online Methods for more details). In summary (Fig. 4.3), reads are mapped to the reference graph or *base graph* and a pileup is computed at each graph position and edge. An *augmented graph* is created by extending the base graph with additional sequences and edges representing possible variants determined from the pileups. This graph is then analyzed for *ultrabubbles* (acyclic, tip-free subgraphs connected to the rest of the graph by at most 2 nodes) which are treated as sites for genotyping[85]. Finally, thresholding heuristics are used to emit a set of genotypes with sufficient read support, one for each site, expressed in the coordinates of the GRCh38 primary reference path as embedded in the graph (see Online Methods).

We compared the results from the graph variant calling pipeline with the Platinum Genomes benchmark data for samples NA12877 and NA12878[21] using vcfeval, which corrects for VCF representation ambiguity by comparing at the haplotype level [10] [115]. To provide additional controls, FreeBayes [23], Platypus [91] and Samtools [57] were run on BWA-MEM [53] alignments of the same input data to GRCh38 with their default options in order to produce positive control variant calls. Figure 4.4 (A) shows the precision and recall of each method aggregated across both samples and all regions. Figure 4.4 (C) and (D) show precision-recall curves for SNPs and indels, respectively. In comparison to the primary graph (the graph containing only the existing

Figure 4.3: Variant Calling with Genome Graphs. (A) Read pileup on a base graph whose reference path is highlighted in green. Only variant or non-reference base values are shown in the reads. (B) The augmented graph contains the base graph as well as new structures implied by the pileup. This graph contains three top-level ultrabubbles, each forming a site. (C) Variant calls for each site. The first two (a heterozygous SNP and a homozygous insertion) are considered reference calls because they were present in the base graph, whereas the third (a heterozygous combination of a SNP and a deletion) is non-reference because it was novel to the augmented graph.


(A)
Read Pileup on Base Graph
G
CC
T
G
CC
G
CC
G
T
CC
G
AA
T
T
AGAAATTAGG
G
CC
(B)
Augmented Graph and Sites
AA
T
T
AGA
A
A
TT
AGG
G
CC
(C)
Calls
Ref. Het. SNP:
T -> T/G
Ref. Hom. Insertion:
T -> TCC
Non-ref. Het SNP/Deletion:
AAATT->A/AAGTT

reference sequence, and therefore a control for the same variant calling algorithm applied to just the knowledge in the existing reference), the 1KG and Cactus graphs' F1-scores (Supplementary Table 1) increased by 3.50% and 1.98%, respectively, increasing for both single nucleotide variants (3.13%, 1.95% respectively) and indels (6.02%, 4.40% respectively). Furthermore, 1KG graphs have the overall highest accuracy (F1 score) of all methods, although Samtools and Platypus perform best overall for SNPs and indels, respectively. Supplementary Section 7 contains additional breakdowns of the F1-scores by region (Supplementary Figs. 7-8), sample (Supplementary Fig. 9), and type (Supplementary Fig. 10), as well as scores without clipping to confident regions (Supplementary Fig. 11). Generally (in 13 out of 18 cases), the 1KG graph has higher accuracy than both the primary and scrambled controls.

We define a *reference call* as a call asserting the presence of a position or edge in the base graph. The experimental graphs can dramatically reduce the number of non-reference calls, as compared to control. For example, the Cactus and 1KG graphs reduce non-reference calls by more than a factor of ten (Fig. 4.5 (A)) relative to the Primary reference graph. Furthermore, the precision of these reference calls is higher than the non-reference calls for the non-scrambled graphs (Fig. 4.5 (B)).

Larger structural variants can be called using the same logic as point mutations, provided they are already in the graph; Figure 4.5 (C) displays the indel length distribution for the two top-performing graphs and the primary control, as well as a breakdown of indel lengths for reference and non-reference calls. The reference call indel lengths in the experimental graphs are larger than the Primary and non-reference lengths and, in the case of Cactus, contain indels exceeding the read length. This adds up to a large number of additional called bases: for example, across the regions the Cactus graphs call 94 indel events larger than 50 base pairs totaling 10045 bases, none of which are found using the Primary graph with the same algorithm.

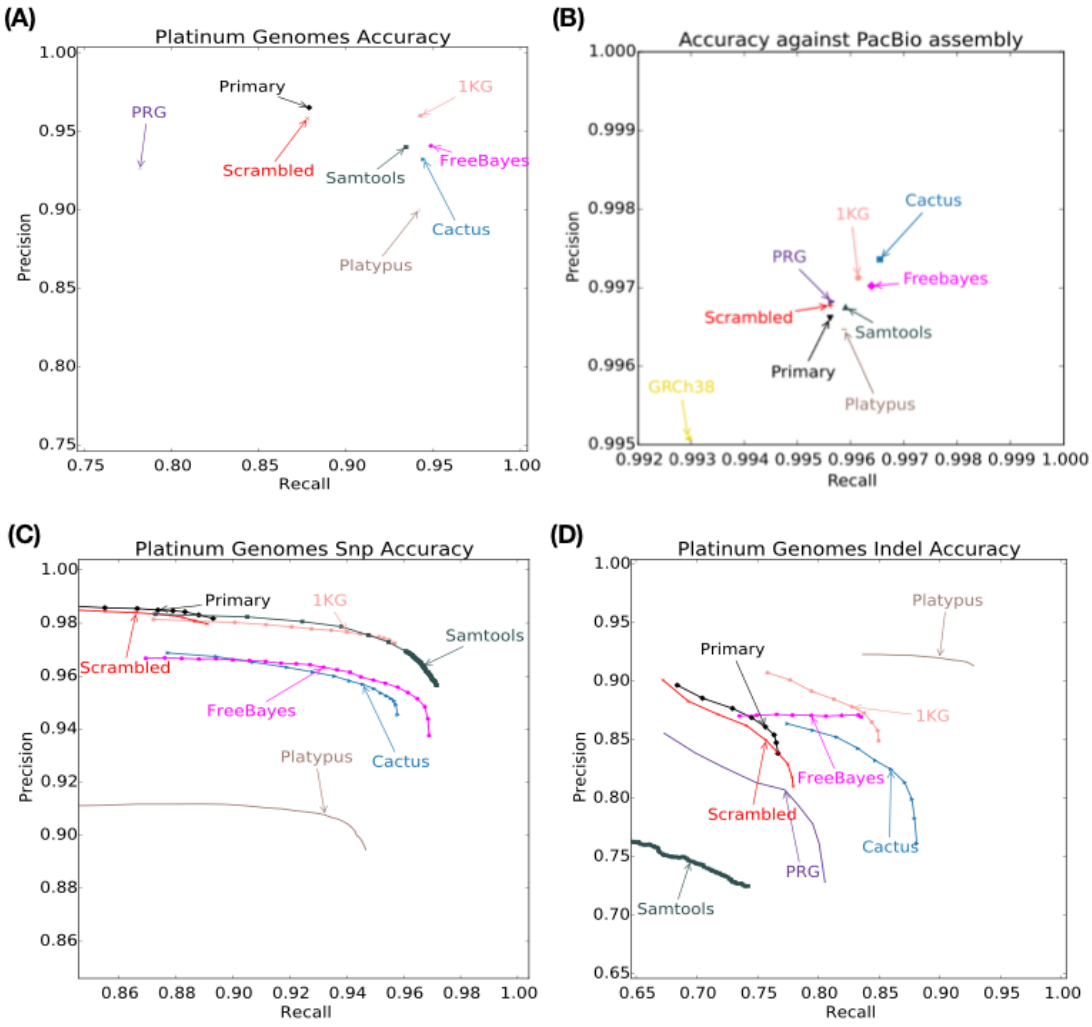
To mitigate potential biases with the Platinum Genomes benchmark data as

a truth set[21], we conducted what we term a "reference-free" evaluation of vg's variant calling accuracy, by comparing against de novo assemblies instead of assumed-true variant calls. In brief, short reads pooled from two haploid assemblies were used to call variants on each sequence graph. The accuracy of this reconstruction was evaluated using PacBio-based *de novo* assembly fragments, which by definition are free of reference artifacts and are derived from a different sequencing technology (see Online Methods, Supplementary Section 8 and Supplementary Figure 12). The results can be seen in Figure 4.4 (B) and Supplementary Figure 13. Several experimental graphs have greater precision and recall than the Scrambled and Primary controls; combined across all regions except SMA (which was insufficiently covered by PacBio assemblies to be usefully analyzed), vg on the Cactus graph outperformed existing methods. The results appear to agree closely with those from the VCF-comparison-based evaluation, considering that the two techniques use different sources of truth and different evaluation metrics.

4.3.3 Short Path Accuracy

We sought to understand how complete and accurate the sequence graphs studied were in their representation of common variants. To approximate this, we measured the fraction of lightly error-pruned K-mer instances (here K=20, see Online Methods) in a large subset of 1KG sequencing reads that were present within each graph, calling this metric *K-mer recall* (see Online Methods). We observe (Fig. 4.6, Supplementary Section 9, and Supplementary Fig. 14) that graphs built from the largest libraries of variation contain the great majority of such K-mer instances. For example the 1KG, PRG and Cactus graphs contain an average across regions of 99% of K-mer instances, while the primary graph contains an average across regions of 97%. Conversely, we asked what fraction of 20-mer instances present in a graph were not present in any 1KG read, calling this metric *K-mer precision*. Strikingly, we find that precision is greatly reduced in some graphs relative to control. For example around 15% (averaged across

Figure 4.4: Variant Calling Evaluation. (A) Precision (portion of called variation in agreement with the truth set) and recall (portion of variation in the truth set in agreement with what was called) against the Platinum Genomes truth VCFs aggregated across NA12877 and NA12878 for all regions, as measured by vcfeval. (B) Per-base precision and recall as measured by the reference-free evaluation in BRCA1, BRCA2, LRC_KIR, and MHC. The GRCh38 point shows a comparison of the existing primary reference haplotype sequence against the de novo assembly. (C) - (D) Breakdown of precision and recall from (A) into SNPs and indels, respectively. Curves are shown by including accuracies at quality thresholds that fall within a radius of 0.1 around the maximum F1. Full results featuring F1-scores for all graphs are in Supplementary Section 7.


(A)
Platinum Genomes Accuracy
Precision
Recall
Primary
1KG
PRG
Scrambled
Samtools
FreeBayes
Cactus
Platypus
(B)
Accuracy against PacBio assembly
Precision
Recall
1KG
Cactus
PRG
Freebayes
Scrambled
Samtools
Primary
GRCh38
Platypus
(C)
Platinum Genomes Snp Accuracy
Precision
Recall
Primary
1KG
Samtools
Scrambled
FreeBayes
Platypus
Cactus
(D)
Platinum Genomes Indel Accuracy
Precision
Recall
Platypus
Primary
1KG
FreeBayes
Scrambled
Cactus
PRG
Samtools

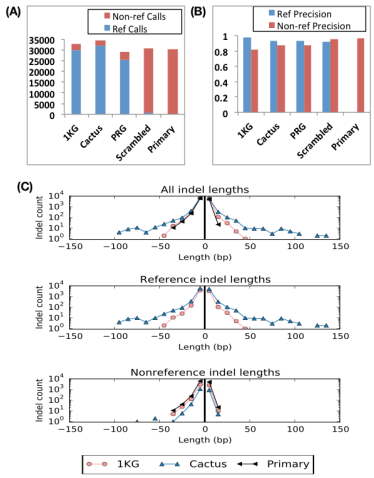


Figure 4.5: Reference versus Non-reference Calls. (A) Total number of reference and non-reference calls across all samples and regions. (B) Precision of reference and non-reference calls. (C) Indel lengths of reference and non-reference calls, where insertions and deletions are represented by positive and negative lengths, respectively. In all cases we ignore calls of GRCh38 reference alleles, as these numbers are reported from GRCh38-based output VCFs.

regions) of 20-mers enumerated from 1KG graphs do not appear in any 1000 Genomes low coverage read. We hypothesize that this is because the density of variation is sufficient in such graphs to admit many paths implying recombinations that are either absent or very rare in the population. To attempt to raise precision, for the 1KG data we constructed graphs using haplotype information to eliminate a substantial subset of unobserved paths, creating the "1KG Haplo 50" graph (Supplementary Section 10). This resulted in increased precision (by about 10 percentage points in BRCA2) with only a small reduction in recall, as shown in Figure 4.6 and Supplementary Figure 13. However, this comes at the cost of a performance degradation in read mapping (Fig. 4.2) and variant calling (Supplementary Section 7). One possible explanation for the performance reduction is that the necessary duplication ("unmerging") of paths in this procedure reduced the aligner's ability to unambiguously map reads.

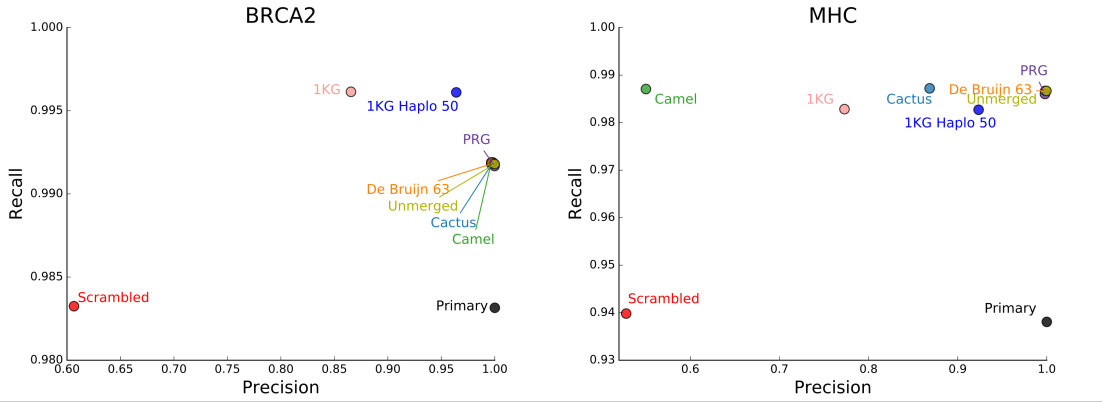


Figure 4.6: Short path completeness and accuracy. Assayed by comparing 20-mer instances.

4.3.4 Graph Character

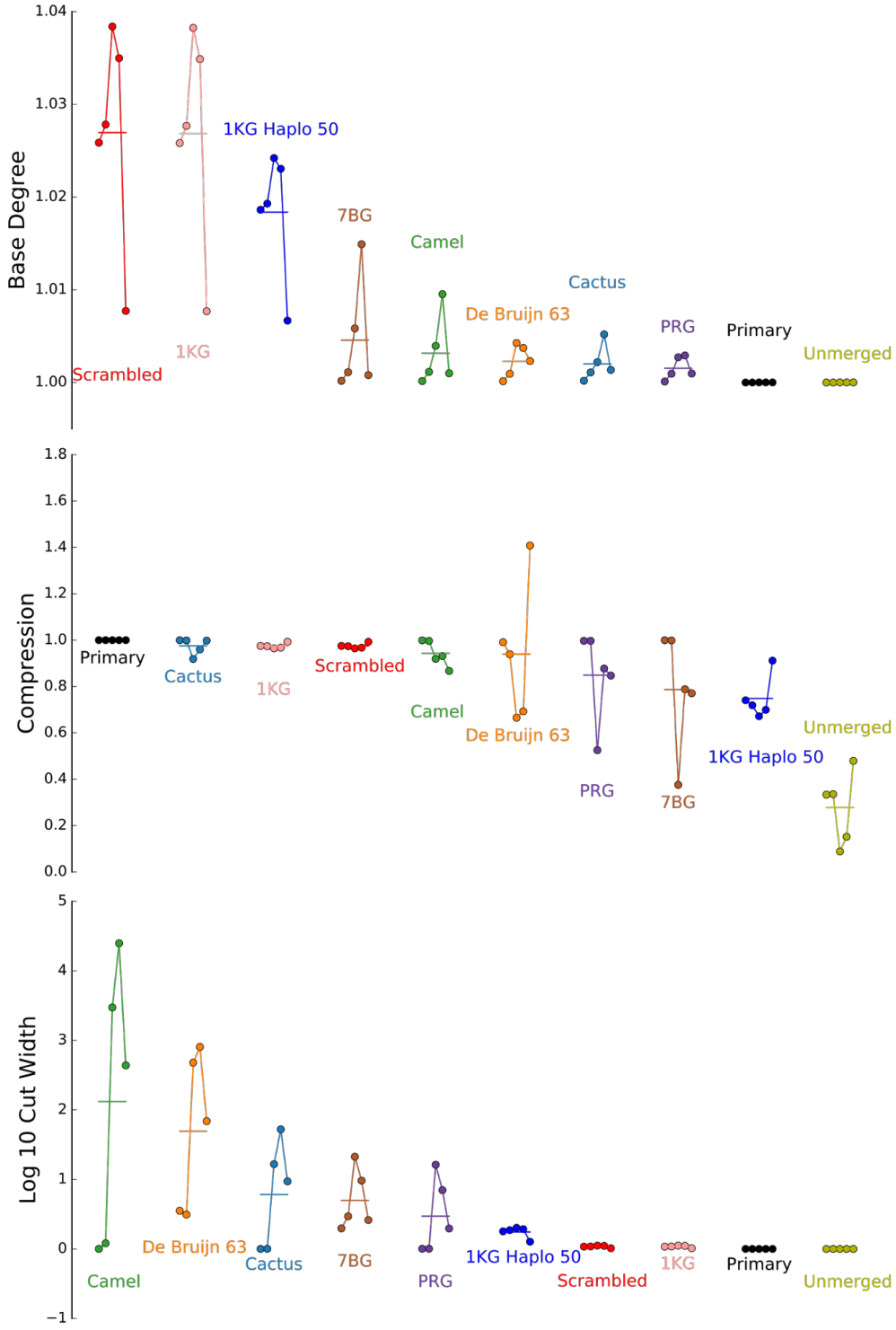
We found that even within each region the different submitted graphs varied substantially in their performance on our evaluations of read mapping and variant calling. They varied even more so with respect to basic graph properties (Supplementary Section 11, Supplementary Tables 2-9). To quantify this variability we defined normalized graph metrics for basic graph properties. *Graph compression* is the length of the primary reference sequence within the region divided by the sum of the lengths of the nodes in the graph. It is a normalized measure of the number of positions in the graph. The *(base) degree* is the average per-side degree of the graph in a bidirected graph representation with single-base nodes, and is a measure of how much branching a graph contains. The *cut width* (Supplementary Table 10) is a measure of apparent sequence rearrangement. Briefly, within a topologically sorted graph, where all positions are ordered, cut width is the average over all gaps between successive positions of the number of edges connecting positions on the left side of the gap to positions on the right side of the gap (Supplementary Section 12)[33]. We see wide variation in these measures across the graphs (Fig. 4.7). Furthermore, across the different regions we find that there is an inverse correlation (R=-0.674, p=0.00230) between cut width and variant calling accuracy and

a positive correlation (R=0.244, p=0.0268) between compression and variant calling accuracy (Supplementary Fig. 15). The base degree does not significantly correlate with variant calling accuracy. These correlations suggest that uncompressed and highly rearranged graphs do not work effectively with our current read mapping and variant calling process.

4.4 Discussion

Contemporary non-graphical variant calling procedures use different algorithms for each class of variants: substitutions, small indels, larger indels, balanced rearrangements, and so on. We have demonstrated that variant calling on a sequence graph mostly obviates this complexity, because being able to call the presence or absence of elements within a sample graph is potentially sufficient for calling known structural and point variation equally well. The simple, nascent variant calling algorithm we tested produced variant calls that were quite concordant with those from other state-of-the-art variant calling pipelines, while unifying the calling of known SNPs and other known structural variation. That individual tools slightly outperformed the variant calling algorithm presented here in terms of individual variant types, i.e. snps and indels, is unsurprising given the relative maturity and algorithmic sophistication of those tools. Importantly, many of the submitted graphs showed improved variant calling performance over the primary and scrambled graphs. The relative improvements come alongside a large reduction in the number of non-reference calls. Furthermore, reference calls were more accurate than non-reference calls, suggesting that variant calling is indeed more accurate overall when the variants themselves are contained within the graph. These results support the notion that sequence graphs can transform variant calling by reducing it to the simpler problem in which only rare variants, absent from the graph, must be discovered de novo. It is possible to foresee cutting the number of non-reference point variation calls from

Figure 4.7: Empirical graph statistics. In each panel the result for each region is shown by a dot, in the following order: BRCA1, BRCA2, LRC_KIR, MHC, and SMA.


Base Degree
1.04
1.03
1.02
1.01
1.00
Scrambled
1KG
1KG Haplo 50
7BG
Camel
De Bruijn 63
Cactus
PRG
Primary
Unmerged
Compression
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Primary
Cactus
1KG
Scrambled
Camel
De Bruijn 63
PRG
7BG
1KG Haplo 50
Unmerged
Log 10 Cut Width
5
4
3
2
1
0
−1
Camel
De Bruijn 63
Cactus
7BG
PRG
1KG Haplo 50
Scrambled
1KG
Primary
Unmerged

the millions, as in standard genome wide pipelines today, to on the order of thousands (see Supplementary Section 3).

During the course of the variant calling comparison, we developed an appreciation for the shortcomings of relying solely on the Platinum Genomes benchmark data as a truth set[21]. A key concern is that the Platinum Genomes calls were derived by means of a consensus of contemporary methods, all of which use the existing linear reference and BWA-MEM-based mappings. Additionally, compared to vg, the Platinum Genomes dataset often uses different combinations of calls to "spell" the same haplotype. Moreover, it often omits calls necessary to spell a haplotype because it is not confident in them. While the omitted calls are in regions marked as low confidence, a variant normalizer cannot normalize a call that is not there. To get around these problems and potential biases we introduced a reference-free method for assessing variation calls. This evaluation demonstrated good consistency with the Platinum Genomes in terms of the relative ranking of the different methods evaluated, and demonstrated clearly that the best graph methods slightly outperform existing methods.

Supporting the observed improvements in variant calling, we demonstrate that read mapping can be made both more comprehensive and less ambiguous with sequence graphs. Increases in perfect mapping and reductions in substitution and indel rates were broadly consistent with the effect we would expect if the graphs were representing the majority of common polymorphisms, leaving the residual read error rates to account for the majority of alignment differences. In this sense read mappings were demonstrated to be less locally ambiguous, with mismatches and edits having a more clearly defined meaning. Furthermore, the fact that read mappings were also less globally ambiguous (i.e. more certain in their overall placement within the genome) is perhaps surprising. We thought at the outset that using detailed graphs would have the drawback of increasing the number of times a read maps to two or more places by increasing the sheer number of mapping possibilities. However, we found that the opposite is true -

the addition of known polymorphisms to the graph allows reads to better distinguish their true mapping location from secondary, paralogous locations. Scaled genome-wide, these improvements could help canonicalize mapping to the vast majority of variation, which will become especially important as genome variants are increasingly used in the clinic. The increases in perfect mapping could also allow alignment to be made more efficient by allowing larger, more stringent seeds or more aggressive ungapped matching. Our early work with vg indicates that there is ample opportunity for improvement and investigation of these novel approaches to the design of high-performance mapping algorithms. We also collected some preliminary data that suggests that the gains in mapping obtained by moving from the existing reference to a graph like the 1KG graph are super-population specific, suggesting that sequence graphs have the potential to reduce the local ethnic bias inherent in a single reference genome.

By taking a community approach, we were able to sample a wide variety of the contemporary software for building sequence graphs. It is apparent that different methods produce dramatically different graphs, as measured both by direct graph analysis and by practical performance as a reference for common genomics tasks, suggesting that the field is just in its formative stages. In trying to understand how "complete" and "accurate" graphs built with today's methods are at representing short sequences present in the population, we encountered several surprises. In particular, we found a large number of short non-biological paths created within the highest degree graphs, such as the de Bruijn graphs, parts of the 1KG graphs, and certain of the Seven Bridges graphs. We tried modifying the 1KG graphs to reduce the number of false recombination possibilities without much success. We may in the future find that we can tolerate these short non-biological paths, or that another approach is needed to eliminate them.

One alternative approach is to create uncompressed, lower-degree graphs by duplicating variable regions to directly represent haplotypes, but it is likely that, as demonstrated by the 1KG Haplo 50 and (at the logical extreme) Unmerged graphs,

the resulting long, equivalent sequence paths would create too much multi-mapping ambiguity. Perhaps a better solution may be the use of haplotype information embedded within the sequence graph[78], making it a *variation graph*. This would allow algorithms to map to a common graph coordinate system while accounting for variants, read errors, and recombinations within the mapping process itself. This approach would eliminate the need for several inelegant heuristics used in contemporary linear-reference-based analysis pipelines [2, 68].

Sequence graphs can now be built from libraries of common variants, and tools like vg, though still experimental, illustrate the huge potential of the graph-based approach. There are a number of questions yet to be tackled. How should duplications and repeats be represented? How can one best map to a graph? How should short variants whose homologies are unclear be parsed? How can graphs be used to enable a more comprehensive taxonomy of variation? These questions all represent open avenues for future research.

4.5 Online Methods

4.5.1 Source Data

Participants were provided with a dataset consisting of five genomic regions (BRCA1, BRCA2, LRC_KIR, SMA, and MHC) to use in the creation of their graphs. The dataset came in the form of a "reference" sequence and one or more "alternate" sequences for each region. For the LRC_KIR, SMA, and MHC regions, those alternate sequences were the alt loci present in GRCh38.p2 for the regions of the same names in the assembly, with the reference being the portion of the corresponding chromosome encompassing the chromosomal coordinates for all of the alts. The reference regions for BRCA1 (ID 672) and BRCA2 (ID 675) were downloaded from Entrez Direct, while alternate sequences were the annotated genes from the CHM1 hydatidiform mole assembly, and the LRG

sequences for those genes [7, 43, 62]. Some participants used additional source data in constructing their graphs.

4.5.2 Graph Format

All graphs were generated in or converted into an SQL text format for submission. The graphs were then loaded into databases compatible with the GA4GH Graph Reference Server, and servers for the graphs were hosted on a Microsoft Azure cloud instance. Individual evaluation tools hit against these API endpoints. For read alignment and variant calling purposes, graphs were downloaded from the servers using the sg2vg converter tool, written for this project, and stored in .vg graph format. This on-disk format could be efficiently indexed for read alignment—a function that the GA4GH server did not support—and so was preferred for evaluations dependent on read alignment. The graphs themselves were created using a variety of methodologies and approaches, detailed in Supplementary Section 10.

4.5.3 Alignment Target Quality

The submitted graphs were used to align reads from 2,691 low-coverage samples from the 1000 Genomes project, which had been aligned to GRCh38 with BWA-MEM [53]. Alignments to the primary reference and, where available, the GRCh38 alt loci for a region were downloaded using Samtools [57]. The process took advantage of the tool's ability to subset indexed files over FTP in order to obtain just reads mapped within the region [57]. Next, the alignments were converted into reads, yielding the relevant reads for that sample and region. Unpaired reads in the downloaded set were discarded. An attempt was made to correct for a known data corruption bug in the version of BWA-MEM used to produce the alignments, by taking the sequences given for alignments to the primary reference over the sequences given for the same read aligned to an alt, where available (Heng Li, personal communication). Input graphs were downloaded from the

reference servers using the sg2vg program. They were then broken into nodes of no more than 100 bases each and re-numbered according to a heuristic extension of topological sort to cyclic graphs. Graphs were indexed and alignment was performed with the vg program, using a K-mer size of 16 and an edge-crossing limit of 3 for the GCSA2 index. The portion of reads mapping uniquely was calculated. To qualify as uniquely mapped, a read had to have a primary mapping with 0.95 matches per alignment column or fewer. Additionally, qualifying reads had to have either no secondary mapping or a secondary mapping having fewer than 0.85 matches per column. The denominator for the portion mapping uniquely was the number of reads having either a secondary mapping distinct from the read's primary mapping or no secondary mapping at all (see Supplementary Section 3). The portion of reads mapping perfectly was defined as the portion having 1 match per alignment column. The substitution rate was defined as the portion of bases in length-conserving replacements out of all substituted or matched bases. Bases matched or substituted against N characters in the reference graph were ignored. The indel rate was defined as indel count divided by substituted and matched bases. Bases matched or substituted against reference Ns were ignored, as were indels that constituted softclips.

4.5.4 Platinum Genomes Variant Calling Evaluation

A graph variant calling pipeline based on the Samtools pileup method was implemented in vg and run independently on three 50x coverage samples from Platinum Genomes (NA12877-9). First, the reads were mapped to each graph as described above. The alignments were then filtered to remove secondary mappings, as well as mappings with mapping quality score less than 15, mappings that had been promoted to primary over another properly paired mapping of greater single-end score, and mappings with soft-clipped or ambiguous ends (more details in Supplementary Section 7). A pileup of aligned read bases was then constructed for each position and edge in the graph

ignoring bases with read quality score less than 10. The SNPs, insertions, and deletions implied by the two most supported non-reference entries in each pileup were then added into the graph to create an "augmented" graph. Sites in the augmented graph were computed using the ultrabubbles algorithm[85]. For each site, the two non-reference paths with the most read support were greedily chosen using a breadth-first search. A path's read support was defined here as the minimum pileup support of any node or edge it contains; each node's support was calculated as the average support across the node's bases. Finally, given the reference path and these two alternate paths for each site, a genotype was computed using a thresholding heuristic based on the ratios of the paths' pileup supports. Alternate alleles were called as heterozygous if they had at least three times as much read support as the reference (or six times for a homozygous alt call). The genotypes were written directly to VCF. The variants were normalized by using vt[106] to flatten multibase alts that contain reference calls. Calls for both NA12877 and NA12878 were compared against their respective Platinum Genomes truth set VCFs; these were the only samples with truth VCFs available. Precision and recall against the truth set were assessed with vcfeval[10]. True and false positives and negatives returned from this tool were classified as SNPs and indels using bcftools, and clipped into the Platinum Genomes confident regions. Precision, recall and F1-scores were then computed for each possible quality threshold in the VCF. For the vg call results, minimum read support (AD field in VCF genotype) across called alleles was used as a proxy for quality. Aggregate results across samples and regions were computed by pooling the vcfeval results together. The precision-recall curves (Fig. 4.4 (C) and (D)) were drawn by filtering the VCF files by all values of variant quality and displaying only those within distance 0.1 of the maximum F1-score. The points shown in Figure 4.4 (A) were chosen to correspond to the quality threshold yielding the maximum F1-score.

4.5.5 Reference-Free Evaluation

A "synthetic diploid" genome was conceptualized by combining data from two haploid samples, CHM1 and CHM13[103]. For each sample, GRCh38-aligned low-coverage Illumina reads and relatively complete PacBio-derived assemblies were obtained. The CHM1 and CHM13 reads were obtained by combining both runs from NCBI SRA SRX1391727 and SRX1082031, respectively, and mapping to GRCh38 using BWA-MEM[53]. The CHM1 assembly used was GenBank accession number GCA_001297185.1, while the CHM13 assembly was GCA_000983455.2. For each region, a pooled collection of the relevant Illumina reads across both CHM1 and CHM13 was created. Next, the reads were subsampled for balanced coverage between the two haploid genomes as would be expected in a real diploid sample. For each submitted graph under tests, the reads were aligned using vg, and the vg variant caller was used to produce variant calls. The resulting VCF for each graph construction method and region combination was turned into a new "sample graph" to which the relevant portions of the PacBio assemblies were aligned. Treating the aligned assembly fragments as the truth, the precision and recall of each sample graph were measured as a function of which original submitted graph it was derived from.

Assembly fragments used for evaluation were selected by alignment of the primary reference sequences for the regions against the CHM1 and CHM13 assemblies using BLAT version 36x2 [45]. Aligned regions in the assembly covering more than either 50% of an assembly contig or 50% of a region, with more than 98% identity, were extracted from the assembly and used for realignment. The SMA region was excluded from the evaluation due to patchy, overlapping coverage of the region in the two assemblies. Additionally, the first 87,796 bases of the LRC_KIR region were excluded from the sample graphs and the aligned truth set contigs due to an apparent lack of representation in the CHM13 assembly.

4.5.6 Assessing Graph Completeness

Reads aligning to the test regions were obtained from 2,691 low-coverage samples in the 1000 Genomes Project, and each sample's reads were used to generate a collection of K-mers (K=20) using Jellyfish [3, 65]. These were compared against the collection of K-mers in each graph as enumerated by vg with an edge-crossing limit of 7. In order to account for K-mer frequencies, duplicate K-mers were *not* ignored. K-mers containing N characters were ignored in both collections, and K-mers only observed once in their sample were ignored in the 1000 Genomes-derived K-mer collection. This latter filter was intended to remove the large majority of erroneous K-mers: we expect errors to be Poisson-distributed one-off events while real variants are likely to recur within a sample. Recall, defined as the portion of all read-derived K-mers present among the graph-derived K-mers, and precision, defined as the converse, were computed for each graph.

4.5.7 URLs

VG, https://github.com/vgteam/vg.

Patches to VG, https://github.com/adamnovak/vg/tree/graph-bakeoff.

GA4GH to VG Importer, https://github.com/glennhickey/sg2vg.

VG to GA4GH Exporter, https://github.com/glennhickey/vg2sg.

GA4GH Graph Schemas, https://github.com/ga4gh/schemas/tree/refVar-graph-unary.

GA4GH Graph Server, https://github.com/ga4gh/server/tree/graph.

Graph evaluation software, https://github.com/BD2KGenomics/hgvm-graph-bakeoff-evaluations.

FASTG, http://fastg.sourceforge.net/.

Illumina Platinum Genomes, http://www.illumina.com/platinumgenomes/.

Jellyfish, http://www.cbcb.umd.edu/software/jellyfish/.

Platypus, http://www.well.ox.ac.uk/platypus.

FreeBayes, https://github.com/ekg/freebayes.

Samtools, http://www.htslib.org/.

VCFeval, https://github.com/RealTimeGenomics/rtg-tools.

4.5.8 Software Versions and Commit Hashes

VG, 158d542497445b532b0e9e40223f5023ee6b52dd.

GA4GH to VG Importer, 468026ad70f0425af1959b287ffcaac91b8a9deb.

VG to GA4GH Exporter, 4efde8e64a8bd113a0e83685628bbaf0cbc2be3f.

GA4GH Graph Schemas, ea58ac46dad84be67c500e517ff2fb05a43a187a.

GA4GH Graph Server, c6daebca4c69a4ff4d9d56cfdf587556f2ce1116.

Graph evaluation software, 52b0537713629471f6ea97ccf552d6727c630f3d.

FreeBayes, 9e983667d47f6b5dcbb90070da8de69714038f46.

Samtools, version 1.3.1.

4.5.9 Acknowledgments

This work would not have been possible without the generous support of the National Human Genome Research Institute (1U54HG007990 [BD2K] to B.P. and D.H., 5U41HG007234 [GENCODE] to B.P.); the W. M. Keck Foundation (DT06172015 to

B.P. and D.H.); the Wellcome Trust (100956/Z/13/Z to G.M.); the Simons Foundation (SFLIFE# 351901 to B.P. and D.H.); the ARCS Foundation (2014-15 ARCS fellowship to A.M.N.) and Edward Schulak (Edward Schulak Fellowship in Genomics to A.M.N.).

4.5.10 Author Contributions

A.M.N. contributed the Camel graphs, wrote read mapping and variant calling code for vg, ran the read mapping and reference-free evaluations, and contributed extensively to the organization of the manuscript. G.H. contributed the Cactus, 1KG, 1KG Haplo 50, Primary, and Scrambled graphs, and performed the variant calling evaluation. E.G. contributed the bulk of the vg tool. S.B. contributed the analysis of graph statistics. A.C., S.G., N.O., and A.W.Z. contributed the Curoverse graphs, with A.W.Z. supervising. A.D., J.K., supervised by G.MV., contributed the PRG graphs. J.E. contributed alignment code to the vg tool. M.A.S.E. contributed advice and corrections to the manuscript. A.K. contributed the De Bruijn 63 graphs. S.K. contributed code to the GA4GH graph server and contributed extensive organizational support. D.K. and G.R. contributed the SBG graphs. H.L. contributed experimental design support for the read mapping evaluation, and advice on the manuscript. M.L. worked on scaling up the variant calling pipeline to whole genomes. K.M. contributed a set of graphs for the chromosome X centromere. M.S-O. contributed code to the GA4GH graph server and managed the graph data import pipeline for this work. R.D. contributed to the design of the GA4GH graph server interface and schemas, and supervised other authors. G.M., D.H., R.D. and B.P. contributed to the design of the project and supervised authors. B.P. and organized the project directly, and wrote extensive portions of the manuscript. All authors edited the manuscript.

Chapter 5

Towards a Human Genome Variation Map

5.1 Introduction

In Chapter 4, it was demonstrated that the use of graph-based genomic references can result in improved variant calling performance over traditional linear references. However, in that study, the graph references that produced the most accurate variant calls compared to truth VCFs were derived from the 1000 Genomes Project's main variant call files [3], and allowed the detection of only relatively short indels, under about 50 bp (Fig. 4.5 (C)). Compared to the approximate 300 bp length of a single Alu repeat insertion [113], this is inadequate.

In addition to comparison against traditional Illumina-based variant calls, variant call accuracy was also evaluated using PacBio-based assembly data, by measuring how well the sample graph produced from the variant calls for a pooled synthetic diploid sample agreed with separate haploid assemblies. By this metric, variant calling performed with Cactus-based graphs was found to be more accurate than variant calling performed with 1000 Genomes-derived graphs (Fig. 4.3 (B)). Cactus-based graphs were

also shown to allow the detection of longer insertions and deletions than 1000 Genomes-based graphs can find (Fig. 4.5 (C)). Overall, Cactus-based graphs, which are produced from the alignment of long alternate loci sequences, have some important advantages that 1000 Genomes-based graphs lack.

In order to construct a versatile graph-based reference that will serve as a community resource for read mapping and variant calling, it is desirable to combine the best aspects of these two types of graph. Additionally, as the bake-off project of Chapter 4 worked only on regions up to a few megabases in size, it is desirable to demonstrate the effectiveness of graph-based methods at larger scales, where qualities like the ability to resolve mappings between ambiguous regions, and the ability to effectively use paired-end information, are more critical.

In this chapter, we present a method to create graph references combining the best qualities of 1000 Genomes-based and Cactus-based graphs, and validate these graphs on the scale of a chromosome.

5.2 Methods

5.2.1 Graph Construction

In order to combine a Cactus-based graph with a 1000 Genomes-based graph, we implemented a new subcommand in the `vg` variation graph toolkit [24]. The new tool, `vg add`, augments an existing graph by inserting variants from a VCF file. It works by extracting local haplotypes around each variant that are consistent with the phased samples in the VCF file, and then aligning them to the relevant region of the graph, as determined by tracing an embedded primary reference path in the graph. For particularly large insertions and deletions, where a complete local alignment would be impractical, the ends of the variant are aligned, and the resulting alignments are stitched together to describe the actual variant.

The `vg add` tool, along with a Toil-based orchestration script [112], were used to combine variation information from three sources. The base level graph was obtained using Cactus [82], by aligning together the chromosome 22 primary sequence and the chromosome 22 alt loci and so-called **random sequences** (which are localized to a chromosome but not placed along it) from GRCh38. The alignment was performed such that the main chromosome 22 sequence and the random sequences were not aligned to themselves or each other; only the alt sequences were allowed to align to the other sequences. The Cactus alignment was converted to a `vg` graph with the `hal2vg` tool[1], and the resulting graph had its nodes chopped to a maximum length of 1000 bp using `vg mod`.

On top of this graph, `vg add` was used to add in variants from the 1000 Genomes Phase 3 GRCh38 lifted-over VCF files[2]. Notably, these variant files as distributed by the 1000 Genomes Project are not valid; variants lifted over to the reverse strand of GRCh38 are marked as marked with a `MATCHED_REV` tag in the `INFO` field but left in their GRCh37 orientations, and needed to be reverse-complemented using a script so that the `REF` field contents will match the actual reference sequence at the variant's location.

The `vg add` tool was also used to add structural variants to the graph. Since the structural variants in GRCh38 coordinates as originally obtained[3] were described using a complex combination of `INFO` tags, additional tables, and references to difficult-to-locate external sequence database records, the files had to be preprocessed in order for `vg add` to be able to parse them. All variant alt information was moved into a fully realized concrete sequence in each variant record's `ALT` column, and the reference sequences, even for very long deletions, were placed in each variant record's `REF` column.

[1] `https://github.com/ComparativeGenomicsToolkit/hal2vg`

[2] `ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20130502/supporting/GRCh38_positions/`

[3] `ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/integrated_sv_map/supporting/GRCh38_positions/`

All symbolic allele references and target site duplication sequences were resolved. For mobile element insertions, the original VCF specified the presence, but not the exact length, of a poly-A tail; in these cases, several duplicate variant records were created, with poly-A tail lengths of 10, 25, and 50 bases. This approach was selected in hopes of providing a mapping target sufficient to collect reads showing the correct poly-A tail length, which could then potentially be determined through graph-based variant calling.

Once the graph was fully constructed, it was indexed using `vg`. This produced the XG and GCSA2 indexes that were required to align reads to the graph. The graph was then subjected to two evaluations, based on aligning reads to and variant calling against it. The structural variant evaluation, described in Subsection 5.2.4, involved comparing variant calls against a truth set, while the assembly realignment evaluation, described in Subsection 5.2.3, involved evaluating the sample graph derived from the variant calls with reference to a pair of haploid assemblies.

5.2.2 Variant Calling Techniques

Both of the evaluations presented here relied on some improvements to the `vg` variant caller, `vg call`, made for the purpose of this study. Previously, the caller operated only on sites defined by top-level ultrabubbles [85], and exclusively produced VCF output. However, when working with the very large top-level ultrabubbles induced by the inclusion of large structural variants in the graph, it became necessary to consider nesting of ultrabubbles. The core of the caller was rewritten to handle ultrabubbles recursively. In this method, each ultrabubble is treated as a site. Calls are made on each top-level site while abstracting away variation inside of nested child sites, and then each nested child site is recursed on and a call is made for it in accordance with the copy number assigned to it by the higher-level call. An attempt was made to patch the lower-level call results into the output for the higher-level containing sites, subject to the limitation that within alternate alleles each child was always represented by its

highest-coverage traversal, even if it was given a heterozygous call. This limitation was added to avoid difficult situations related to phasing between adjacent heterozygous children.

Additionally, in order to allow ultrabubbles that were not traversed by a primary reference path to be usefully called, the caller was modified to be able to output its calls in `Locus` format, in addition to VCF. This format is a binary, Protobuf-based format, similar to the other formats used by `vg`, and represents calls as `Locus` objects. A `Locus` can have a series of graph paths stored within it as alternative alleles, along with zero or more genotypes called as potentially-empty collections of the available alleles at the `Locus`. This format allowed for each top-level site to be represented, and also allowed for child sites to be represented. Additionally, it allowed gVCF-like assertions to be made about the existence of a primary reference path outside of variable sites, by creating `Locus` objects covering non-variable material.

To improve performance on calling large structural variants, and to account for the transition to a recursive, child-abstracting architecture, numerous changes to the internal heuristics used by the variant caller were made. These changes were made manually, working primarily on the NA12878 sample; no automatic optimization of caller parameters was performed. All variant calls created in this study were performed using the default `vg call` heuristic parameters and an additional `--max-dp-multiple 2.5` setting.

Since sample graphs were required for the assembly realignment evaluation, the `vg mod` tool that produces sample graphs was enhanced to allow it to read and process calls in `Locus` format. Sample graphs were produced for `Locus`-format calls by eliminating all nodes and edges not called as present in some `Locus`.

5.2.3 Assembly Realignment Evaluation

The first evaluation was a variant of the assembly realignment evaluation from Chapter 4. A synthetic diploid sample was created from the CHM1 and CHM13 hydatidiform mole samples aligned to GRCh38 (it was actually the same sample used in Chapter 4). From this sample, read pairs where either member mapped to chromosome 22 or any of its random or alt scaffolds were collected.

These reads were aligned to the graph under test and used for variant calling against it. However, instead of calling variants to VCF, we used our improved version of `vg call` to produce `Locus`-format variant calls for the ultrabubbles in the augmented graph (including both parent and nested child ultrabubbles) [85], and assertions of the presence of all primary reference path edges not involved in ultrabubbles. Finally, as in Chapter 4, the augmented graph was subsetted to create a sample graph. Each graph under test was also evaluated as if it were a sample graph, without the variant calling and subsetting steps, in order to provide a control.

We evaluated two graphs in this way: the "HGVM" graph, created using Cactus and `vg add`, as described above, and a "Control" graph, consisting of just the chromosome 22 GRCh38 scaffold and associated random scaffolds, with no variants added. Each of these gave rise to one actual sample graph, produced by the variant caller, and one control sample graph, produced by passing through the entire graph under test as if it were the sample graph.

To evaluate each sample graph, the scaffolds from the CHM1 and CHM13 assemblies relevant to chromosome 22 were determined using a script. This script aligned 10 kb chunks of each scaffold every 100,000 kb to the 24 primary GRCh38 chromosome scaffolds, until a hit scoring 95% of the maximum possible score was obtained. All scaffolds for which that first sufficiently good hit was to chromosome 22 were taken and chopped into pieces every 1000 bp to produce a set of assembly fragments. (The

exception was scaffold `LBHZ02000095.1` from CHM13, which was manually found to consist of sequence mapping primarily to chromosome 13, and consequently excluded.) Overall, the analysis used 37,931,872 bp of sequence from CHM1 and 36,306,973 bp of sequence from CHM13. The assembly fragments were realigned against each indexed sample graph, and the quality of each sample graph as a representation of the assembly fragments was then measured. This was accomplished by going over the alignments and tabulating the total number of inserted, deleted, substituted, and softclipped bases, and dividing that total by the size of the Control graph (which consisted of chromosome 22 and the associated random sequences), to get a number of affected bases per primary reference base in each category.

5.2.4 Structural Variant Evaluation

The second evaluation, by contrast, was a truth set VCF-based measurement of the accuracy of structural variant calls. Reads aligned to GRCh38 were obtained for five samples: NA12878, NA12889, and NA12890 from the Illumina Platinum Genomes dataset, and HG00513 and HG00732 from the 1000 Genomes High Coverage dataset. From each file of aligned reads, read pairs where either member mapped to chromosome 22 or any of its random or alt scaffolds were collected. These reads were then mapped to the graph under test, and variant calling for each sample was performed. Variant calls in VCF format were obtained. For each sample, the called VCF was compared against the GRCh38 structural variant files that were used for preparing the graph. Recall was computed by considering each unique variant position in the truth VCF for which an alternate allele was called in the sample, and treating it as recalled if the variant calls for the sample in question contained a variant with a length change of 25 bp or more, having a position within 25 bp of the truth position. Filtered variants were ignored in both VCFs. Because the truth set VCF used was not believed or warranted to be complete, precision was computed manually, by randomly sampling a certain number

of calls for variants with length changes of 25 bp or more with calls of alternate alleles, and manually classifying each selected positive call as true or false, by looking at the original input reads and the truth VCF at the variant's location on the UCSC genome browser.

5.2.5 Software and Hardware

The graph construction and evaluation was performed on a Microsoft Azure cluster using five `Standard_G5` worker nodes and one `Standard_A5` master, managed using the Toil workflow engine [112]. The `quay.io/vgteam/vg:v1.5.0-303-gb1a6cc8c-t62-run` Docker container provided the `vg` build used in this study.

5.3 Results

5.3.1 Graph Construction

The cluster run to build the graph, starting from the vg-format Cactus graph, and to run the structural variant and assembly realignment evaluations, took 20 hours, 16 minutes, and 35 seconds on the Microsoft Azure cluster described above in Subsection 5.2.5. No attempt was made to fit the size or resource allocations of the cluster to the requirements of the workflow; the analysis succeeded on the smallest (and only) cluster on which it was tried.

The final chromosome 22 graph contained 3,630,637 nodes and 4,736,765 edges, with a total length (summed over all nodes) of 57,097,953 bp. The initial input data consisted of 51,857,516 bp of primary reference and random scaffold sequence, and 1,625,159 bp of alternate loci (much of which should have aligned to and been merged with the primary reference and random sequences), meaning that at least 3,615,278 bp of material, or 6.97%, was created by `vg add` from VCF files. The input VCF files only had 132,693 bp of structural variant alternate allele sequence and and 1,167,426 bp of

point variant alternate allele sequence. The graph was found to contain 3,750,000 bp of `N` bases not on any of the paths from the base Cactus graph. Extraneous `N` bases were observed occurring in large collections of parallel nodes containing mostly `N` bases. However, as extraneous runs of `N`s do not attract reads that ought to map elsewhere, the graph was still used for further alignment-based analyses.

The graph contained 10 "head" nodes, with nothing attached to their left sides, and 10 "tail" nodes, with nothing attached to their right sides. This was consistent with the 10 primary path sequences (`chr22` and 9 unlocalized chromosome 22 scaffolds) used to construct the base graph.

5.3.2 Assembly Realignment Evaluation

For the first evaluation, based on realignment of the mole reads, results are visible in Figure 5.1. Two graphs were used in the evaluation: the final chromosome 22 HGVM graph, and the Control graph constructed only from chromosome 22 and the associated random sequences in GRCh38, without any alignment or additional variants. Each of these graphs was used as a reference for read alignment and variant calling, and the resulting sample graphs are the "HGVM" and "Control" conditions in the figure. Each of the graphs was also evaluated as if it were a sample graph, producing the "No Call" and "All Ref" conditions, respectively.

On the Deletions, Insertions, and Substitutions metrics (although by a very small margin on the Insertions metric), the HGVM condition performed best. Calling variants based on the graph reference, with its included known variation, resulted in needing to delete fewer bases, insert fewer bases, and substitute fewer bases to explain the assembly fragment truth set, compared to when variant calling was done using the Control graph, which contained no embedded variation. Additionally, the variant calling step itself reduced the number of bases that needed to be deleted by a large factor, and the number of bases that needed to be inserted or substituted slightly, as can be seen

by comparing the HGVM condition to the No Call condition (Fig. 5.1). This suggests that the variant caller is successful in incorporating information from aligned reads into the sample graph. Finally, note that, for deletions, substitutions, and insertions, the decrease in required base modifications attributable to the variant caller operating on the variation-containing graph (i.e. the drop from No Call to HGVM) was greater than the decrease attributable to the variant caller operating on the no-variation graph (i.e. the drop from All Ref to Control). This shows that including variation in the reference can make variant callers more effective.

On the softclips metric, on the other hand, the "No Call" condition outperformed the "HGVM" condition by a small margin, meaning that the graph reference with no variant call performed on it was a better match for the assembly fragments being realigned than was the sample graph produced by variant calling, in terms of the number of softclipped bases required in the assembly fragments' alignments.

5.3.3 Structural Variant Evaluation

For the second, VCF-based evaluation, the precision statistics for the five samples analyzed (HG00513, HG00732, NA12878, NA12887, and NA12890) are visible in Table 5.1, while the recall results for the samples are visible in Table 5.2. Summing across samples, the overall precision was 20 out of 25, or 0.80, while the overall recall was 106 of 151, or 0.702. Together, these produce an F1 score of 0.76.

Figure 5.1: Bases involved in events required to align fragments of the CHM1 and CHM13 haploid assemblies to the sample graph created with the `vg` variant caller for the combined synthetic diploid sample. Quantities are expressed as bases involved in each type of event per base in the control graph. For the All Ref condition (blue), the performance of the primary-reference-only control graph as a sample graph was evaluated. For the Control condition (green), that reference-only graph was used as a reference for variant calling, and the resulting sample graph was evaluated. For the HGVM condition (red), the Human Genome Variation Map graph under test was used as a reference for variant calling, and the resulting sample graph was evaluated. Finally, for the No Call condition (black), the HGVM graph was evaluated directly as a sample graph, with no calling step, to serve as a positive control.

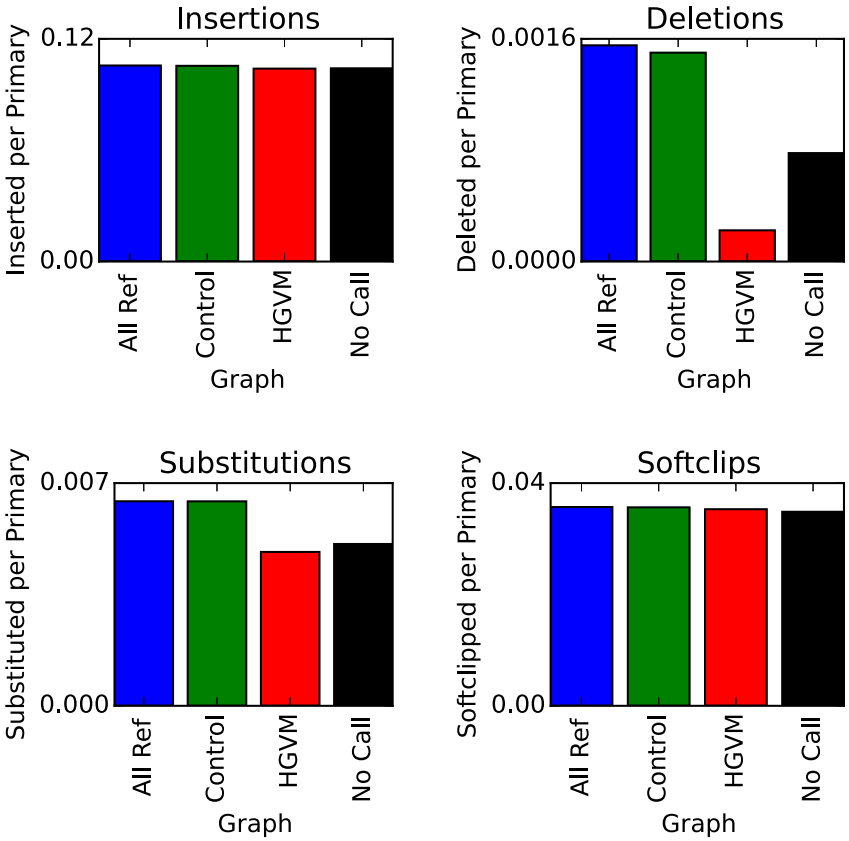

Insertions
Inserted per Primary
0.12
0.00
All Ref
Control
HGVM
No Call
Graph
Deletions
Deleted per Primary
0.0016
0.0000
All Ref
Control
HGVM
No Call
Graph
Substitutions
Substituted per Primary
0.007
0.000
All Ref
Control
HGVM
No Call
Graph
Softclips
Softclipped per Primary
0.04
0.00
All Ref
Control
HGVM
No Call
Graph

Table 5.1: Precision estimation from 25 randomly-sampled calls of variants inducing length changes of 25 bp or more on chromosome 22. From each sample, five called variants were selected randomly. Variants were manually assessed for correspondence to calls for their sample from the 1000 Genomes structural variant set, correspondence to variants in dbSNP 147, and support in the original GRCh38-aligned input reads, using the UCSC Genome Browser. Variants supported either by the 1000 Genomes truth set or by the reads were designated as true variants, while other variants were designated as false variants. Overall, 20 of 25 variants examined were designated as true, producing a precision estimate of 0.80.

Sample	Position	Type	Length (bp)	1KG SV Call	In dbSNP	In Reads	True
HG00513	37661330	Deletion	28		•	•	•
HG00513	44246856	Insertion	33		•	•	•
HG00513	45210755	Insertion	36		•	•	•
HG00513	49343564	Deletion	28		•	•	•
HG00513	49354907	Deletion	57	•	•	•	•
HG00732	23946708	Insertion	1164			•	•
HG00732	22955032	Deletion	38			•	•
HG00732	36731029	Deletion	172	•	•	•	•
HG00732	39195778	Deletion	28		•	•	•
HG00732	41552577	Insertion	40		•	•	•
NA12878	17801142	Deletion	322				
NA12878	24005821	Deletion	52				
NA12878	25232595	Deletion	42		•	•	•
NA12878	44246856	Insertion	33		•	•	•
NA12878	47920718	Deletion	25		•	•	•
NA12889	17717386	Insertion	33		•	•	•
NA12887	20224942	Insertion	29		•	•	•
NA12887	23506174	Insertion	1095				
NA12887	40148663	Insertion	30		•	•	•
NA12889	42119330	Deletion	28		•	•	•
NA12890	17224418	Insertion	300	•		•	•
NA12890	20354675	Deletion	27		•	•	•
NA12890	27258892	Deletion	3150				
NA12890	40652380	Insertion	37		•	•	•
NA12890	43678002	Deletion	6979				

Sample	Total SVs	Called SVs	Recall
HG00513	29	20	0.69
HG00732	31	19	0.61
NA12878	30	21	0.70
NA12889	29	21	0.72
NA12890	32	25	0.78

Table 5.2: Recall statistics for structural variants called by `vg` in five samples, with the structural variant VCF used to construct the graph used as the truth set. Overall recall was 106 of 151 variants, or 0.702.

5.4 Conclusion

The results of the assembly realignment evaluation show that the graph built in this study is a superior reference for chromosome 22 compared to the primary, linear reference currently in use today, for the purpose of variant calling with the `vg` toolkit. In terms of the inserted, deleted, and substituted bases required to represent the CHM1 and CHM13 assemblies on the sample graph called for the synthetic diploid, the variation-containing graph is superior, producing sample graphs that are more similar to the assemblies, and amplifying the effectiveness of the variant caller. However, as evidenced by the overall required insert frequency of about 1% of primary reference bases (Fig. 5.1), and by the relatively low overall structural variant F1 score of 0.76, the `vg` variant caller is, overall, still not particularly good.

On the one hand, the `vg` variant caller is capable of feats which ordinary pileup-based callers cannot accomplish. For example, at `chr22:17224418` in NA12890 in Table 5.1, the `vg` caller successfully used short read data to detect a 300 bp insertion, which the 1000 Genomes structural variation dataset identifies as an Alu insertion [104]. The use of a graph that already contains the Alu insertion in question allows the insertion to be detected using the `vg` caller's simple pileup-based approach, whereas ordinarily the detection of such an event would require sophisticated techniques to handle split or discordantly-paired reads or perform local reassembly [114]. This clearly illustrates the

power of the graph-based approach.

However, on the other hand, the `vg` caller, being a simple pileup-based caller operating on a few manually-tuned heuristics, makes embarrassing mistakes. Take for example the deletion that the caller asserts in NA12890 at `chr22:43678002`, where the caller asserts a heterozygous deletion of 6,979 base pairs. Nothing of the sort is visible in that region in the genome browser. Indeed, given the allelic depths that the caller computes on that particular allele (53 for the reference and 22 for the deletion), in comparison to the baseline coverage estimate it computes of 35 for the sample, it seems likely that in this case the caller has somehow been fooled by some additional extraneous reads supporting the deletion. Cases like this came up in testing, and some heuristics to reject calls with excessively unbalanced allele depths were added to the caller. However, there is clearly more work to be done in characterizing this failure mode, in apportioning blame between the aligner, the graph, and the caller, and in improving the system's resistance to failures of this type.

Moreover, if the caller had a more clever architecture (perhaps, like FreeBayes [23], something based on a concept of per-read support for local haplotypes, instead of on pileups), it could potentially be more robust to a variety of failure modes. As it is, it uses poorly-justified heuristics to try and reduce all of the reads aligned to that potentially-deleted region down to just forward- and reverse-strand "support" values, which it compares against the "support" values of the deleted allele to guess the copy number of each. A more robust read-based framework would potentially allow the caller to single out low-mapping-quality, misaligned, contaminant, ambiguous, or supernumerary reads, and to discount their support, ultimately resulting in better calls, or at least less embarrassing mistakes.

In addition to improvements to the caller, this study has identified ways to improve graph generation. For example, it is hypothesized that the extraneous `N` bases in the graph were generated when `N`-containing potential haplotype strings were aligned

back to the graph in `vg add`. Because the `vg` aligner never matches `N` bases in the query against even `N` bases in the reference, and because `vg add` creates new nodes for all the pieces of sequence in its candidate haplotypes that do not match against bases in the existing graph, the `vg add` logic can produce extra nodes in the final graph when the base graph contains `N` bases. The handling of `N` bases in the input graph to `vg add` needs to be improved, so that those bases are not duplicated when candidate haplotypes are aligned into the graph.

There are also future improvements that should be made to this study's analytical methods. For example, one shortcoming of the structural variant analysis is that, of the broad diversity available in the 1000 Genomes dataset, the five samples analyzed here consisted of three European-ancestry Utah-resident (CEU) individuals, one Southern Han Chinese (CHS) individual, and one Puerto Rican (PUR) individual. These individuals were selected because they were included in the 1000 Genomes structural variation study, and also had high-coverage short-read data aligned to GRCh38 readily available for download. The selection process consisted of taking acceptable samples present in Illumina's Platinum Genomes dataset of a CEU pedigree [20], and augmenting the three samples obtained with two others selected by trying sample names manually. Consequently, they are not particularly representative of the human population, excluding, for example, the entire African continent. Additional individuals also meeting the data-availability criteria likely could have been added to the analysis, and should be included in the future. To fairly evaluate the graph reference constructed in this study, a broader panel of test subjects is needed.

Another reason to test a broader panel of subjects is to avoid overfitting of caller parameters. This study, working only with the five structural variant samples and the one synthetic diploid, had no formal separation between training, test, and validation sets. It is possible that even the relatively low performance of the structural variant caller is overfit, and that it will be reduced when analysis is expanded to more

samples. Additionally, the samples used to evaluate the caller were not removed from the input datasets, so it is possible that the presence of variants private to these individuals in the graph artificially inflates the measured performance, relative to what it would be on a genuinely new sample. Future studies with a formal separation of training, test, and validation samples might benefit from automatic, machine-learning-based tuning of the numerous configurable heuristics available in the variant caller used here, or that will be available in a newly-designed variant caller. In the present study, the heuristics and parameters were hand-tuned, and are almost certainly not optimal.

Overall, the results of this study indicate that graph-based references can be used at chromosome scale to improve variant calling performance. They show that it is possible to combine variation information from disparate sources (in this case, the GRCh38 alternate loci, the 1000 Genomes point variation dataset, and the 1000 Genomes structural variation dataset) to produce a working graph reference. They show, given the construction runtimes and resource requirements achieved at this scale, that it would not be impractical to construct and evaluate a whole-genome graph reference using these techniques. However, they also show that, in order to use such a whole-genome graph effectively, more research into graph-based variant calling is needed. A particular emphasis should be placed on adapting proven, state-of-the-art read- and read-pair-backed approaches to the graph context.

5.5 Availability of Materials

The `hgvm-builder` software, used to coordinate the construction and analysis of the graphs presented here, is available from `https://github.com/BD2KGenomics/hgvm-builder`. The constructed chromosome 22 Human Genome Variation Map graph, along with its succinct structural index and substring search indices, is available from `http://hgwdev.soe.ucsc.edu/~anovak/outbox/builds/2017-05-26/hgvm/`. This particular

graph build has been assigned a UUID of `9ef69e94-a95f-455e-8fca-f705a334968a` by the `hgvm-builder` software.

5.6 Acknowledgements

The author would like to thank Joel Armstrong for performing Cactus alignments used in this work. The author would also like to thank Charles Markello for preparing flat structural variant VCF files and for contributing to the useful `toil-vg` library. The author would like to thank Glenn Hickey for preparing the synthetic diploid sample used in the evaluations, for creating the `hal2vg` tool, and also for contributing to `toil-vg`. The author would like to thank Mike Lin for creating the `vg_docker` system used to package `vg` for this study. The author would like to thank Anna Henderson for copy-editing assistance.

Bibliography

[1] 1000 Genomes Project Consortium. A map of human genome variation from population-scale sequencing. *Nature*, 467(7319):1061–1073, 2010.

[2] 1000 Genomes Project Consortium. An integrated map of genetic variation from 1,092 human genomes. *Nature*, 491(7422):56–65, 1 Nov. 2012.

[3] 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526(7571):68–74, 2015.

[4] P. M. Bodily, M. S. Fujimoto, Q. Snell, D. Ventura, and M. J. Clement. ScaffoldScaffolder: solving contig orientation via bidirected to directed graph reduction. *Bioinformatics*, 32(1):17–24, 2016.

[5] D. Y. Brandt, V. R. Aguiar, B. D. Bitarello, K. Nunes, J. Goudet, and D. Meyer. Mapping bias overestimates reference allele frequencies at the HLA genes in the 1000 Genomes Project phase I data. *G3: Genes— Genomes— Genetics*, 5(5):931–941, 2015.

[6] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. In *SRC Research Report 124*. Digital Equipment Corporation Systems Research Center, 1994.

[7] M. J. P. Chaisson, J. Huddleston, M. Y. Dennis, P. H. Sudmant, M. Malig, F. Hormozdiari, F. Antonacci, U. Surti, R. Sandstrom, M. Boitano, J. M. Landolin, J. A. Stamatoyannopoulos, M. W. Hunkapiller, J. Korlach, and E. E. Eichler. Resolving the complexity of the human genome using single-molecule sequencing. *Nature*, 517(7536):608–611, 29 Jan. 2015.

[8] D. M. Church, V. A. Schneider, T. Graves, K. Auger, F. Cunningham, N. Bouk, H.-C. Chen, R. Agarwala, W. M. McLaren, G. R. Ritchie, et al. Modernizing reference genome assemblies. *PLOS Biology*, 9(7):e1001091, 2011.

[9] G. M. Church. The personal genome project. *Molecular systems biology*, 1(1), 2005.

[10] J. G. Cleary, R. Braithwaite, K. Gaastra, B. S. Hilbush, S. Inglis, S. A. Irvine, A. Jackson, R. Littin, M. Rathod, D. Ware, et al. Comparing variant call files

for performance benchmarking of next-generation sequencing variant calling pipelines. *bioRxiv*, 2015. doi: 10.1101/023754. URL `http://biorxiv.org/content/early/2015/08/03/023754`.

[11] Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, page bbw089, 2016.

[12] P. Danecek, A. Auton, G. Abecasis, C. A. Albers, E. Banks, M. A. DePristo, R. E. Handsaker, G. Lunter, G. T. Marth, S. T. Sherry, et al. The variant call format and VCFtools. *Bioinformatics*, 27(15):2156–2158, 2011.

[13] J. F. Degner, J. C. Marioni, A. A. Pai, J. K. Pickrell, E. Nkadori, Y. Gilad, and J. K. Pritchard. Effect of read-mapping biases on detecting allele-specific expression from RNA-sequencing data. *Bioinformatics*, 25(24):3207–3212, 2009.

[14] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11): 2369–2376, 1999.

[15] M. A. DePristo, E. Banks, R. Poplin, K. V. Garimella, J. R. Maguire, C. Hartl, A. A. Philippakis, G. Del Angel, M. A. Rivas, M. Hanna, et al. A framework for variation discovery and genotyping using next-generation DNA sequencing data. *Nature Genetics*, 43(5):491–498, 2011.

[16] A. Dilthey, C. Cox, Z. Iqbal, M. R. Nelson, and G. McVean. Improved genome inference in the MHC using a population reference graph. *Nature Genetics*, 47 (6):682–688, 2015.

[17] A. Dobin, C. A. Davis, F. Schlesinger, J. Drenkow, C. Zaleski, S. Jha, P. Batut, M. Chaisson, and T. R. Gingeras. STAR: ultrafast universal RNA-seq aligner. *Bioinformatics*, 29(1):15–21, 2013.

[18] R. Durbin. Efficient haplotype matching and storage using the positional Burrows–Wheeler transform (PBWT). *Bioinformatics*, 30(9):1266–1272, 2014.

[19] D. Earl, N. Nguyen, G. Hickey, R. S. Harris, S. Fitzgerald, K. Beal, I. Seledtsov, V. Molodtsov, B. J. Raney, H. Clawson, et al. Alignathon: a competitive assessment of whole-genome alignment methods. *Genome Research*, 24(12): 2077–2089, 2014.

[20] M. Eberle, M. Kallberg, H. Chuang, P. Tedder, S. Humphray, D. Bentley, and E. Margulies. Platinum Genomes: A systematic assessment of variant accuracy using a large family pedigree. In *60th Annual Meeting of The American Society of Human Genetics*, pages 22–26, 2013.

[21] M. A. Eberle, F. Epameinondas, K. Peter, K. Morten, B. L. Moore, M. A. Bekritsky, I. Zamin, C. Han-Yu, S. J. Humphray, A. L. Halpern, K. Semyon,

E. H. Margulies, M. Gil, and D. R. Bentley. A reference dataset of 5.4 million phased human variants validated by genetic inheritance from sequencing a three-generation 17-member pedigree. Technical report, 2016.

[22] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Symposium on Foundations of Computer Science (FOCS 2000)*, pages 390–398. IEEE, 2000.

[23] E. Garrison and G. Marth. Haplotype-based variant detection from short-read sequencing. *arXiv preprint arXiv:1207.3907*, 2012.

[24] E. Garrison, J. Sirén, A. M. Novak, G. Hickey, J. M. Eizenga, E. T. Dawson, W. Jones, M. F. Lin, B. Paten, and R. Durbin. Sequence variation aware references and read mapping with the variation graph toolkit, in preparation.

[25] Genome Reference Consortium. GRCh37, 2009. URL `https://www.ncbi.nlm.nih.gov/grc/human/data?asm=GRCh37`.

[26] Genome Reference Consortium. Announcing GRCh38, 2013. URL `http://genomeref.blogspot.com/2013/12/announcing-grch38.html`.

[27] `GFA-spec` contributors. GFA-spec. URL `https://github.com/GFA-spec/GFA-spec`.

[28] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms (SEA 2014)*, pages 326–337, 2014.

[29] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the fourteenth annual ACM-SIAM symposium on discrete algorithms*, pages 841–850. Society for Industrial and Applied Mathematics, 2003.

[30] A. Halevy, P. Norvig, and F. Pereira. The unreasonable effectiveness of data. *Intelligent Systems, IEEE*, 24(2):8–12, 2009.

[31] R. Harris. *Improved pairwise alignment of genomic DNA*. PhD thesis, The Pennsylvania State University, 2007.

[32] J. Harrow, A. Frankish, J. M. Gonzalez, E. Tapanari, M. Diekhans, F. Kokocinski, B. L. Aken, D. Barrell, A. Zadissa, S. Searle, I. Barnes, A. Bignell, V. Boychenko, T. Hunt, M. Kay, G. Mukherjee, J. Rajan, G. Despacio-Reyes, G. Saunders, C. Steward, R. Harte, M. Lin, C. Howald, A. Tanzer, T. Derrien, J. Chrast, N. Walters, S. Balasubramanian, B. Pei, M. Tress, J. M. Rodriguez, I. Ezkurdia, J. van Baren, M. Brent, D. Haussler, M. Kellis, A. Valencia, A. Reymond, M. Gerstein, R. Guigó, and T. J. Hubbard. GENCODE: the reference human genome annotation for the ENCODE project. *Genome Research*, 22(9):1760–1774, Sept. 2012.

[33] D. Haussler, M. Smuga-Otto, B. Paten, A. M. Novak, S. Nikitin, M. Zueva, and D. Miagkov. A flow procedure for the linearization of genome sequence graphs. In *International Conference on Research in Computational Molecular Biology*, pages 34–49. Springer, 2017.

[34] S. Heber, M. Alekseyev, S.-H. Sze, H. Tang, and P. A. Pevzner. Splicing graphs and EST assembly problem. *Bioinformatics*, 18 Suppl 1:S181–8, 2002.

[35] G. Hickey, B. Paten, D. Earl, D. Zerbino, and D. Haussler. HAL: a hierarchical format for storing and analyzing multiple genome alignments. *Bioinformatics*, page btt128, 2013.

[36] R. Horton, R. Gibson, P. Coggill, M. Miretti, R. J. Allcock, J. Almeida, S. Forbes, J. G. Gilbert, K. Halls, J. L. Harrow, et al. Variation analysis and gene annotation of eight MHC haplotypes: the MHC Haplotype Project. *Immunogenetics*, 60(1):1–18, 2008.

[37] K. L. Howe, B. J. Bolt, S. Cain, J. Chan, W. J. Chen, P. Davis, J. Done, T. Down, S. Gao, C. Grove, et al. WormBase 2016: expanding to enable helminth genomic research. *Nucleic Acids Research*, page gkv1217, 2015.

[38] L. Huang, V. Popic, and S. Batzoglou. Short read alignment with populations of genomes. *Bioinformatics*, 29(13):i361–i370, 2013.

[39] K. Hudson, R. Lifton, B. Patrick-Lake, et al. The precision medicine initiative cohort program — building a research foundation for 21st century medicine, 2015. URL `https://www.nih.gov/sites/default/files/research-training/initiatives/pmi/pmi-working-group-report-20150917-2.pdf`.

[40] International HapMap Consortium. Integrating ethics and science in the International HapMap Project. *Nature Reviews Genetics*, 5(6):467, 2004.

[41] International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 15 Feb. 2001.

[42] M. Jäger, M. Schubach, T. Zemojtel, K. Reinert, D. M. Church, and P. N. Robinson. Alternate-locus aware variant calling in whole genome sequencing. *Genome Medicine*, 8(1):130, 2016.

[43] J. Kans. *Entrez Direct: E-utilities on the UNIX Command Line*. National Center for Biotechnology Information (US), 8 Feb. 2016.

[44] D. Karolchik. The new GRCh38 human genome browser has arrived!, 2014. URL `https://groups.google.com/a/soe.ucsc.edu/d/msg/genome-announce/52Kv41YBXNY/n__rnGKMgKwJ`.

[45] W. J. Kent. BLAT–the BLAST-like alignment tool. *Genome Research*, 12(4): 656–664, Apr. 2002.

[46] B. Langmead. Introduction to the Burrows-Wheeler Transform and FM index, 2013. URL `http://www.cs.jhu.edu/~langmea/resources/bwt_fm.pdf`.

[47] B. Langmead, C. Trapnell, M. Pop, S. L. Salzberg, et al. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol*, 10(3):R25, 2009.

[48] H. Li. Exploring single-sample SNP and INDEL calling with whole-genome *de novo* assembly. *Bioinformatics*, 28(14):1838–1844, 2012.

[49] H. Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv preprint arXiv:1303.3997*, 2013.

[50] H. Li. Burrows-Wheeler Aligner, 2014. URL `https://github.com/lh3/bwa`.

[51] H. Li. The new ALT mapping mode of BWA-MEM, 2014. URL `https://sourceforge.net/p/bio-bwa/mailman/message/32845712/`.

[52] H. Li. BGT: efficient and flexible genotype query across many samples. *Bioinformatics*, page btv613, 2015.

[53] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[54] H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

[55] H. Li and N. Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 11(5):473–483, 2010.

[56] H. Li, J. Ruan, and R. Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Research*, 18(11): 1851–1858, 2008.

[57] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, et al. The Sequence Alignment/Map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[58] A. Limasset, B. Cazaux, E. Rivals, and P. Peterlongo. Read mapping on de Bruijn graphs. *BMC Bioinformatics*, 17(1):237, 16 June 2016.

[59] P.-R. Loh, P. Danecek, P. F. Palamara, C. Fuchsberger, Y. A. Reshef, H. K. Finucane, S. Schoenherr, L. Forer, S. McCarthy, G. R. Abecasis, et al. Reference-based phasing using the Haplotype Reference Consortium panel. *Nature Genetics*, 48(11):1443–1448, 2016.

[60] G. Lunter. Fast haplotype matching in very large cohorts using the Li and Stephens model. *bioRxiv*, 2016. doi: 10.1101/048280. URL `http://biorxiv.org/content/early/2016/04/12/048280`.

[61] G. Lunter and M. Goodson. Stampy: a statistical algorithm for sensitive and fast mapping of Illumina sequence reads. *Genome Research*, 21(6):936–939, 2011.

[62] J. A. L. MacArthur, J. Morales, R. E. Tully, A. Astashyn, L. Gil, E. A. Bruford, P. Larsson, P. Flicek, R. Dalgleish, D. R. Maglott, and F. Cunningham. Locus Reference Genomic: reference sequences for the reporting of clinically relevant sequence variants. *Nucleic Acids Research*, 42(Database issue):D873–8, Jan. 2014.

[63] S. Maciuca, C. del Ojo Elias, G. McVean, and Z. Iqbal. A natural encoding of genetic variation in a Burrows-Wheeler transform to enable mapping and genome inference. In M. Frith and C. N. Storm Pedersen, editors, *Algorithms in Bioinformatics*, pages 222–233, Cham, 2016. Springer International Publishing.

[64] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[65] G. Marçais and C. Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 15 Mar. 2011.

[66] J. Marshall et al. The Variant Call Format (VCF) 4.2 specification, 2013. URL `http://samtools.github.io/hts-specs/VCFv4.2.pdf`.

[67] S. McCarthy, S. Das, W. Kretzschmar, O. Delaneau, A. R. Wood, A. Teumer, H. M. Kang, C. Fuchsberger, P. Danecek, K. Sharp, et al. A reference panel of 64,976 haplotypes for genotype imputation. *Nature Genetics*, 2016.

[68] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, and M. A. DePristo. The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Research*, 20(9):1297–1303, Sept. 2010.

[69] P. Medvedev and M. Brudno. Maximum likelihood genome assembly. *Journal of Computational Biology*, 16(8):1101–1116, 2009.

[70] P. Medvedev, K. Georgiou, G. Myers, and M. Brudno. Computability of models for sequence assembly. In *Algorithms in Bioinformatics*, pages 289–301, Cham, 2007. Springer International Publishing.

[71] L. R. Meyer, A. S. Zweig, A. S. Hinrichs, D. Karolchik, R. M. Kuhn, M. Wong, C. A. Sloan, K. R. Rosenbloom, G. Roe, B. Rhead, et al. The UCSC genome browser database: extensions and updates 2013. *Nucleic Acids Research*, 41(D1): D64–D69, 2013.

[72] K. H. Miga, Y. Newton, M. Jain, N. Altemose, H. F. Willard, and W. J. Kent. Centromere reference models for human chromosomes X and Y satellite arrays. *Genome Research*, 24(4):697–707, 2014.

[73] R. E. Mills, W. S. Pittard, J. M. Mullaney, U. Farooq, T. H. Creasy, A. A. Mahurkar, D. M. Kemeza, D. S. Strassler, C. P. Ponting, C. Webber, and S. E. Devine. Natural genetic variation caused by small insertions and deletions in the human genome. *Genome Research*, 21(6):830–839, June 2011.

[74] A. E. Minoche, J. C. Dohm, and H. Himmelbauer. Evaluation of genomic high-throughput sequencing data generated on illumina HiSeq and genome analyzer systems. *Genome Biology*, 12(11):R112, 8 Nov. 2011.

[75] E. W. Myers. The fragment assembly string graph. *Bioinformatics*, 21 Suppl 2: ii79–85, 1 Sept. 2005.

[76] F. A. Nothaft, M. Massie, T. Danford, Z. Zhang, U. Laserson, C. Yeksigian, J. Kottalam, A. Ahuja, J. Hammerbacher, M. Linderman, et al. Rethinking data-intensive science using scalable analytics systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 631–646. ACM, 2015.

[77] A. M. Novak, Y. Rosen, D. Haussler, and B. Paten. Canonical, stable, general mapping using context schemes. *Bioinformatics*, page btv435, 2015.

[78] A. M. Novak, E. Garrison, and B. Paten. A graph extension of the positional Burrows-Wheeler transform and its applications. In M. Frith and C. N. Storm Pedersen, editors, *Algorithms in Bioinformatics*, pages 246–256, Cham, 2016. Springer International Publishing.

[79] A. M. Novak, G. Hickey, E. Garrison, S. Blum, A. Connelly, A. Dilthey, J. Eizenga, M. S. Elmohamed, S. Guthrie, A. Kahles, et al. Genome graphs. *bioRxiv*, 2017. doi: 10.1101/101378. URL `http://biorxiv.org/content/early/2017/01/18/101378`.

[80] S. Ortiz. The problem with cloud-computing standardization. *Computer*, 44(7): 13–16, 2011.

[81] B. Paten, M. Diekhans, D. Earl, J. S. John, J. Ma, B. Suh, and D. Haussler. Cactus graphs for genome comparisons. *Journal of Computational Biology*, 18 (3):469–481, 2011.

[82] B. Paten, D. Earl, N. Nguyen, M. Diekhans, D. Zerbino, and D. Haussler. Cactus: Algorithms for genome multiple sequence alignment. *Genome Research*, 21(9):1512–1528, 2011.

[83] B. Paten, A. Novak, and D. Haussler. Mapping to a reference genome structure. *arXiv preprint arXiv:1404.5010*, 2014.

[84] B. Paten, A. M. Novak, J. M. Eizenga, and E. Garrison. Genome graphs and the evolution of genome inference. *Genome Research*, 27(5):665, 2017.

[85] B. Paten, A. M. Novak, E. Garrison, and G. Hickey. Superbubbles, ultrabubbles and cacti. *bioRxiv*, 2017. doi: 10.1101/101493. URL `http://biorxiv.org/content/early/2017/01/18/101493`.

[86] B. Pei, C. Sisu, A. Frankish, C. Howald, L. Habegger, X. J. Mu, R. Harte, S. Balasubramanian, A. Tanzer, M. Diekhans, A. Reymond, T. J. Hubbard, J. Harrow, and M. B. Gerstein. The GENCODE pseudogene resource. *Genome Biology*, 13(9):R51, 26 Sept. 2012.

[87] P. A. Pevzner, H. Tang, and M. S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences of the United States of America*, 98(17):9748–9753, 14 Aug. 2001.

[88] T. M. Powledge. Human genome project completed. *Genome Biology*, 4(1): spotlight–20030415, 2003.

[89] K. Prüfer, K. Munch, I. Hellmann, K. Akagi, J. R. Miller, B. Walenz, S. Koren, G. Sutton, C. Kodira, R. Winer, et al. The bonobo genome compared with the chimpanzee and human genomes. *Nature*, 486(7404):527–531, 2012.

[90] M. A. Quail, M. Smith, P. Coupland, T. D. Otto, S. R. Harris, T. R. Connor, A. Bertoni, H. P. Swerdlow, and Y. Gu. A tale of three next generation sequencing platforms: comparison of Ion Torrent, Pacific Biosciences and Illumina MiSeq sequencers. *BMC Genomics*, 13(1):341, 2012.

[91] A. Rimmer, H. Phan, I. Mathieson, Z. Iqbal, S. R. F. Twigg, WGS500 Consortium, A. O. M. Wilkie, G. McVean, and G. Lunter. Integrating mapping-, assembly- and haplotype-based approaches for calling variants in clinical sequencing applications. *Nature Genetics*, 46(8):912–918, Aug. 2014.

[92] C. Saunders. About gVCF, 2013. URL `https://sites.google.com/site/gvcftools/home/about-gvcf`.

[93] V. Schneider. GRC contract email UM-2040. Private communication, 2015.

[94] V. Schneider and D. Church. Genome reference consortium. In *The NCBI Handbook [Internet]*. National Center for Biotechnology Information (US), Bethesda, MD, 2 edition, 2013. URL `http://www.ncbi.nlm.nih.gov/books/NBK153600/`.

[95] S. T. Sherry, M.-H. Ward, M. Kholodov, J. Baker, L. Phan, E. M. Smigielski, and K. Sirotkin. dbSNP: the NCBI database of genetic variation. *Nucleic Acids Research*, 29(1):308–311, 2001.

[96] Simons Foundation. Simons genome diversity project, 2014. URL `https://www.simonsfoundation.org/life-sciences/simons-human-diversity-project/`.

[97] Simons Foundation. Simons genome diversity project dataset, 2017. URL https://www.simonsfoundation.org/life-sciences/simons-genome-diversity-project-dataset/.

[98] J. T. Simpson and R. Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, 26(12):i367–73, 15 June 2010.

[99] J. T. Simpson and R. Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*, 22(3):549–556, 2012.

[100] J. Sirén. Indexing variation graphs. In *2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 13–27. SIAM, 2017.

[101] J. Sirén, N. Välimäki, V. Mäkinen, and G. Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *String Processing and Information Retrieval*, pages 164–175. Springer, 2009.

[102] J. Sirén, N. Valimaki, and V. Makinen. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11(2):375–388, 2014.

[103] K. M. Steinberg, V. A. Schneider, T. A. Graves-Lindsay, R. S. Fulton, A. Richa, H. John, S. A. Shiryev, M. Aleksandr, S. Urvashi, W. C. Warren, D. M. Church, E. E. Eichler, and R. K. Wilson. Single haplotype assembly of the human genome from a hydatidiform mole. *Genome Research*, 24(12):2066–2076, 2014.

[104] P. H. Sudmant, T. Rausch, E. J. Gardner, R. E. Handsaker, A. Abyzov, J. Huddleston, Y. Zhang, K. Ye, G. Jun, M. H.-Y. Fritz, et al. An integrated map of structural variation in 2,504 human genomes. *Nature*, 526(7571):75–81, 2015.

[105] J. Talton. The economics of 7.5 billion people on one planet. *The Seattle Times*, 17 May 2017. URL http://www.seattletimes.com/business/economy/the-economics-of-7-5-billion-people-on-one-planet/.

[106] A. Tan, G. R. Abecasis, and H. M. Kang. Unified representation of genetic variants. *Bioinformatics*, 31(13):2202–2204, 1 July 2015.

[107] vg contributors. vg. URL https://github.com/vgteam/vg.

[108] The ENCODE Project Consortium. An integrated encyclopedia of DNA elements in the human genome. *Nature*, 489(7414):57–74, 6 Sept. 2012.

[109] The FASTG Format Specification Working Group. The FASTG format specification (v1.00), 12 Dec. 2012. URL http://fastg.sourceforge.net/FASTG_Spec_v1.00.pdf.

[110] The MHC Sequencing Consortium. Complete sequence and gene map of a human major histocompatibility complex. *Nature*, 401(6756):921–923, 1999.

[111] K. Varda. Protocol buffers: Google's data interchange format, 2008. URL `https://opensource.googleblog.com/2008/07/protocol-buffers-googles-data.html`.

[112] J. Vivian, A. A. Rao, F. A. Nothaft, C. Ketchum, J. Armstrong, A. Novak, J. Pfeil, J. Narkizian, A. D. Deran, A. Musselman-Brown, et al. Toil enables reproducible, open source, big biomedical data analyses. *Nature Biotechnology*, 35(4):314–316, 2017.

[113] A. M. Weiner. An abundant cytoplasmic 7S RNA is complementary to the dominant interspersed middle repetitive DNA sequence family in the human genome. *Cell*, 22(1):209–218, 1980.

[114] J. H. Wildschutte, A. Baron, N. M. Diroff, and J. M. Kidd. Discovery and characterization of Alu repeat sequences via precise local read assembly. *Nucleic Acids Research*, page gkv1089, 2015.

[115] J. M. Zook, B. Chapman, J. Wang, D. Mittelman, O. Hofmann, W. Hide, and M. Salit. Integrating human sequence data sets provides a resource of benchmark SNP and indel genotype calls. *Nature Biotechnology*, 32(3):246–251, Mar. 2014.