Leveraging Similar Regions to Improve Genome Data Processing

Kristal Curtis

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-199
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-199.html

September 15, 2015

Leveraging Similar Regions to Improve Genome Data Processing

by

Kristal Lyn Curtis

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor David A. Patterson, Chair
Professor Armando Fox
Professor Adam Arkin

Fall 2015

Leveraging Similar Regions to Improve Genome Data Processing

Abstract

Leveraging Similar Regions to Improve Genome Data Processing

by

Kristal Lyn Curtis

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David A. Patterson, Chair

Though DNA sequencing has improved dramatically over the past decade, variant calling, which is the process of reconstructing a patient's genome from the reads that the sequencers produce, remains a difficult problem, largely due to the genome's redundant structure. In this thesis, we describe SiRen, our algorithm for characterizing the genome's structure in a way that makes sense from the perspective of the reads themselves. We use the term *similar regions* to refer to the areas of redundancy that we have identified. We then confirm that the similar regions are characterized by low variant calling accuracy. We show that the structure of the similar regions provides a platform for repairing alignment errors, thus leading to significantly improved variant calling accuracy.

To my village

According to a well-known African proverb, "It takes a village to raise a child." It also took a village to raise this dissertation, and I truly appreciate each person involved.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank the many people who helped and supported me during my time at Berkeley.

I was very fortunate to have David Patterson and Armando Fox as my advisors. From the beginning of my graduate school career, they showed an interest in my success. Over the years, they have shared lots of brilliant ideas as well as given me space to develop ideas on my own. In addition, they have always impressed me with their multi-faceted lives, including tremendous engagement with family, sports, music, and theatre. I also appreciate their patience and flexibility as I learned to combine scholarship and motherhood. I am truly fortunate to have had the opportunity to work with such incredible individuals.

I am grateful to Yun Song and Adam Arkin for serving on my thesis committee. Both Yun and Adam are wonderful scholars and professors, and their involvement rounded out the advice that Dave and Armando provided.

As a graduate student at Berkeley, I was lucky to have incredible labmates. I would especially like to thank Ameet Talwalkar and Matei Zaharia, my collaborators on the work presented in this thesis. Each of them devoted significant time and energy to helping me with both the high-level ideas as well as the implementation of the work involved.

I also really enjoyed being a part of the AMP-X group, which focused on genomics projects. The times we spent in stand-up progress meetings and sharing meals together really energized my work. Frank Austin Nothaft and I had many helpful discussions related to how to use the similar regions. I would also like to thank Bill Bolosky, Ma'ayan Bresler, Christos Kozanitis, Jesse Liptrap, Matt Massie, Julie Newcomb, Ravi Pandya, and Jonathan Terhorst for tackling a new area with me.

I have been fortunate to be part of the RAD and AMP Labs since shortly after starting my studies at Berkeley. I received incredible help and support from many of my labmates, on course assignments as well as research problems. I would particularly like to thank Michael Armbrust, Peter Bodík, Archana Ganapathi, Evan Sparks, and Shivaram Venkataraman. I appreciate the understanding and good humor of everyone in the lab and particularly the lab director, Michael Franklin, as my infant son frequently accompanied me to meetings. It was also wonderful to receive administrative and technical support from the AMP Lab staff, including Kattt Atchley, Carlyn Chinen, Jon Kuroda, Sean McMahon, and Boban Zarkovich. They managed to smooth out many headaches and make me smile at the same time.

I benefitted immensely from my participation in WICSE, which gave me a space to share my thoughts as well as a great opportunity to make friends. Many thanks to Sheila Humphreys for all her tireless efforts to support WICSE. I am also grateful for all the friends I made through WICSE, especially Erika Chin, Cynthia Sturton, Erin Summers, and Beth Trushkowsky.

I could not have finished the work for this degree without the unflagging support of my family and community. My parents always believed that I could succeed at Berkeley, and they never stopped believing in me through the years. My husband, Ryan Curtis, has always loved me and has been a vital support to me, as well as always being willing to lend

an ear when I needed to blow off steam. Ryan, my son Nathan, and my daughter (still in utero at the time of this writing) have been a beautiful source of inspiration for me. Many thanks to Meghan Smith, Sophie Cooper, and Cornerstone Children's Center for providing excellent childcare for my son. I also appreciate my church communities at Veritas, Christ Church of Berkeley, and City Church of San Francisco, which have been an immense source of friendship and encouragement. Above all, I give thanks to my Lord and Savior Jesus Christ; Your grace has been sufficient for all my weaknesses.

Finally, I appreciate the financial support that enabled me to complete this research. I am grateful to have been supported by a UC Berkeley Chancellor's Fellowship and a NSF Graduate Research Fellowship, as well as by NSF CISE Expeditions Award CCF-1139158, DOE Award SN10040 DE-SC0012463, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, IBM, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple, Inc., Blue Goji, Bosch, Cisco, Cray, Cloudera, EMC2, Ericsson, Facebook, Guavus, HP, Huawei, Informatica, Intel, Microsoft, NetApp, Pivotal, Samsung, Schlumberger, Splunk, Virdata and VMware.

Chapter 1

Introduction

1.1 The Promise of Genomics

Since the completion of the Human Genome Project in 2003, there has been a tremendous surge of excitement and momentum in genomics. As the technology to sequence a genome has improved dramatically in the last decade or so, an immense amount of energy has been devoted to discovering and developing new ways to utilize genomic information in fields ranging from medicine to crop breeding. A diverse group of researchers and practitioners is now working to leverage the knowledge gained by mapping the human genome, as well as other genomes, to make advances that were previously impossible.

Medicine is an area in which genomics has the potential to make significant changes. For example, the Obama administration recently announced its plans to increase research funding via its "Precision Medicine Initiative" [9]. The idea of precision medicine is that rather than applying the same standard of care to everyone, a patient's care should be targeted to his or her particular genetic makeup, or *genotype*. The goal of customized care is to improve patient outcomes while lessening detrimental side effects from inappropriate treatments. The University of California system has launched a similar effort to enhance patient care, diagnosis, and treatment by integrating clinical, genomic, environmental, and other types of data [131, 64].

One of the main areas where researchers and clinicians are excited about using genomics and precision medicine is in oncology, *i.e.,* the diagnosis and treatment of cancer [101]. Cancer is considered a particularly good target for applying genomics since cancer is a molecular disease, *i.e.,* it is caused by mutations in a person's cells [40]. For example, normal cells go through a life cycle that culminates in their death; however, cancer cells mutate such that the cell death mechanisms are disabled. Because cancer involves genetic mutations, sequencing is becoming important to understanding its development. First, it is often desirable to sequence both a patient's normal and cancer cells in order to enumerate the mutations that have occurred between them. It is also important to sequence a tumor over time, since mutations continue to accumulate as the cancer grows [11].

A convincing use of personalized oncology is the case of Dr. Lukas Wartman, a genomics researcher at Washington University [61]. After he was diagnosed with leukemia, his colleagues sequenced his genome to learn more about how his cancer was progressing. They learned that his cancer was fueled by a rogue gene that was producing too much of a particular protein. Fortunately, the gene could be inhibited by a drug that was developed for another purpose (treating kidney cancer), and he went into remission.

Though cancer is a compelling application area for genomics, genomics can also help with diagnosing other types of illnesses. For example, genomics was used to save the life of Joshua Osborn, a 14-year-old boy suffering from a long illness that had culminated in a coma [144, 135]. The doctors treating him at the University of Wisconsin School of Medicine tried guessing at the cause of his illness, doing separate tests to rule out each potential cause and wasting precious time along the way. Desperate, they reached out to their colleagues at the University of California, San Francisco, who had developed an experimental procedure to use DNA sequencing to identify pathogens. The researchers sequenced the fluid accumulating in Osborn's brain, and in less than two hours, after discarding the human DNA, they identified a bacteria called Leptospira that could be treated effectively with penicillin. Soon after receiving the antibiotic, the boy recovered.

Another exciting application of genomics is the development of targeted therapies, as some of the energy in pharmaceuticals shifts from universally applied drugs to drugs that are targeted at patients with particular biomarkers [93]. In the case of cancer, these targeted therapies have the potential not only to be more effective than general drugs but also to be much easier for patients to tolerate than traditional chemotherapy, which has notorious side effects such as nausea and hair loss. For example, Gleevec was developed to target chronic myeloid leukemia by blocking an enzyme that exists only in cancer cells, thus causing the cancer cells to die and leaving healthy cells unaffected. Another targeted therapy is Herceptin, which targets HER2-positive (HER2+) breast cancer. HER2 is a protein that, when overproduced, causes cells to reproduce uncontrollably; Herceptin inhibits this action in cells that are HER2+. Exciting developments like these have stimulated many efforts to develop additional targeted therapies, and patients are sure to benefit as their use expands.

1.2 Obstacles

The main problem at the center of all these efforts is deducing the relationship between a person's *genotype* (*i.e.,* content of their genome) and their *phenotype* (*i.e.,* physical characteristics such as drug response). This inference is difficult, since other factors besides the genotype can influence the phenotype. The most basic precondition for learning this relationship, of course, is to accurately determine the person's genotype. In order to gather information about a patient's genotype, we begin by sequencing a sample of their DNA.

Though DNA sequencers have made mighty strides in the past decade, dramatically improving their throughput and reducing their cost, sequencers are far from capable of reading a patient's genome in a single, long pass. Instead, sequencers read a genome in

short segments called *reads*. After reads have been produced, we must tackle the challenging computational problem of reconstructing the patient's genome from the short reads. The reconstruction process is called *variant calling* (VC), because its goal is to determine the differences (or variants) between the patient's genome and a representative human genome called the *reference genome*. Because humans are 99.9% identical, the number of variants is small relative to the size of the genome; the size of the human genome is approximately three billion bases, so we expect to identify about three million variants when sequencing a new genome.

To make decisions about a person's medical care based on their variant calls, it is crucial that tools used to produce those variant calls be both efficient and accurate. However, VC is a challenging process. Due to its computational cost, variant calling incurs high monetary costs. A more fundamental issue is that because of the genome's innate structural features, variant calling is susceptible to errors.

A crucial step usually involved in VC is called *alignment*, which is the problem of determining where in the genome a read likely originated. Once all reads have been aligned, for each location, variant calling tools use the information presented in the reads overlapping that location to infer whether or not the sample differs from the reference. If the genome were a random string of three billion characters, variant calling would be straightforward, since it would be highly unlikely for a read to align to multiple locations. However, in reality, the genome is characterized by both near and exact duplication. Therefore, ambiguity arises as to where many reads should be aligned, and the resulting uncertainty propagates throughout the analysis pipeline to cause difficulties for variant calling as well.

1.3 Challenges and Contributions

In this thesis, we tackle the complexities presented by the genome's redundancy. In particular, we have sought to characterize the redundancy from a perspective driven by the short read data. We use the term *similar regions* to refer to duplicated areas of the genome that we have identified as having a significant impact on variant calling.

The goal of this thesis is to identify similar regions that not only correspond to difficult areas for variant calling but can also be used to improve VC accuracy. In our efforts, we have encountered several challenges. This thesis presents our contributions addressing the following:

Identifying the similar regions: It is well known that the genome has duplicated sequence. As we discuss in Chapter 3, there are several existing approaches to characterizing the genome's redundancy. However, these approaches are unsuitable to our problem for various reasons, including their unstructured output or large size. Thus, one of the primary challenges we address in this thesis is devising an approach that identifies a manageable amount of the genome as being redundant. It is also crucial for the approach to scale to the large problem size; computing on the genome is expensive due to the N^2 possible similarity relationships.

Validating the similar regions: One way to find duplication in the genome is the method favored by Repbase,[1] a popular database of repetitive sequence in the genome, which involves manual curation, whereby researchers contribute repeats that they have identified. However, manual curation is unwieldy, due to the size of the genome. Our approach relies on automatically identifying duplication. While more convenient and consistent than manual curation, an automatic approach must be validated to ensure that its output is correct. Thus, one challenge is how to validate our output to ensure that the regions our algorithm identifies correspond to those in which alignment errors occur. We also wanted to confirm that alignment errors lead to variant calling errors.

Alignment errors: After we have identified the proper similar regions and verified that the similar regions indeed correspond to poor VC accuracy, a remaining challenge is to develop a strategy for improving the VC accuracy. One method for improving VC accuracy would involve developing a new VC algorithm. Another approach is to target alignment errors, which contribute to difficulties with variant calling; any new VC algorithm would also have to contend with alignment errors. It may also be necessary to identify which factors that contribute to alignment errors can be targeted with the similar regions. The goal for an approach that corrects alignment errors would be to see variant calling errors decline as well.

In this thesis, we addressed these challenges [25]. First, we constrained the problem of locating the similar regions using the paradigm of short reads. As we describe in Chapter 2, genomic data is produced in short segments called reads, which processing algorithms must reconstruct to recover the sample genome. In order to devise a simple algorithm, we used the fact that we knew our similar regions would be used for read processing. We also designed our algorithm to be distributed, which enabled us to shrink our development cycle and achieve an implementation that facilitates exploration for parameter selection. Our algorithm is called *SiRen*, because we use it to locate the similar regions.

The SiRen algorithm is unsupervised (*i.e.,* it does not rely on labeled input data); thus, we had to develop a method for verifying that its output actually located interesting areas of redundancy. We therefore devised and executed a validation strategy to confirm that the similar regions we located are highly correlated with difficulties for both alignment and variant calling. In order to do so, we had to create a metric for identifying difficult alignments by considering multiple aligners and then assessing the level of agreement between them. After we demonstrated that difficult alignments were overwhelmingly in similar regions, we had to assess whether the problematic alignments in similar regions propagated to result in VC errors. We performed an end-to-end analysis that shows that indeed, VC in the similar regions is much more difficult than in the unique regions.

Confirming that the similar regions that we have identified coincide with alignment and VC errors is necessary but not sufficient to achieve our goal of ensuring that the similar regions can be used to improve VC accuracy. We also sought to develop a strategy for repairing alignment errors in the similar regions, making use of the properties of the similar regions themselves. In order to do so, we extended the SiRen algorithm to identify relation-

[1] http://www.girinst.org/repbase/

ships among the similar regions, so that we obtain a structure that indicates which regions are similar to which other regions. We also devised a realignment strategy to repair known alignment errors, using the structure that the extended SiRen algorithm identifies. Given our realignment techniques, we validated that they substantially improve the VC accuracy in the similar regions.

We may therefore summarize the contributions of this thesis as follows:

- We present the SiRen algorithm that defines and locates the similar regions.
- We provide evidence that our similar regions correspond to difficulties in alignment and variant calling.
- We develop a strategy for leveraging the structure in our similar regions to repair known alignment errors, thereby improving VC accuracy in the similar regions.

1.4 Thesis Organization

This thesis is organized as follows. In Chapter 2, we define key genomics terms that form the background for the material we present. We also give a short history of DNA sequencing, from the original technology, called Sanger sequencing, through the current dominant technology, called second-generation sequencing. We also preview long-read technology, an emerging area with the potential to disrupt second-generation sequencing. Once DNA has been sequenced, a processing pipeline transforms the raw output of the sequencer to variant calls, which express the difference between the sample genome and a reference genome. The bulk of the chapter presents the key stages of the pipeline, most notably alignment and variant calling.

We continue in Chapter 3 by describing near and exact duplication in the genome as well as discussing the challenges that duplication poses to processing sequencing data. We then present the SiRen algorithm for locating similar regions in the genome, from the perspective of the short reads themselves. The chapter ends with a description of SiRen's distributed implementation.

In Chapter 4, we present our validation of the similar regions. We begin by giving basic metrics of the SiRen output, including the number of clusters we identified as well as the size of the similar regions. Then, we provide a more application-driven evaluation. First, we show how the similar regions affect alignment. We devise a measure to quantify alignment difficulty and then show that difficult alignments occur overwhelmingly in the similar regions. Then, we perform an end-to-end analysis showing how alignment errors propagate through the pipeline to present challenges to variant calling as well. We provide evidence that variant calling in the similar regions is much less accurate than in the unique regions. Finally, we perform a quantitative comparison between the similar regions and other "blacklists" (*i.e.,* lists of sequence to be ignored due to the difficulty of processing the sequence) that practitioners use. We discovered that the similar regions are similar in size to one other blacklist, GEM [28].

In order to distinguish our similar regions from other blacklists, we continue in Chapter 5 by showing the value of the structure in our similar regions. This structure is in contrast to blacklists like GEM that are simple binary indicators of whether or not a region is similar to another genome sequence. We begin the chapter by describing how we go beyond treating the similar regions like a binary list by constructing relationships between the similar regions. Then, we motivate the importance of repairing alignment errors in the similar regions by showing that modulo structural variants, correcting alignment errors in the similar regions yields comparable VC accuracy between the similar and unique regions. We then demonstrate that we can use relationships between the similar regions, which we call similar region *families*, to repair known alignment errors. We complete the chapter by discussing our perspectives about how to recognize alignment errors automatically to fully automate the realignment process.

We end the thesis in Chapter 6 by presenting our conclusions and ideas for future work. After reflecting on what we have accomplished, we outline how to build on our work to create an alignment repair pipeline stage as a preprocessing step to variant calling that will enhance VC accuracy in the similar regions. We also present our ideas about how to use the pipeline stage to enable new biological analyses.

Chapter 2

Working with Genomic Data

2.1 Introduction

The field of genomics has undergone dramatic changes during the recent past. In just over a decade since the Human Genome Project was completed, the interest in and capability for working with genomic data for both research and clinical purposes has exploded. While it initially cost billions of dollars to map the human genome, current machines can sequence a genome for $1000. This tremendous accomplishment has the potential to enable new scientific studies and integrate genomics into routine medical care.

In this work, we consider the genome primarily from a computational perspective. However, some key terms are needed to understand this thesis. Thus, we will begin the chapter in Section 2.2 by reviewing the relevant biology foundation before launching into the rest of the content so readers from a computer science audience will be prepared to grasp the ideas we present.

Once a sample of DNA has been provided, the first step in working with genomic data is to perform DNA sequencing, the wet-lab process by which a sequencer produces a human- and machine-readable version of the sample's genomic information. Since DNA sequencing was first developed, its speed and cost have improved dramatically. We begin Section 2.3 by discussing the original version of this technology, which is called Sanger sequencing. This approach was used for mapping the first human genome by the various research teams involved with the Human Genome Project. The second generation, short-read sequencing, has made DNA sequencing much more practical for widespread use. Finally, we will also review developments with a third generation of sequencing technology called single-molecule sequencing, which can produce much longer reads than current tools.

Since no current technology can read the entire genome at once, sequencers produce a set of reads, each of which is a short segment of DNA. Then, data processing must take place in order to reconstruct the sample's genome from the read set. We will provide an overview of some of the most important steps in the data processing pipeline in Section 2.4. The product of this phase is a set of *variant calls*, which express how the sample's genome differs from a

reference, or the species' representative genome. In this work, we are primarily concerned with the human genome.

Many steps are required to translate the output of sequencing machines to data that is useful for downstream analysis, such as an oncologist discovering that a patient has HER2+ breast cancer. First, the sequencer output must be translated from raw output to base calls, *i.e.,* the letters of DNA, which are `A`, `C`, `G`, and `T` (Section 2.5). Then, reads are typically aligned to a reference genome. Alignment (Section 2.6) is an important concept in this thesis, so we will discuss it at some length. Another major concept in this thesis is variant calling (Section 2.7), which is the process of identifying where the sample genome differs from the reference. We will consider the types of variants that can be encountered as well as techniques used to locate each type.

If the genome were a random string of three billion bases, the process of reconstructing the short reads would be relatively simple, since random short reads would be unlikely to match in multiple locations. However, it actually has both exact and near duplication throughout, which significantly complicate the data analysis phase. This thesis is primarily focused on characterizing the genome's structure (Chapter 3) and quantifying the difficulties that structure presents for variant calling, largely due to making alignment more problematic (Chapter 4). This work also provides a foundation for developing techniques to make variant calling more resilient to the genome's challenging structure (Chapter 5). Therefore, the goal of this chapter is to provide the necessary background for readers to understand the rest of the thesis.

2.2 Genomics 101

In this section, we will outline a basic foundation in biological concepts that are relevant to this work. We will refer to the terms presented here throughout this thesis. The following information is largely based on [35].

Deoxyribonucleic acid (DNA) was discovered in 1952 by Watson and Crick. Shaped like a double helix, DNA is the molecule that stores and transmits our genetic code. It is made up of four nucleotides: adenine (`A`), cytosine (`C`), guanine (`G`), and thymine (`T`). It is characterized by the principle of complentarity, which means that `A` can only bind with `T`, and `C` can only bind with `G`. Each pair of complementary nucleotides is called a *base pair*. Each of DNA's two strands is the complement of the other. We say that one strand is in the forward direction, while the other strand is in the reverse direction.

Complementarity is the foundation of DNA replication. First, a splitting enzyme shears the double helix into two strands. Then, a clipping enzyme binds a complementary free nucleotide to each position on each of the two strands. For example, if the double helix begins with the sequence `AC` on the forward strand, after the helix is split, free nucleotides `G` and `T` will be bound to the now single forward strand, in that order. The clipping enzyme is called *DNA polymerase* and is important to our discussion of DNA sequencing in Section 2.3.

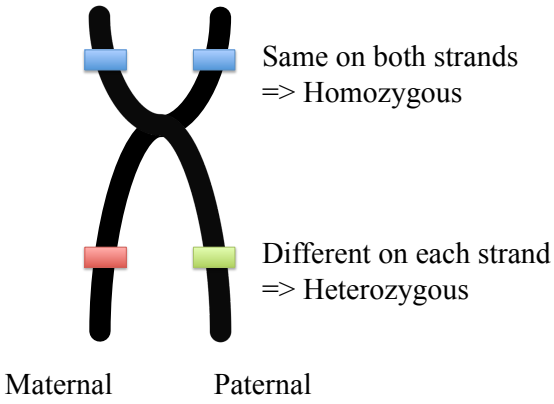


Figure 2.1: A pair of chromosomes showing both homozygosity and heterozygosity.

The human genome consists of approximately three billion base pairs. Our genome is not one long strand of DNA; rather, it consists of 23 pairs of *chromosomes*, for a total of 46. One set of chromosomes comes from each parent. Since we have two sets of chromosomes, we are *diploid* organisms; a cell with only one set of chromosomes, like a sperm or egg cell, is called *haploid*.

Our chromosomes consist of *genes*, which are defined as the units of heredity. Humans have between 20,000 and 25,000 genes. For example, a gene might govern one's hair color or eye color. To produce different outcomes, such as brown or blue eyes, genes come in different variants, or *alleles*. If each chromosome in a pair has the same allele at a given location, the person is *homozygous* at that location; otherwise, the person is *heterozygous*. We illustrate zygosity in Figure 2.1. A person's *genotype* is the set of alleles, *i.e.,* the particular DNA sequence, that makes up their genome. A person's *phenotype* is the set of their observable traits, *e.g.,* tall with brown hair, and is governed by their genotype.

This thesis is concerned with whole-genome sequencing, where the person's entire genome is sequenced. However, it is worth noting that due to the cost and complexity involved with whole-genome sequencing, much recent activity in genomics has focused on the *exome*, which is the set of all the *exons*. Exons are the segments of genes that code for protein synthesis. The remaining non-coding regions of the genes are called *introns* and are involved with regulating gene expression. The exome makes up approximately 1% of the human genome [97]; thus, the vast majority of the genome lies in the introns. As there is mounting evidence surrounding the importance of introns (*e.g.,* [106]), as well as the increasing affordability of whole-genome sequencing, we have chosen to focus our work on whole-genome analysis.

2.3 Sequencing

We will begin our discussion of working with genomic data with a high-level explanation of DNA sequencing. *DNA sequencing* is the process of starting with a DNA sample, *e.g.,* from

a patient's saliva or tumor, and reading the three billion bases that make up the patient's (or tumor's) genome. Unfortunately, currect technology is incapable of reading the entire genome at once. Therefore, all DNA sequencers produce short segments of the genome called *reads*. This process could be compared to starting with a picture and producing a set of puzzle pieces corresponding to that picture. In Section 2.4, we will discuss what to do with the resulting puzzle pieces, *i.e.,* reads.

DNA sequencing has been in development over several decades. The first major sequencing technology is called *Sanger sequencing* and is an accurate but expensive and slow approach to producing long reads of up to one thousand base pairs. The second generation, most notably from Illumina,[1] is called *short-read sequencing* and produces a much higher quantity of reads for a significantly lower price. The advantage in price, however, comes with drawbacks including an increased error rate and shorter read length of only 100-250 base pairs. The third and most recent type of sequencing, called *long-read sequencing*, is an emerging area that has the potential to greatly simplify working with genomic data. As the name implies, the reads produced by these sequencers are at least an order of magnitude longer than those produced by second-generation sequencers; nevertheless, the throughput is still too low and the error rate too high for these techniques to be widely used for variant calling. However, they have already started to enable assembly to be more accurate and complete [123, 102].

In this thesis, we will focus primarily on second-generation sequencing, and particularly on the technology developed by Illumina. We made this decision because we foresee that Illumina will maintain its dominance in the near- to medium-term future as scientists tackle the still considerable hurdles to third-generation sequencing. The vast majority of genomes presently being sequenced for research and clinical purposes are being sequenced by Illumina machines. Also, most of the currently-available bioinformatics software, which we will review in Sections 2.5 – 2.7, was designed for the particular characteristics of short reads (and often for Illumina specifically). Therefore, we will focus our discussion in this section and the rest of this thesis on second-generation sequencing from Illumina. We will still briefly explain Sanger and long-read sequencing in order to provide additional context, both historical and future, respectively.

Sanger Sequencing

Sanger sequencing was the original sequencing technology and was used for the Human Genome Project [50]. The output of the Human Genome Project, a composite produced from the genomes of several volunteers, is called the *reference genome*, and we will refer to it often. Since all humans share about 99.9% of our genome, the map produced is representative of all human genomes. The Human Genome Project was a collaboration among 20 laboratories around the world, together constituting the International Human Genome Sequencing Consortium. The project was in progress from 1990 to 2003, finishing two years

[1] `http://www.illumina.com/`

ahead of schedule, and cost $2.7 billion, below its budget of $3 billion. The first draft of the genome was published in 2000, with about 90% of the genome sequenced [65]. The effort was considered complete in 2003, although a small amount of the genome is inaccessible with today's sequencing technology. As the technology continues to improve, our map, *i.e.,* the reference genome, is periodically updated.

Sanger sequencing is based on processing the input sample DNA to produce redundant, single-stranded segments of different lengths [111, 115]. For example, the first segment might be `CTGGAAG`, while another is `CTGGAAGC`, and the next is `CTGGAAGT`; note that each successive segment is one base longer than the previous one. A fluorescent label is attached to the terminal end of each segment, where the color of the label is determined by the final base. In our example, the first segment ends with `G`, so it might get a yellow label, while the next segment ends with `C`, so it gets a blue label. Then, the segments are sorted according to their lengths, and the sequencer reads the labels in order to reproduce the sequence.

Sanger sequencing can produce reads of several hundred to one thousand bases, and it does so with error rates of less than 0.1% [128]. The process is expensive and slow, however, with limited throughput. To provide a sense of the cost, when the Human Genome Project transitioned in 2007 from Sanger to second-generation technology, its teams were paying over $100 per megabase of sequence, while the current cost of second-generation reads is less than $0.10 per megabase [134]. Sanger sequencing is therefore impractical for widespread use, and second-generation approaches, which we discuss in the following section, have supplanted Sanger sequencing for most common applications. Nevertheless, due to its high accuracy, Sanger sequencing is sometimes still used for selectively validating small subsets of sequence.

Second-Generation Sequencing

The dominant technology for second-generation sequencing is called *sequencing by synthesis* (SBS) and is provided by Illumina [8]. In what follows, we will focus on Illumina's technology since it is the most widely utilized. Other companies providing second-generation sequencing via different approaches include Ion Torrent and SOLiD from Life Technologies[2] and 454 from Roche.[3]

In Figure 2.2, we give a high-level illustration of how sequencing works. The sequencing process begins with a sample of DNA. The original sample is replicated and then divided into short segments, called *fragments*. A common size for fragments is about 700 bp. The fragments are bound to adaptors on the flowcell, which is the surface of the sequencer, and each fragment is amplified to create a cluster of many identical copies [49]. Many such clusters form on the flowcell, and each cluster is read independently by the image sensor.

The sequencer reads a fragment one base at a time. Recall from Section 2.2 that each base has a complementary base: `A` and `T` are complements, and so are `C` and `G`. During each cycle, reagents are added to the flowcell, and the reagents compete to bind to the strand.

[2] `http://www.lifetechnologies.com/`

[3] `http://www.454.com/`

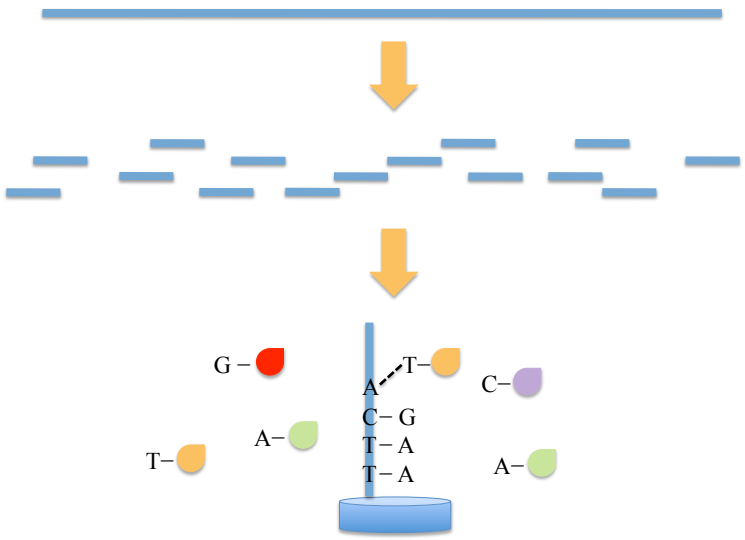


Figure 2.2: Second generation sequencing. To read a DNA fragment, polymerase binds the complementary base to the fragment. The sequencer can infer which base was attached based on the color of the light emitted.

Only a complementary base can attach at the current position on the strand; in Figure 2.2, the current base is an `A`, so only a `T` can attach to it. In the figure, an orange fluorescent label is connected to each `T`. When DNA polymerase joins the `T` to the strand, the label is released, and the image sensor can detect its color. Thus, the sequencer determines that a `T` was attached, and it infers that the base just read is an `A`.

Though for simplicity the figure depicts a single fragment being read, each cluster's many identical fragments are all read in a synchronous fashion. That is, if the current base is an `A`, free `T` reagents will bind to the current position on each fragment. At some point, the synchrony degrades, so the sequencer's read length is bounded. Currently, Illumina reads are limited to between 100-250 bp.

Compared to Sanger sequencing, second-generation sequencing costs less and has higher throughput. However, the error rate is higher; it was initially around 2% but is now less than 1% [128]. That is, fewer than 1% of bases will be sequenced incorrectly. An additional drawback is the reduced read length. An important feature of these reads that somewhat compensates for their shorter length is that they are usually read in pairs. That is, since due to synchronization breakdown the sequencer cannot read the entire fragment, the sequencer reads both ends of the fragment to produce what are called *paired-end reads*.

In part (a) of Figure 2.3, we show how the sequencer produces a pair of reads [48]. The DNA fragment, shown in blue, has an adapter attached to each end. The reading takes place in phases. First, the fragment attaches to the flowcell via one of the adapters; in this example, the green adapter attaches first. The sequencer reads the fragment up to its maximum read length in the forward direction via the process illustrated in Figure 2.2.

Next, the sequencing product is washed away, and the fragment binds to the flowcell via its other adapter as well, creating a bridge. The first adapter releases, leaving the fragment attached by only its second adapter; in this example, the orange adapter attaches second. Then, the sequencer reads the fragment in the reverse direction, again up until its maximum read length.

Figure 2.3(b) shows the product of paired-end sequencing. The image depicts the two reads, each of which is much shorter than the fragment. Typical values for the read length and fragment length are 100 bp and 700 bp, respectively. The remaining DNA in the center of the fragment that the sequencer could not read is called the *insert*; given our values of read length and fragment length, the length of the remainder, called the *insert size*, would be 500 bp. Note that the fragment length is not fixed for a sample; rather, the length can vary between fragments. Thus, we can only estimate the insert size. Nevertheless, this partial knowledge is quite useful. Since the reads in a pair were produced from a single fragment, we know approximately how far apart they are located in the sample DNA. This helps to resolve some ambiguity when it comes to reconstructing the sample genome. Even with the valuable information provided by paired-end reads, however, some ambiguity persists. This uncertainty is in fact the subject of this thesis, and we will discuss it at length in later chapters.

An interesting concept related to paired-end reads is *mate-pair sequencing*. The goal of mate-pair sequencing is to produce a pair of reads that is separated by much more than the typical fragment length, *e.g.,* with a separation of several thousand bases. Very long DNA segments are achieved through a process called DNA circularization, which is outside the scope of this thesis. Though mate-pair sequencing is more expensive than typical paired-end sequencing, the benefit is that it can help resolve some, but not all, of the ambiguity that persists with typical fragment sizes. Mate-pair libraries are widely used for *de novo* genome assembly (described in Section 2.7) but are not commonly used for variant detection due to their high cost.

Long-Read Technology

Long-read technology is an emerging area. Instead of pairs of reads that are around 100-200 bp, long-read sequencers can produce single reads that are between 1,000 and 10,000 bp. Though we do not focus on this approach in this thesis, since it is not yet the dominant technology, we include a high-level description of long-read technology for completeness in our discussion of sequencing, as well as because we will refer to it in Section 3.2.

Two main companies have emerged as the leaders in this space. In what follows, we will briefly introduce the technology employed by each of these companies. Both are single-molecule technologies; instead of jointly sequencing many identical fragments per cluster as Illumina does, each unit of sequencing is a single molecule of DNA. Like Illumina, many units are sequenced in parallel. Long-read sequencers are able to achieve much greater read lengths since they do not rely on synchronous reading, which gets out of phase after 100-250

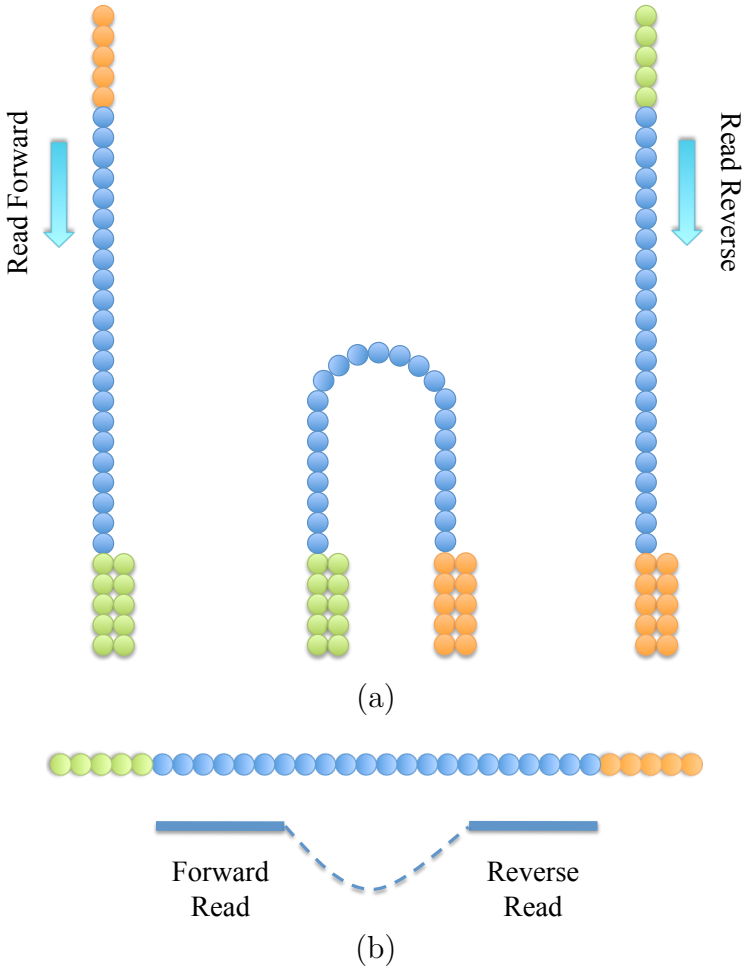


Figure 2.3: Paired-end sequencing. The sequencer reads both ends of a fragment to produce a pair of reads.

bases. We will conclude the section by enumerating the advantages and disadvantages of this space.

Pacific Biosciences,[4] commonly known as PacBio, provides one approach to long-read technology [112]. In their approach, each individual molecule of DNA is denatured to be single-stranded and then isolated into a small hole called a zero-mode waveguide (ZMW). Like Illumina, the sequencers rely on an image sensor that detects colored fluorescent labels attached to reagents. Each ZMW is initialized with its own molecule of DNA polymerase, *i.e.,* the enzyme that constructs DNA, which we discussed in Section 2.2. The reading cycle begins when reagents are added to the ZMW. As the DNA polymerase binds a complementary base to the current base on the strand, the new base's label is detected by the image sensor. For example, if the current base on the strand is a `G`, the DNA polymerase will attach

[4]http://www.pacificbiosciences.com/

a `C` to the strand. If the `C` is characterized by blue labels, the image sensor will register a blue light and infer that the base just read was a `G`.

Rather than using a light-based approach like Illumina and PacBio, Oxford Nanopore's technique is based on detecting fluctuations in electic current [6]. They create a substrate with very high resistance and then insert nanopores, each of which is only a few nanometers wide, into its surface. Then, a voltage is applied to the substrate. Since the substrate has high resistance, current can only flow through the nanopores. An electrode is attached to each nanopore to measure the current flowing through it. If anything flows through the nanopore, it will cause a disruption in the current; measuring that disruption gives information about what passed through the nanopore.

During sequencing, DNA molecules are brought near the substrate. An enzyme corresponding to each DNA molecule helps it attach to a nanopore and guides the molecule through the nanopore, one base at a time. The enzyme also unzips the DNA molecule so it can flow through the nanopore as a single strand. Then, as each base flows through the nanopore, it alters the nanopore's baseline current in a characteristic way, allowing the nanopore's electrode to detect the current base [95].

Long-read technology, with its greater read lengths, has the potential to make reconstructing reads to obtain the sample's complete genome much easier than with second-generation technology. Another advantage of long-read technology is the ease of sample preparation; since these are single-molecule technologies, no sample amplification is necessary, which is advantageous because amplification can introduce biases that complicate downstream analysis.

Despite its appeal, however, long-read technology currently suffers from some disadvantages. First, the throughput is low, so the sequencers cannot produce reads of sufficiently high coverage and low cost for this technology to be truly market-ready. Second, the reads produced have much higher error rates than second-generation reads. Much of the existing bioinformatics software is designed with many assumptions about second-generation reads, especially their very low error rates. Therefore, new software must be designed to cope with the specific characteristics of long reads, should they eventually become economically viable.

2.4 Data Processing Overview

Now that we have covered how sequencing works, we will discuss what comes next, *i.e.,* how we recover a sample's full genome from the output of a sequencer. Since second-generation sequencing technology, and Illumina in particular, is by far the most popular technology dominating the market today, we will focus on algorithms and software that are specifically designed for the unique characteristics and error types of Illumina sequencer output.

The process of reconstructing the genome from sequencer output is called *variant calling.* It is so named because its aim is to find variants between the sample genome and the reference genome, or representative human genome. Since humans share 99.9% of our DNA, only one in 1,000 bases is expected to be a variant, or about three million bases total.

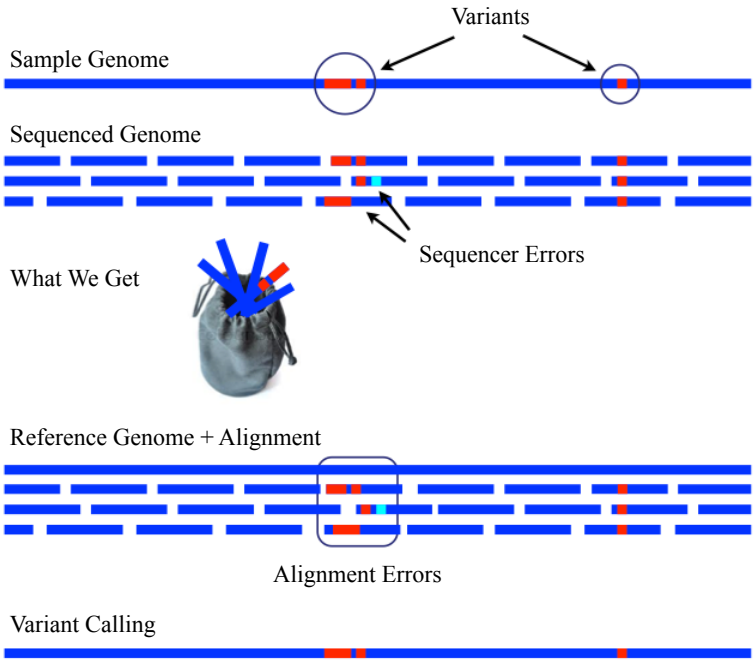


Figure 2.4: End-to-end pipeline, from DNA to variant calls. Used with permission from Ameet Talwalkar.

In Figure 2.4, we illustrate the steps involved in variant calling. The top line of the figure shows the sample genome, with its variants in red. Next, the figure depicts the reads produced by the sequencer; some of the reads contain the variants, while others contain sequencer errors. The reads are shown in a bag because they are entirely unordered when they are submitted to the variant calling pipeline; thus, part of the problem of processing them is determining where in the genome they likely originated. Most variant calling pipelines start with alignment, which is the problem of choosing the location in the genome against which the read matches best. Once all reads have been aligned, and some preprocessing has been applied, variant calling tools infer the sample genome from the reads.

A subtle point occurs near the bottom of the figure, in the "Reference Genome + Alignment" band. Note the box labeled "Alignment Errors." If we compare this box to the reads in the "Sequenced Genome" band, we can see that the second read, which has both a red variant and a light blue sequencing error, is shifted to the right; it should be in line with the read above it. Alignment errors like this one are one of the main factors that complicate variant calling; in fact, we will discuss this extensively in Chapters 4 and 5.

Several file formats are involved in this pipeline, which we summarize in Figure 2.5. The reads produced by the sequencer start out as raw intensity images; the process of *base calling* converts them to text, containing both read sequence and quality scores, which is stored in the FASTQ format. FASTQ reads are aligned and output as SAM/BAM files, which are preprocessed for variant calling. These analysis-ready SAM/BAM files are then fed into

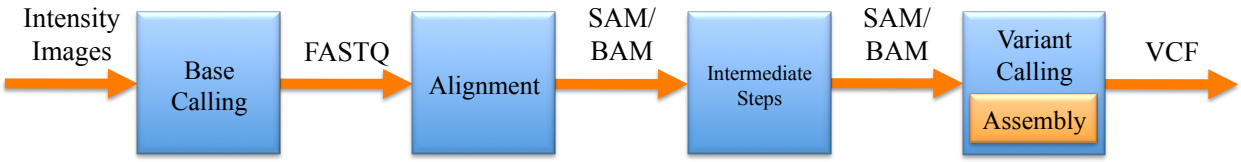


Figure 2.5: File format flowchart, from raw images from the sequencer to analysis-ready variants.

variant calling, which sometimes involves a sequence assembly component. After variants are produced, they can be analyzed for myriad interesting medical and scientific applications.

In the remainder of this chapter, we will discuss each of the steps in this pipeline, with a level of detail proportionate to each step's relevance to this thesis. In Section 2.5, we will briefly define base calling. Then, in Section 2.6, we will more thoroughly discuss alignment, as it emerges throughout this thesis as an important problem. Finally, we will provide some background on variant calling in Section 2.7, including the intermediate steps that precede it, as well as its occasional reliance on assembly.

2.5 Base Calling

Base calling is the process of converting images produced by the sequencer of the fluorescent labels attached to reagents to letters representing the four DNA bases. When it comes off the sequencer, each read is initially a series of intensity images produced by the sequencer's image sensors. Base calling converts these intensity images from the sequencer to the FASTQ file format. In the FASTQ format, each read is represented by an ID, the read sequence, and a quality string whose length is the same as the length of the read. There is one quality character per base in the read sequence; these are called *base quality scores*, and they represent the base caller's confidence in its assessment. Figure 2.5 shows how base calling fits into the analysis pipeline.

An excellent review paper [68] provides illustrations of common problematic phenomena that take place in Illumina sequencers that make base calling more difficult. For example, crosstalk occurs because there is some overlap in the intensity signature of the fluorescent labels associated with each of the four bases. Ideally, the intensities would be nicely separated so they could be distinguished easily; however, in practice, their slight overlap generates confusion for base calling.

Base calling algorithms aim to make correct inferences despite these sources of ambiguity. To do so, they leverage machine learning algorithms of varying sophistication, from linear classifiers like support vector machines (SVMs) to unsupervised models with many parameters. These models incorporate sequencer-specific knowledge, so they explicitly account for the various sources of uncertainty. Example base calling algorithms are BayesCall [56], Ibis [59], and All Your Base [91].

2.6 Alignment

We begin by providing an overview of the short-read alignment problem and some general information about how alignment algorithms, or aligners, work. Then, we briefly discuss each of the three categories into which these algorithms fall, namely, algorithms that hash the reads, algorithms that hash the reference genome, and algorithms based on the Burrows-Wheeler transform (BWT). The reason we present alignment in detail is twofold: first, this work was born out of our experiences gained by developing an alignment algorithm; and second, this work is largely focused on the difficulties presented during the alignment stage by the redundant structure of the genome and our efforts to ameliorate this issue.

Overview

Starting with a read, alignment is the problem of determining where in the genome it most likely originated. Alignment uses the reference genome, which we defined in our description of Sanger sequencing in Section 2.3. Many reads will match the reference exactly, while others will differ by up to a few bases, due to either sequencing errors or true variants. Some reads will differ quite a bit more due to larger genomic changes, which we discuss further in Section 2.7.

As Figure 2.5 shows, the input to the alignment step is reads, stored in FASTQ files, and the reference genome, stored in a FASTA file. The results of alignment are written in the text-based Sequence Alignment/Map (SAM) format, or in the binary BAM format once compressed [37]. The SAM format consists of a header and one entry per read, where each entry consists of the read's ID, its mapped location, the read itself, as well as a few other quantities of interest by which the aligner may give extra information, *e.g.,* the number of edits between the read and the reference.

Recall from the discussion of second-generation sequencing in Section 2.3, and especially Figure 2.3, that reads come in pairs. Tools may align each read individually (single-end alignment), or they may align the pair jointly (paired-end alignment). Paired-end alignment is almost always preferable due to the extra information it provides, which can be extremely useful for dealing with ambiguity. For example, say the first read in a pair maps to many different locations equally well, while the second read maps to only one location. Even though the first read cannot be unambiguously aligned on its own, using the information from its paired read, it can be mapped uniquely, *i.e.,* to the candidate near the second read's single location. Thus, while most aligners provide both single-end and paired-end alignment modes, paired-end is almost always used.

The problem of alignment has been well-studied for several years, and many tools have been developed. The tools tend to use algorithms that fall into three categories, each of which we discuss below.

Hashing the Reads

Many aligners utilize hashing in order to find candidate locations against which a read may match. After a candidate is found, the aligner compares the read to the reference genome at the location to compute how well it matches. These algorithms are often called *seed-and-extend* algorithms, where *seed* refers to using a substring, or seed, of the read to find possible matches, and *extend* refers to comparing the full read to the reference genome. Many hash-based aligners use an algorithm such as Smith-Waterman to efficiently execute the extend phase [121].

One group of hash-based aligners relies on hashing the reads and then scanning the reference genome. As each substring of the reference genome is scanned, the aligner consults the index for reads that match against the current substring. Instead of hashing the entire read, only a portion of the read is hashed; that portion is called a seed. A portion is hashed, rather than the whole, in order to allow a read to align against the reference even if it differs due to a sequencing error or true variant. In the case of such a read, if it has a seed that matches the reference exactly, candidate locations may still be suggested by the aligner. Aligners that take this approach, ordered by date of publication, include ELAND [24], RMAP [120], MAQ [75], ZOOM [83, 143], SeqMap [54], CloudBurst [113], and SHRiMP [110, 26].

Few modern aligners use the technique of hashing the reads. The advantage of this approach is that the memory footprint is adjustable. The algorithm designer and the user, if adequate parameters are provided, can decide how many seeds should be hashed. If fewer seeds are chosen, less memory is required, at the cost of more hash misses and potential reductions in the number of reads that can be aligned. For example, a read that mismatches the reference in all seeds that are hashed but matches the reference in a location that was skipped will be unmappable with the low-memory hashing approach, whereas if the memory were increased, the read could be mapped. Similarly, if more seeds are selected, the hash will require more memory but will result in a higher fraction of the reads being mapped. This approach quickly becomes unwieldy when attempting to support more mismatches.

Hashing the Reference

A second category of aligners hash the reference instead of the reads. Like aligners that hash the reads, these aligners employ the "seed-and-extend" approach of generating a list of candidate locations and then comparing the read against some or all of the candidates to find the matching location. Aligners in this category create an index of the genome and look up seeds from the read to get candidate locations for where reads could align. Many different aligners of this type exist, including, by date of publication, BLAST [5], BLAT [57], SOAP [79], PASS [16], MOM [29], ProbeMatch [58], NovoAlign [100], PerM [18], BFAST [45], mrFAST [4], mrsFAST [39], SNAP [139], Stampy [84], Isaac [104], BWA-MEM [72], and Mosaik [69].

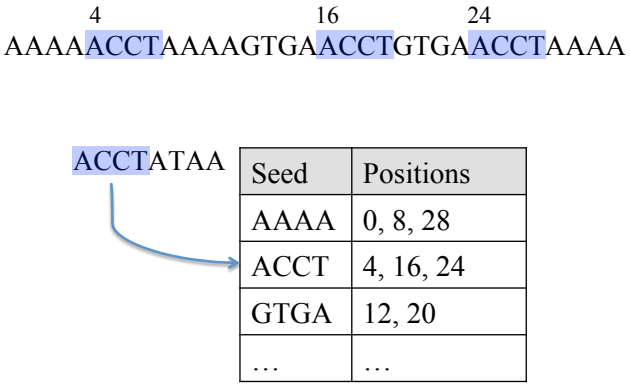


Figure 2.6: Aligners that hash the reference leverage an index of short seeds and their positions in the genome.

We illustrate this approach in Figure 2.6, where we attempt to align the read `ACCTATAA`. We begin by looking up the seed `ACCT` in the index, which we precomputed by hashing each seed from the reference along with a list of positions at which that seed occurs. The index indicates that the seed appears in the reference at positions 4, 16, and 24. Then, we compare the full read against the reference, anchored by the seed at each of these positions, to find the best match. In this case, the read matches positions 4 and 24 equally well, with one mismatch.

One advantage of hashing the reference instead of hashing the reads is that the index must only be recomputed whenever the reference genome changes, which is infrequent (*i.e.,* once every few years). In contrast, when the reads are hashed, this must be done for each new dataset, since each dataset is unique. Another advantage of hashing the reference is that only one index is necessary per seed size, regardless of how many mismatches the aligner's developers wish to support. These aligners are often quite fast as well, especially SNAP and BWA-MEM. A disadvantage of hashing the reference is that it requires a lot of memory. For example, the SNAP aligner's index of the full human reference genome hg19 is about 32 GB. However, as memory prices are declining rapidly, this is not a significant obstacle.

Burrows-Wheeler Transform

An alternate approach to alignment relies on the Burrows-Wheeler transform (BWT) [14]. Some aligners use a related index based on the BWT, which is called the FM index [31]. The basic idea of the BWT, which is a fairly complicated transformation, is to compress a group of substrings using a trie structure [60] so that similar substrings are stored together. Aligners in this category perform the BWT on all substrings of the reference genome. Then, in order to match a read against the reference, the aligner traverses the trie, backtracking whenever a mismatch is encountered. The most well-known and widely-used of the BWT-based aligners is BWA [74], but others include SOAP2 [80], Bowtie [67], Bowtie2 [66], and

GEM [88].

The main advantage of the BWT-based aligners is their low memory footprint, due to the compressed nature of their method of storing the reference. However, the tradeoff is that as more mismatches are supported, more backtracking will result, so speed will suffer.

2.7 Variant Calling

Variant calling is the problem of deducing a sample's variants with respect to the reference genome based on a set of reads that have been aligned to the reference. As Figure 2.5 shows, the input to variant calling is a SAM or BAM file containing the aligned and possibly preprocessed reads, and the output is a variant call format (VCF) file [127]. A VCF file contains a header as well as one entry per variant, each of which contains the position and genotype of the variant (including its zygosity, defined in Section 2.2) and other information from the caller such as a variant quality score.

We begin by discussing several intermediate steps that are often performed before attempting variant calling on aligned reads. Then, we describe variant calling techniques and tools, first for SNPs and indels and then for structural variants. Although this section focuses on alignment-based variant calling, assembly is sometimes involved in variant calling in addition to or instead of alignment. Thus, we end the section by defining assembly and explaining how it can be used as a part of variant calling.

Intermediate Steps

Before aligned reads can be analyzed, some intermediate steps are necessary. One reason is the occurrence of sequencer-specific systematic errors; some have been described for the Illumina sequencer [92]. They can occur due to factors like uneven amplification, leading some segments to be overrepresented in the sample. Another contributing factor is the presence of certain motifs like high `GC` composition that can make the sequencers more likely to read the sequence incorrectly. Errors like these, as well as others, can complicate downstream processing. Therefore, the following intermediate steps are often employed before performing variant calling on aligned reads in order to improve accuracy.

First, reads must be sorted by the position in the reference to which they aligned. Usually, when an aligner produces an alignment, it outputs the reads in the order in which it encountered them, which in turn corresponds to the (random) order in which they were sequenced. However, many downstream tools prefer to operate on reads by their location; therefore, sorting is necessary.

Duplicate removal is the process of marking reads that are likely duplicates due to biases in the library preparation phase. As the authors report in [99], duplicate reads are identified when all reads in a group are aligned to the same position with the same orientation (*i.e.,* all forward or all reverse-complement). To avoid biasing downstream tools, *e.g.,* by erroneously

inflating the coverage at the aligned position, all but one member of the group of duplicates are discarded; the read with the highest quality is retained.

Another step is local realignment, which targets reads that have been misaligned due to indels. For example, if there has been a small insertion, reads occurring near the insertion may have been aligned slightly incorrectly, posing difficulties for variant calling. In local realignment, the algorithm identifies where realignment may be necessary and then improves the alignment of reads in the vicinity, *e.g.,* by generating alternative sequences to represent the reference with the possible indel embedded in them. The GATK paper [27] has a compelling illustration showing how local realignment can greatly improve the consistency of the reads around an indel.

Another important prerequisite for variant calling is base quality score recalibration (BQSR). Recall that base calls are associated with a quality score representing the confidence that the base call is correct. After the reads' alignments have been adjusted, if necessary, the goal of this phase is to check, for each quality score, whether the number of mismatches corresponds to the error probability associated with that quality score. If not, the tool will adjust the quality score so the error probability is more consistent with the data.

Frameworks like the GATK [27], which we will discuss in more detail when we introduce variant calling, and ADAM [99] provide implementations of these four steps. The GATK's implementation is available via the Picard[5] suite. In addition, the SNAP aligner [139] includes sorting and duplicate removal.

Calling SNPs and Indels

The simplest variants are SNPs and indels. A single nucleotide polymorphism, or *SNP*, is a single-base difference between the sample and reference obtained via substitution of one base for another.

Figure 2.7 illustrates a SNP; we obtained the image using IGV [109, 129]. The band at the top of the figure consisting of consecutive vertical bars represents the coverage, or number of reads, overlapping each position. Gray bars indicate that the reads match the reference, while the blue bar near the middle of the band indicates that all the reads overlapping the reference at that position differ from the reference. The horizontal bars making up the rest of the figure represent reads that have been aligned to this section of the reference. Gray reads are aligned in the forward direction, while white reads represent reads whose reverse complement has been aligned. Where the reads are solid gray or white, they match the reference. However, underneath the blue bar in the top band, we see that all the reads indicate that the sample has a `C`, while the reference has a different base. This `C` is a SNP, since the sample differs from the reference in one base via a substitution.

Indel is the shorthand we use to refer to an insertion or deletion between the sample and the reference. In this section, we use indel for short insertions or deletions that are 50 bases

[5] http://broadinstitute.github.io/picard/

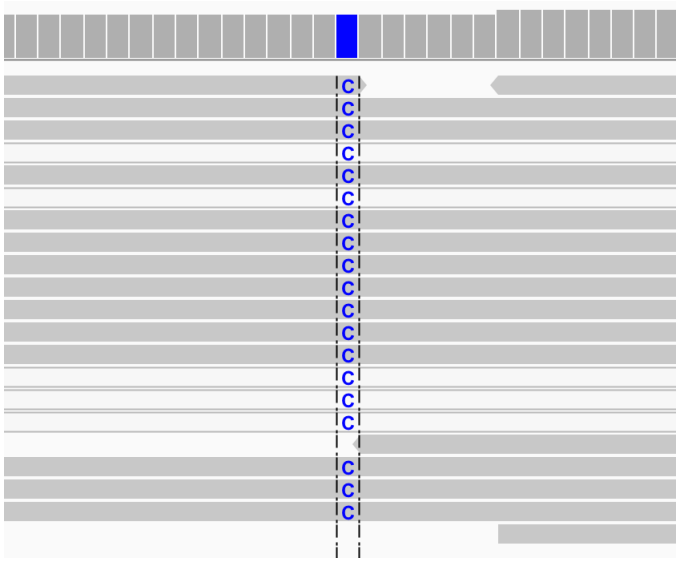


Figure 2.7: Reads showing a single nucleotide polymorphism (SNP) in the sample DNA.

or fewer. Longer indels fall under the category of structural variants, which we will define later in this section. For example, if the reference in a given location is `ACTG`, a one-base insertion of `C` between the sample and the reference would result in the sample genome being `ACCTG` at the corresponding location. Likewise, a one-base deletion of `T` would yield `ACG` for the sample genome.

Early variant calling algorithms used simple rules to call variants [98]. One rule that was used was an 80/20 rule, where if 20-80% of the reads indicated one allele and the remaining reads indicated another, the tool would call a heterozygous SNP at that location. Otherwise, the caller would consider the location to be homozygous, either for the reference base or for the base indicated by the reads.

For example, say at one location the reference is `C`, and there are 10 overlapping reads, with 3 of the reads indicating an `A`, while the remaining 7 reads indicate a `C`. The tool would call the location as a heterozygous SNP, with the sample having both `A` and `C` at that location. Another sample might also have 10 reads overlapping this location; however, in the second sample, one read implies a `G`, while the other nine imply a `C`. The caller would infer that the second sample is homozygous at this location, matching the reference. The one read that indicates a `G` may be misaligned, or it may contain a sequencing error.

Modern, state-of-the-art variant callers employ a probabilistic framework rather than simple rules in order to incorporate uncertainty from the reads [98]. Assuming that the genotype likelihood $P(X|G)$ (*i.e.,* the probability of the data given the genotype) can be computed, these tools use Bayes' Rule to obtain the posterior $P(G|X)$, where X is the data and G is the sample genotype. Then, they select the genotype that maximizes the posterior.

Variant calling has been studied for several years, resulting in a wide range of solutions

of varying complexity and effectiveness. Early, simple tools for variant calling include MAQ [75], mpileup [71], and SOAPsnp [78]. By far the most widely-used variant caller, though, is the Genome Analysis Toolkit (GATK) [27], which also has functionality for improving data quality (discussed above). The GATK is a sophisticated and computationally expensive tool that aims to reduce false positive calls by incorporating prior knowledge from variant databases about probable locations for SNPs and discounting any calls that lie outside these likely locations. That is, from a Bayesian perspective, GATK incorporates an informative prior $p(G)$.

Other more recent variant callers include FreeBayes [32], SOAPindel [81], and ISAAC [104]. However, these tools have yet to displace GATK as the state-of-the-art. Practitioners are hesitant to make changes to their pipelines, since the pipelines often require regulatory approval if they are used in a clinical setting. Therefore, while research prototypes may embody useful ideas, they have a great deal of difficulty gaining adoption.

In this work, we use mpileup and GATK. We chose these tools because we wanted to explore how both simple and sophisticated tools would succeed or stumble in the similar regions, and both mpileup and GATK are well-known in their respective categories. We will discuss our usage of these tools in Chapters 4 and 5.

Detecting Structural Variation

We continue our description of variant calling by defining structural variants and briefly discussing detection techniques. Structural variants are an important type of variation, so part of our motivation for including this discussion is for completeness in our presentation of variation and its calling. Structural variants are also relevant to our demonstration that the similar regions can be used to improve variant calling in Chapter 5. However, we mostly focus on how the similar regions impact SNP and indel calling, so this discussion will be brief compared to that one.

A *structural variant* (SV), compared to SNPs and indels, is a large change between the reference and the sample. Figure 2.8 illustrates several types of structural variants. We begin with *deletions* and *insertions*, which of course are the same type of variant as indels. The difference is that in order to be classified as a structural variant, the insertion/deletion must be over 50 bases long. Another type is a *duplication*, in which a segment of the reference is repeated in another location. The repeat could appear adjacent to the original sequence or elsewhere. Copy number variation (CNV) is related to duplication, as a CNV refers to when a sequence is repeated one or more times. Another type of SV is *inversion*, in which a segment of the reference is flipped and then re-inserted into the sample genome. For example, given a reference sequence of `ACG`, the inverted version would be `GCA`. A final type of structural variant is called a *translocation*, where two segments of the reference are swapped; in some diseases, portions of one chromosome are swapped with portions of another chromosome.

Structural variant discovery is widely considered to be much more difficult than calling SNPs and short indels; consequently, the error rates are often significantly higher. Some

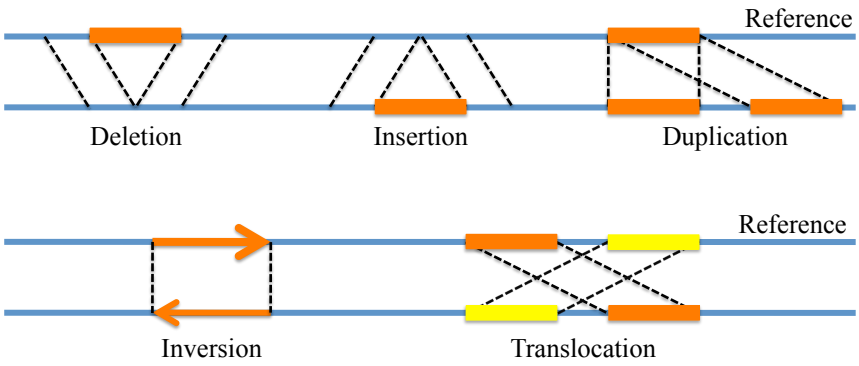


Figure 2.8: Types of structural variants.

well-known tools for SV detection, described in [3], include Variation Hunter [46, 47], Pindel [138], and BreakDancer [17].

One way to detect structural variants is to consider reads that align improperly, given the expectation for how paired reads normally align. For intuition, consider that a large deletion has occurred in the sample. As Figure 2.8 shows, the flanking sequence in the reference will be adjacent in the sample. Thus, if a fragment is sequenced from the newly connected flanks, the reads will map further apart than expected in the reference, since they will be on either side of the deleted sequence that the reference retains. Similarly, in the case of a translocation, reads in a pair could map to different chromosomes. For example, if there has been a translocation between chromosomes 5 and 10, a read pair spanning the newly spliced segments of chromosomes 5 and 10 would map to different chromosomes, even though the reads belong to a single pair.

Assembly

Assembly is the problem of joining short reads together into longer contiguous sequences, or contigs, and is usually done by focusing on the reads themselves instead of leveraging a reference genome to help. A review paper covering the different approaches to assembly and many of the assemblers in use is available [94]. In this thesis, we focus on alignment-based approaches to variant calling, some of which use local assembly.

Like alignment, assembly is complicated by the presence of near and exact duplication in the genome. Though assemblers rely on the assumption that similar reads originated from the same area of the genome and should therefore be considered jointly, this assumption often fails because of the presence of repetitive sequence. As a result, the assembler may incorrectly try to stitch reads together that belong in different regions.

Three main approaches to assembly exist [94]. The first and most primitive is a greedy approach that makes the best decision as each new read is added, rather than considering multiple reads to optimize the assembly. This approach was used for the earliest assemblers,

but it has fallen out of favor since it makes mostly local decisions and cannot take advantage of longer-range information from paired and mate-pair reads.

The second and slightly more popular approach is based on creating overlap graphs out of the reads, where a node corresponds to each read, and an edge connects any two reads that overlap; *e.g.,* the reads `ACTT` and `TTAG` would be connected by an edge because they share the sequence `TT`. Unlike greedy assemblers, overlap graph assemblers can exploit information from paired and mate-pair reads due to the graph structure. Though this approach is not widely used due to the computational expense of constructing the graph, SGA [118] is an example of an overlap graph assembler.

The third and most popular approach to assembly is based on the de Bruijn graph [13]. In a de Bruijn graph, the edges represent k-mers, and edges connect any nodes whose k-mers overlap by $k-1$ bases. Note that k is less than the read length. Then, reads are represented as paths through the graph. For example, consider the read `ACTTGGA`. If $k = 5$, we would create three nodes from this read, namely `ACTTG`, `CTTGG`, and `TTGGA`. An edge would connect the first and second nodes, and another edge would connect the second and third nodes. Then, our read would consist of the path through the three nodes. The disadvantage of this approach is that it relies on exact overlap among k-mers, so error correction is an important prerequisite.

In *de novo* assembly, a reference genome is not used; rather, assemblers leverage information solely from the reads to put them together. *De novo* assembly is the only option for analyzing a genome that has not yet been mapped; since no reference is available, alignment is impossible. Small genomes can also be assembled. However, though recent assemblers perform better than their predecessors, *de novo* assembly is generally considered too expensive to be practically applicable for routinely processing large genomes such as the human genome. *De novo* assemblers, in order by publication date, include Velvet [142, 141], ALLPATHS [15], ABySS [119], SOAPdenovo [77], ALLPATHS-LG [34], SOAPdenovo2 [85], Cortex [52], Discovar [133], and FermiKit [73].

In contrast, local or targeted assembly relies on aligning reads to the reference genome first, putting the reads together only in certain regions of interest. Targeted assemblers include GATK's HaplotypeCaller [27], Telescoper [12], Platypus [105], and Scalpel [96]. Targeted assembly is used to augment alignment-based variant calling in some situations. Telescoper is used to assemble reads in the telomeres, which are very challenging for variant calling due to their repeat structure. In addition, the GATK's HaplotypeCaller utilizes local assembly to achieve higher accuracy in variant-dense regions.

Another reason to use local assembly is for genotyping structural variants [94]. Suppose there has been a large, novel insertion of hundreds of bases, *i.e.,* well over a read length, in a sample genome. The reads near the edges of the insertion may partially map to the reference, but the reads from the insertion itself will fail to map. Therefore, the only way to recover the sequence of the insertion is to assemble the reads and then use the pairs of reads that map to the insertion's flank to anchor the insertion.

2.8 Conclusion

Genomics has broad potential to transform research and medicine. In this chapter, we provided more detail about the specifics of working with full-genome data. From the time that a patient provides a sample, the sample must be sequenced, and then the sequencer output must be analyzed via a data processing pipeline until a set of variant calls is produced. Variants are where a sample differs from the species' representative genome, describing how the sample is unique. The variant calls can then be used, for example, to reason about a patient's disease in order to recommend the most suitable treatments.

We started by defining some important genomics terms and by discussing the technologies used to sequence a DNA sample. In the time since DNA sequencing entered the scene, it has changed dramatically; therefore, we covered the historical techniques as well as current and future ones in order to provide a context as well as give a sense for the technology's trajectory. The first technology, Sanger sequencing, was used for the Human Genome Project and is still used to provide validation data in certain targeted settings. In response to its high cost and low throughput, second-generation, short-read sequencers were developed, with the most prominent sequencers being made by Illumina. In the future, single-molecule technologies may take over, with their much longer reads potentially further simplifying data analysis.

In this chapter, we focused on second-generation technology and the algorithms required to transform the raw sequencer output into variant calls. The process starts with base calling, where reads of the letters `A`, `C`, `T`, and `G` are produced from the sequencers' image output. Then, reads are aligned to the reference genome, or matched up against the subsequence to which they most likely correspond. After some intermediate steps to improve accuracy, variant calling algorithms use the information in all the reads covering a particular location to infer the sample's genome at that point, including whether or not the sample differs from the reference.

Despite much work on all aspects of this pipeline, challenges to accuracy remain due to the complicated structure of the genome. In Chapter 3, we will discuss the nature of that structure and the difficulties it presents to data processing. In particular, we will focus on the uncertainty faced during alignment. We will also present our algorithm for characterizing the structure and locating the *similar regions*. In Chapter 4, we validate the similar regions by quantifying how much alignment and variant calling struggle in those regions as compared to how they perform in the unique regions. In Chapter 5, we present a case study detailing how the structure inherent in the similar regions we identify may be used to improve variant calling accuracy in those regions.

Chapter 3

The Problem of Similarity

3.1 Introduction

Advances in DNA sequencing technologies have the potential to change the landscape of biological research [114]. Over a decade ago, the Human Genome Project [65] sequenced the human genome on a $3 billion budget. However, as a result of massively parallel sequencing machines, which produce *short reads*, or DNA segments of around 100 characters, sequencing costs are decreasing at a rate faster than Moore's Law and have recently broken the $1000 per genome barrier. This technology has the potential for widespread clinical usage in the near future, pending the development of short-read analysis pipelines of sufficient accuracy and efficiency.

Prior to semantic analysis, such as whether a person is likely to get cancer, short reads must be converted to variant calls. *Variant calling*, which is largely a statistical inference problem, is the process of constructing a sample genome from the noisy short-read data. It relies on a reference genome, which is a composite of the genomes of a few donors and was produced using sequencing technology that is far more expensive and accurate than short-read technology. The name of this process comes from its goal, which is to find variants between the sample and the reference; since humans share 99.9% of their DNA, the set of variants is small relative to the genome size.

Most variant calling algorithms rely on reference-based alignment, which is the problem of determining for each short read the genome location from which that read likely was generated. Then, for each location in the genome, variant calling algorithms use all the reads that aligned near the location to determine whether or not a variant exists. Therefore, in order to obtain high-accuracy variant calls, it is crucial to perform alignment with as few errors as possible. Any errors that are made during the alignment stage can lead to variant calling errors as well, since a read that is placed in the wrong location may give erroneous information about the presence of variants there. It could lead to true variants being missed, if the incorrect read matches the reference, or false variants being called, if the incorrect read differs from the reference. Thus, in order to optimize our variant calling accuracy, we were

motivated to identify factors that make alignment difficult and then ameliorate them.

We participated in developing the SNAP alignment algorithm [139], which gave us valuable insights into the complexities of alignment. Our main finding was that the structure of the genome itself is an obstacle to accurate alignment because of the presence of near and exact duplication, or repeats. Repeats make alignment difficult because if a read comes from a repeat region, there may be multiple, or even many, potential locations where the read could align. However, as alignment is a computationally expensive stage of the short-read processing pipeline, in order to reduce their runtime, most aligners are designed to minimize the work they do for any particular read. Thus, most aligners use heuristics by which they skip some potential alignment locations, *e.g.,* if the aligner detects that part or all of a location's sequence appears too often by its standards. Skipping alignment candidates can cause the aligner to miss the correct alignment location. Therefore, repeats contribute to alignment errors.

One important observation is that although aligners can be configured to skip reads that have many potential alignments, this strategy is problematic when the alignment is used for variant calling. If the aligner avoids placing multi-mapping reads, many repeat regions will have very few aligned reads, and variant calling in these regions will be difficult or impossible. The lack of variant calls in repeat regions is undesirable since we usually want to call variants in the entire genome to facilitate a broad range of studies. In addition, since there is not only exact duplication but also near duplication, there is often an unambiguous best alignment for multi-mapping reads, so it is necessary to try to align these reads. For many reads, given sufficient time and after performing a sufficient number of comparisons, the correct location may be found.

It is well known that the genome is characterized by repeats; a commonly stated figure for the human genome is that 50% of the sequence is repetitive. We will discuss this in more detail in Section 3.2. The genome's repeats exhibit incredible variation in characteristics like scale, number of times the sequence is repeated, degree of similarity between repeats, and more. Following from the repeats' heterogeneity, some of the repeats cause more trouble for the aligners than others. Most aligners get accuracy figures over 90%, so clearly, aligners do not struggle with all 50% of the genome that is repetitive. For example, some of the repeats are shorter than a read length, so aligners can distinguish between potential alignment locations based on the unique sequence that the read contains. Other repeats are longer than a read but shorter than a fragment, so aligners can use paired-end information to disambiguate; *i.e.,* if one read lies in the repeat while the other one is in a unique region, the aligner will know to place the ambiguous read near its uniquely-mapping pair.

In order to improve alignment accuracy, rather than considering half of the genome, which would be unwieldy and unnecessary, we endeavor to identify and focus on the subset of the repeats that make alignment hard. Therefore, we take a different approach to finding the repeats that is driven by the perspective of the short reads, in which we seek to identify the parts of the genome to which it is difficult to align reads. Furthermore, rather than a binary indicator of whether or not a region is in a repeat, we compute the genome's similarity structure, so we know which regions are a given repeat's neighbors. As we will discuss in

Chapter 5, knowing the similarity structure provides insight into how to fix alignments since we know where incorrectly aligned reads were likely to originate, which in turn helps to improve variant calling accuracy.

In this chapter, we begin in Section 3.2 by providing concrete information about the repeats in the genome. Then, based on our practical experiences working on an alignment algorithm, we argue that in many cases, unambiguous alignments are possible, even in the similar regions; thus, it is worth striving to improve these alignments' accuracy rather than just throwing up our hands. In Section 3.3, we describe our initial approach to locating the similar regions. While this approach was ultimately unsuccessful, due both to the complexity of the algorithm and to some practical infrastructure issues, it is instructive since it inspired our ultimate approach and because it illustrates the difficulty of computing on the genome. Section 3.4 introduces the final SiRen algorithm, named for its focus on locating similar regions. We begin by presenting the basic algorithm and continue by discussing both algorithmic and implementation optimizations that were necessary to achieve a working tool. We also discuss the computational setting in which we deployed SiRen. We conclude the chapter in Section 3.5 with a discussion of the relevant literature.

3.2 Similarity in the Reference Genome

As we discuss in Chapter 2, variant calling pipelines rely heavily on an underlying reference genome, particularly in the alignment step. In this section, we begin by discussing the complex structure of the human reference genome, namely, the presence of near and exact duplication, from a biological perspective. We then illustrate how this structure leads to both computational and statistical problems in alignment and therefore variant calling. Finally, we provide some of our experiences working on short-read alignment that relate to similar regions in the genome.

Biological Perspective

A recent review paper by Treangen and Salzberg [130] provides an excellent discussion of the origins and characteristics of repeats in the human reference genome. They discuss repeats in the genomes of other species as well; however, we primarily focus on the human genome in this work, so we will not refer to their observations about other species here. As mentioned above, approximately half of the human genome is made up of repeated sequence. Repeat regions can vary along many degrees of freedom, including semantics, degree of identity, length, spacing, and copy number. Regarding semantics, some of the repeated sequence in the genome is thought to be non-functional, while other portions seem to have played a role in our evolution. Repeat sequences can be either exact or near duplicates; near duplicates present more computational challenges, as we will discuss in the next section. Repeats can also vary tremendously in length; they can be as short as two base pairs, or as long as millions of base pairs. The spacing between repeats can vary from none, in the case of repeats that

are adjacent to each other and are called tandem repeats, to long distances, in the case of interspersed repeats. Finally, repeats can come in pairs, or they can have many similar sequences.

We reproduce Figure 3.1 from [130] with author permission. Part (a) of the figure provides a summary of the different classes of repeats in the human genome, along with many of their characteristics, including type, number, genome coverage, and length, in base pairs (bp). Microsatellites, minisatellites, and satellites are short tandem repeats from 2-100 bp. An example of a microsatellite would be `ATATATAT...`; the sequence "AT" appears several times, with each repeat next to the previous one. Short interspersed nuclear elements (SINEs) are longer, ranging from 100-300 bp, and are spaced throughout the genome. An example of a SINE is an *Alu* repeat, which is a well-known and abundant type of transposable element in the human genome. DNA transposons, 200-2,000 bp in length, are sequences that can move from one location in the genome to another. Retrotransposons are DNA elements that can modify DNA by first modifying the RNA after it has been transcribed from the DNA, and then modifying the DNA through reverse transcription. Long terminal repeat (LTR) retrotransposons have long repeats, ranging from 200-5,000 bp in length, on each end. A long interspersed nuclear element (LINE) is 500-8,000 bp in length and, like a SINE, is scattered throughout the genome. The ribosome is the cell location where protein synthesis occurs; ribosomal DNA (rDNA) consists of the tracts listed in parentheses in Figure 3.1 and is quite long with a range from 2,000-43,000 bp. Finally, other large variants such as segmental duplications range from 1,000-100,000 bp.

Clearly, the repeats in the human genome vary widely in terms of their length and characteristics. They also vary in terms of their impact on short-read data processing, which is of course our main concern. We are principally concerned with repeats that are at least as long as a short read, which is about 100 bp in length. Shorter repeats are not too concerning, since the read is long enough to contain some unique sequence around the repeat that helps to distinguish between potential alignment locations. However, longer repeats can present difficulties when we attempt to align reads to them, since there may be multiple locations at which the reads coming from those genome locations can align well. In what follows, we will discuss in more detail how genome repeats impact short-read processing, particularly alignment and variant calling.

Computational Perspective

If the human reference genome was generated uniformly at random, then processing short reads of 100 bases would be straightforward, since identical 100-base substrings within the reference would occur with extremely low probability. However, as we explained in the previous section, the repeats in the genome can be divided into two categories: exact duplication and near duplication. Exact duplication is easy to address. First, regions of exact duplication of a certain size can be located using a simple hash table, with the key being the substring of the indicated length, and the value defined as a list of locations in the reference where the substring appears. Second, during alignment itself, when a read aligns to any

a

Repeat class	Repeat type	Number (hg19)	Cvg	Length (bp)
Minisatellite, microsatellite or satellite	Tandem	426,918	3%	2–100
SINE	Interspersed	1,797,575	15%	100–300
DNA transposon	Interspersed	463,776	3%	200–2,000
LTR retrotransposon	Interspersed	718,125	9%	200–5,000
LINE	Interspersed	1,506,845	21%	500–8,000
rDNA (16S, 18S, 5.8S and 28S)	Tandem	698	0.01%	2,000–43,000
Segmental duplications and other classes	Tandem or interspersed	2,270	0.20%	1,000–100,000

b

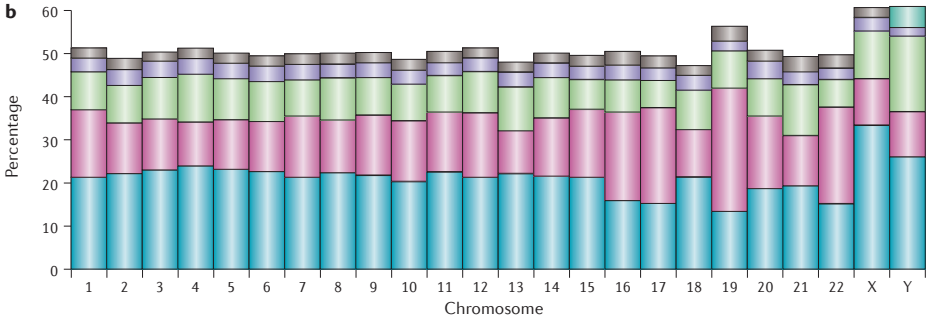


Figure 3.1: Treangen and Salzberg's summary of repeats in the reference genome [130]. (a) Types of repeats and their characteristics. (b) Per-chromosome repeat prevalence.

region of exact duplication, it can be marked as ambiguous since it is impossible to locate it uniquely.

At this point, it is natural to ask how much of the genome is characterized by exact duplication. The simple hashing strategy just mentioned works well for a single chromosome, but it fails on the genome due to inordinate memory requirements. Therefore, we employ the strategy we outline in Section 3.4, and in Figure 3.5 in particular, setting the merge distance h to 0. We thereby determine that exact duplication makes up 5.2% of the genome.

Unfortunately, even in theory, near duplication is a drastically more challenging problem to tackle. First, locating near-duplicate regions is much more difficult, requiring unsupervised approaches like clustering that are quite expensive given the scale of the genome. Many approaches exist for finding repeats; however, they do not meet our needs for several reasons, which we will discuss in detail, both in Section 3.5 and in Chapter 5. Moreover, even if suitable regions of near duplication are identified, analyzing reads that align against them is complicated. Since these are regions of *near* duplication, there is often an unambiguous best answer to where a given read originated. However, to find this location, an alignment algorithm must compare the read to all locations associated with a region of near duplication. Performing these comparisons can be quite expensive, and the common practice of using heuristics to prune the search space introduces errors.

Recall that alignment is an intermediate step in short-read processing, and our ultimate goal is to call variants. Nonetheless, alignment errors resulting from near-duplicate regions

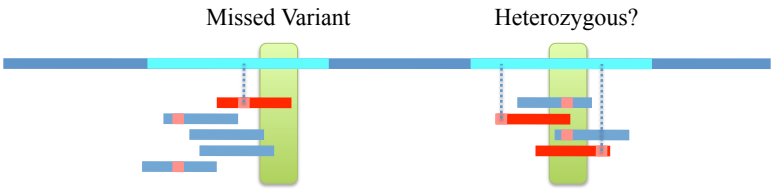


Figure 3.2: Alignment errors lead to difficulty during variant calling.

can also cause variant calling errors, as we illustrate in Figure 3.2. In the figure, the top line represents the sample genome, and the two cyan regions indicate locations in the sample genome that correspond to regions of near duplication in the reference genome. While most of the short reads are correctly aligned to the underlying sample genome, a few of the reads, which are marked in red, are aligned incorrectly. As the figure shows, these alignment errors can introduce errors in variant calling. First, consider the region on the left which has two variants, denoted in pink. However, due to the incorrectly aligned read, we may miss the second variant. Second, if we look at the region on the right, we see that the sample has a single variant in that region. However, since we have some misaligned reads, we end up with a conflicting story for whether a variant is present; half of the reads say yes, and half say no. This split might lead us to assume that a systematic sequencing error is the culprit for this situation. We also might assert that the sample is heterozygous, *i.e.,* its two corresponding chromosomes differ at this location.

We observed this issue in our dealings with real data from the Venter genome, which we obtained via the SMaSH data and benchmarking suite [125]; see Figure 3.3 for details. The data is from two regions in chromosome 22 that are similar to each other. The top track is the true variants, and the lower tracks are the reads that aligned to each region. The visualization is from IGV, which is described in [109] and [129]. Some reads aligned correctly, and others aligned incorrectly; this is indicated by the "Correct" and "Errors" labels. The correct reads aligned to region 1 indicate the presence of a homozygous variant at the emphasized location, which is true, while the errors do not. In region 2, we see the reads that should have aligned to region 1 indicating a variant where one does not exist. Hence, near-duplicate regions in the genome can cause problems not only in the alignment step, but throughout the entire variant processing pipeline.

A natural question is whether alignment difficulties can be ameliorated through the use of paired-end data, which we discussed in Chapter 2. The answer depends on the length of the repeats in question. Certainly if a repeat is longer than a read but shorter than a fragment, paired-end data can be very useful, since even if one read in the pair aligns to the repeat region, the other read may align to unique sequence nearby. The read that aligns to unique sequence near one instance of the repeated sequence can help the aligner disambiguate between multiple potential alignment locations. If the repeat is longer than the fragment length, however, paired-end data will be of limited use for choosing among multiple good alignments, since the entire pair may align equally well to each instance of

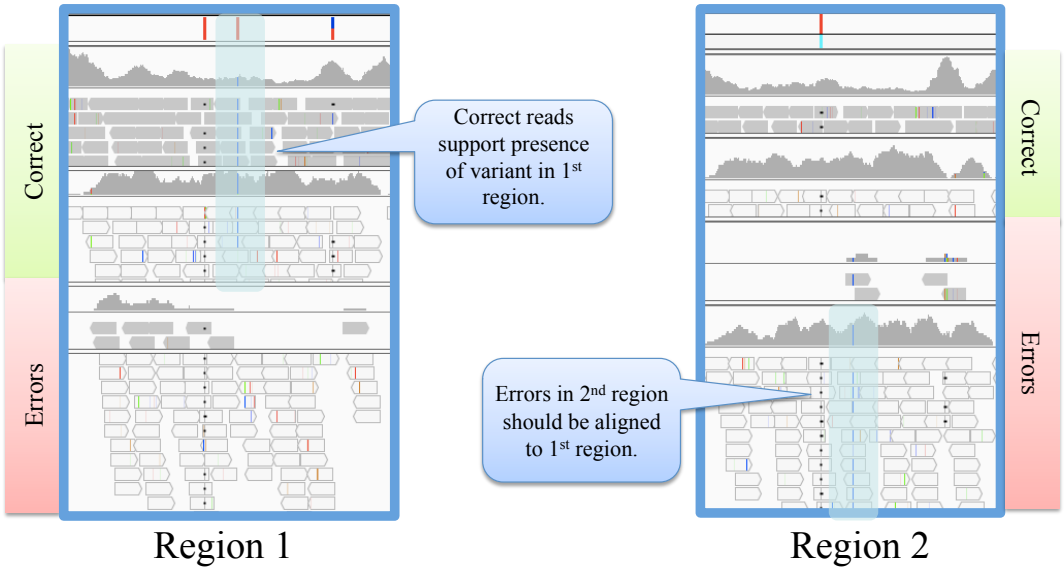


Figure 3.3: Alignment errors in real data from the Venter genome.

the repeat. Looking ahead, in Figure 4.8, we provide information about the lengths of the similar regions we identify. Many similar regions are shorter than a fragment; however, they still present difficulties for alignment due to the fact that regions are often close together.

One way to increase the set of repeats the aligner can align to uniquely is to use multiple libraries, each with a different insert size, as discussed in [12]. For example, a common insert size used to prepare short reads is around 500 bp. Therefore, if a repeat is longer than about 700 bp, it will be difficult to align reads generated from that sequence. If another library is used, say with a long insert size of several thousand base pairs, read pairs from this library, often called "mate pairs," will be able to align uniquely even in the presence of much longer repeats of a few thousand bases. Therefore, by adding another library, we can confidently align reads to more repeat regions. However, as Figure 3.1 indicates, even with a mate pair library, some long repeats will continue to cause confusion.

An exciting recent development, which we discuss in Chapter 2, Section 2.3, is long reads made available via third-generation sequencing technology, from companies such as Oxford Nanopore[1] and Pacific Biosciences, or PacBio[2]. Rather than generating paired reads of about 100 bp each, these new sequencers produce single reads that are one to two orders or magnitude longer, *i.e.,* from 1k-10k bp. Like long-insert libraries, these reads enable unique alignment to repeats that would make confident alignment of short reads impossible. However, we face the same difficulty as with mate-pair libraries, which is that some repeats are longer than these reads, so we still cannot align uniquely to those repeats. For example, an analysis of mappability of 1k bp reads shows that some biologically important regions

[1]https://www.nanoporetech.com/
[2]http://www.pacificbiosciences.com/

remain unmappable with these reads [82].

In what follows, we will discuss some of our practical experience gained from working on an alignment algorithm. While the details of the alignment algorithm itself are outside the scope of this work, by participating in its development, we gained valuable insights into the difficulties in aligning short reads that are caused by repeat regions.

Experiences Working on Alignment

We participated in the development of the SNAP alignment algorithm [139]. SNAP is a seed-based aligner which first builds an index of the reference genome, where the key is a *seed*, which is a short genome substring around 20 bases long, and the value is a list of positions at which the seed occurs. SNAP puts all overlapping seeds into the index rather than only disjoint seeds. To align reads, SNAP selects multiple seeds from the read and looks them up in the index to get candidate alignment locations. It then performs an edit distance calculation between the read and the candidate location in the genome to determine whether the location is a good match. SNAP indexes the seeds rather than read-length substrings to allow for some mismatches between reads and the genome, which could be caused either by sequencing errors or true variation between the individual being sequenced and the reference genome. Even if the read has some mismatches, a few of its seeds will likely match the genome exactly, allowing SNAP to find some good candidates for alignment. Of course, SNAP uses many clever optimizations to reduce the runtime, such as only comparing against genome locations that are indicated by multiple seeds from the read; however, we will not discuss those in detail.

We performed a parameter sweep to determine which of SNAP's parameters affect its performance. The metrics of interest were the error rate, the fraction of the reads that SNAP could confidently align, and the speed. By "confidently align," we mean that SNAP is sure about the position it chose because the next-best position is worse in terms of edit distance by at least one edit. We discovered that the only parameter that has a significant impact on these metrics is m, or "max hits," which is the cutoff for seed popularity. Normally, SNAP chooses a seed from the read and looks it up in the index, and the index provides a list of positions at which that seed occurs. Then, it compares the read against the sequence at those positions. However, if the number of positions at which the seed occurs is greater than m, SNAP will not compare the read against any of the positions indicated by the overly popular seed. The reason for this parameter is to reduce the number of comparisons performed and therefore increase the speed. This strategy is helpful in the case of degenerate seeds that occur many thousands of times in the genome. However, the cost is that we may miss the correct alignment location.

In Figure 3.4, we provide a quantitative demonstration of this phenomenon. In part (a) of the figure, the graph shows the error rate and the fraction of the reads that SNAP aligned confidently as m increases. As m increases, SNAP is able to align more reads, since it explores more potential alignments. Another benefit is that the error rate substantially decreases. This decrease can be attributed to the fact that SNAP performs more comparisons, so it

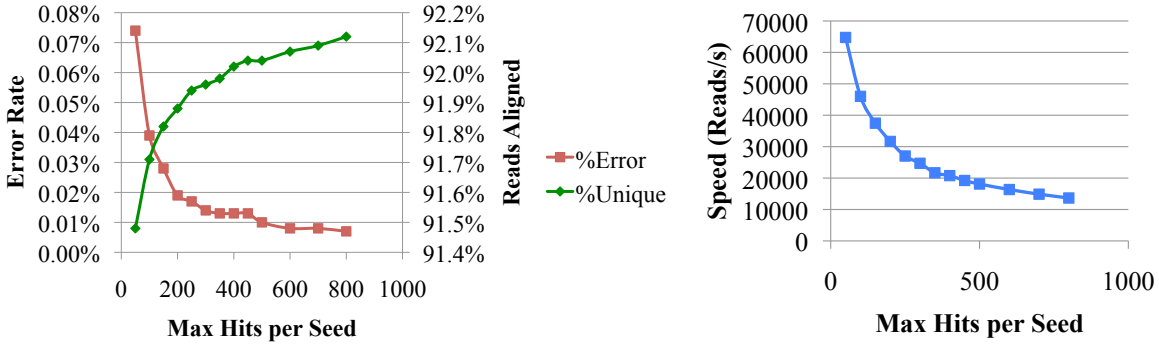


Figure 3.4: Effect of the "max hits" parameter on (a) error rate and fraction of reads aligned and (b) speed for the SNAP aligner.

is less likely to choose the wrong location. Clearly, some frequently-occurring seeds can provide valuable information about where a read should be aligned; this suggests that there is unique sequence around the popular seeds that allows SNAP to unambiguously align the reads. However, in part (b) of the figure, we see that the cost of exploring more genome locations is significantly reduced speed, which is obviously undesirable.

Based on these observations, we were motivated to learn more about similar regions in the genome. Clearly, during alignment, the similar regions cause a lot of difficulties. It is predictable that large values of m would cause a reduction in speed; however, it is somewhat surprising that increasing m allows SNAP to align reads more confidently and accurately. This improvement means that the repeats that are represented by these popular seeds are not just exact duplicates, but are more likely near duplicates. If they were exact duplicates, comparing against more locations would actually decrease the number of reads you could confidently align, since for many reads, you would discover that there are multiple locations at which a given read aligns equally well. However, since in many cases we are able to find the correct location by doing more comparisons, we are likely hitting similar regions. Though there may be many locations at which a given read aligns well, in many cases, there is an unambiguous best alignment for the read. Finding the correct alignment location is crucial since it will impact variant calling later in the processing pipeline. Therefore, based on our experiences with alignment, we decided to locate the similar regions, in hopes of eventually using them to improve alignment accuracy in those regions. In what follows, we will discuss our initial strategy for locating the similar regions and what we learned from attempting to implement it.

3.3 Initial Approach to Locating Similar Regions

Instead of looking for similar regions of arbitrary length with widely varying characteristics, as they do in some of the previous work, we constrain our search to regions whose characteristics are more closely related to the short reads themselves. This framing of the problem is possible since our goal is to improve the accuracy of short-read processing pipelines, rather than to learn general information about repeat regions in the genome. Therefore, we restrict our attention to substrings whose length is the same as that of the short reads.

A natural way to find the similar regions is to create a similarity graph that represents the genome, with a node for each substring and an edge between any pair of substrings that are similar. The question, then, is how to find the edges. The naïve strategy would be to do all $N \times N$ comparisons, *i.e.,* to compute the Hamming Distance between each pair of substrings. However, this is intractable since N is approximately 3.1 billion. Therefore, we must find a way to execute fewer comparisons. Note that while this step is a precomputation, so the performance does not have to be extremely fast, we still must create a solution that runs within the resource constraints of our available hardware. It must also complete in a reasonable amount of time, to enable debugging and parameter selection. In addition, we may want to rerun our computation when read lengths increase or the reference genome is updated.

Our initial strategy, which we describe in this section, uses multiple hashes so that similar substrings co-occur more often than unrelated substrings. This strategy failed, however, largely due to infrastructure limitations. In Section 3.4, we discuss an alternate, and ultimately successful, strategy for reducing the number of comparisons we had to perform.

Algorithm Sketch

Our initial approach to locating the similar regions was inspired by our observations about locating regions of exact duplication, *i.e.,* that you can do so easily by hashing each overlapping substring of a given length, say the length of a read, or R. The key is the substring, and the value is a list of positions at which that substring occurs. Then, each (key, value) pair with two or more entries in the value corresponds to an instance of exact duplication.

If two substrings are similar to each other, they would hash to different locations in the map, and we would not receive any indication that they are related. Therefore, we cannot use a standard hash map to recognize near duplication. Instead of hashing on the entire substring of length R, like we do in the case of exact duplication, to locate the similar regions, we hash on a subset of the substring. Rather than choosing a contiguous subset of the substring on which to hash, we instead choose C random columns of the substring, *i.e.,* positions in the substring. With this hashing input, the probability that two unrelated substrings will hash jointly is $1/4^C$ since there are four possible values for each column (the four DNA bases `A`, `C`, `T`, and `G`), which will be a very small value for most choices of C. However, the situation is somewhat different for two related substrings. Two substrings with a fraction P of their bases in common will be much more likely to hash together; the probability is P^C, where

p(same bucket)	Expression	Value for $C = 16$ and $P = 0.9$
Two random substrings	$1/4^C$	$2.33e^{-10}$
Two P-similar substrings	P^C	0.19

Table 3.1: Hashing probabilities for random vs. similar substrings.

P is much larger than 1/4. In Table 3.1, we provide the values of the probabilities that unrelated and related strings will jointly hash given example parameters.

Given our choices of P and C, the probability that two similar substrings will co-occur is still quite low at 0.19. We would like to increase our chances of finding similar substrings without at the same time generating lots of false positives, *i.e.,* strings that are not actually similar hashing together. Thus, we add another step to our approach. Instead of hashing the substrings once, under a particular set of C columns, we hash the substrings many times with H independent hash functions. The independent hash functions are obtained by different selections of C random columns. Then, similar substrings will hash together much more often than random substrings.

The expected number of buckets B in which two substrings will jointly hash is given by $B = p(\text{same bucket}) * H$. Since we have H hash functions, each of which leads to a set of buckets, we have H sets of buckets. If two substrings are colocated in at least B buckets, we say they are similar. For our choices of P and C, and setting $H = 100$, we obtain $B = 19$ for similar substrings, while still only obtaining $B = 2.3e - 8$ for random substrings.

This algorithm has three stages. First, we must produce H sets of buckets via independent hash functions. Our second task is to determine which substrings are similar to at least one other substring. To do so, for each pair, we must count the number of buckets in which they co-occur. Then, we say that any pair that co-occurs at least B times is similar. Third, to find groups of similar substrings, we must find the connected components of the graph.

Implementation Issues

Our implementation strategy was primarily to use Spark [139] to execute what we expected to be a large computational load in a distributed manner. The first stage of our computation involves computing the H independent hashes. This strategy proceeded smoothly when we tried running it on a single chromosome; however, we encountered many issues when attempting to apply it to the full genome. This complication was mostly due to our choice to run our jobs on Amazon EC2[3], a cloud-based distributed computing platform where users can deploy their virtual machines on Amazon hardware. Though the hashing jobs often completed successfully, some of our jobs failed due to being unable to publish their data to S3[4], Amazon's cloud storage service. This difficulty was due to transient errors with S3.

[3] `aws.amazon.com/ec2`

[4] `aws.amazon.com/s3`

Thus, we had to rerun many of our hashing jobs, incurring extra monetary costs and delaying our research.

For the pair counting task, we used a MapReduce-style computation. For each hash, we iterate over the buckets; for each bucket, we enumerate the pairs. During the Map step of the computation, we emit a tuple of the form `(pair, 1)`, where the one indicates that the pair co-occurs in one bucket. The Reduce step involves counting how many times each pair co-occurs by grouping tuples by their key, *i.e.,* pair, and summing the values. Again, we encountered problems trying to read the hashes from S3 and then publish pair counts back to S3. It is possible that in the time since we attempted this, S3 has improved such that this task could be run successfully. However, at implementation time, this proved to be an intractable obstacle, and we could not wait for S3 to fix these issues. We also generated several terabytes of intermediate data since we had to retain all $H = 100$ hashes, and our storage costs were significant. Even with generous support from Amazon, we exhausted our budget through many attempts to complete our jobs. Therefore, we made the decision to abandon this approach.

We would like to note that selection of optimal values of the parameters of this algorithm, *i.e.,* C, P, and H, was impossible due to the difficulties in running our implementation. We chose reasonable initial values based on our experience with alignment. If further work along these lines were to be attempted, a parameter sweep would be appropriate.

In the following section, we will describe what proved to be a successful, tractable approach to identifying the similar regions in the genome. It also involves the paradigm of a similarity graph of the genome. However, the subsequent techniques we employed to build and process the similarity graph proved to be more amenable to our computational setting.

3.4 SiRen

In this section, we will discuss the SiRen algorithm. It was inspired by our difficulties with our initial approach, which we describe above. We streamlined our approach; and even with a more straightforward algorithm, we had to perform many algorithmic optimizations and implementation tricks to render it tractable. We take a much more constrained approach to finding similar regions than most of the previous work, which we discuss in Section 3.5; this also helps us simplify our problem. We begin by presenting the core of the SiRen algorithm, which is based on applying the well-known union-find algorithm to a similarity graph that represents the genome. Then, we discuss some significant optimizations that we had to make in order to actually run this basic algorithm; they involved leveraging an index to reduce the number of comparisons we had to perform and partitioning the graph so that we could parallelize our approach. We also describe some of the implementation choices that we made. Finally, we explain how we deployed our SiRen tool.

Detecting Similar Regions with SiRen

We have developed the SiRen algorithm for detecting similar regions in the reference genome. We begin by creating a similarity graph to represent the genome. In our graph, there is a node for each overlapping substring of length R, where R is the length of the short reads of interest; we have approximately three billion nodes. There is an edge between any two nodes for which the Hamming distance between the two substrings is less than or equal to a given threshold h. We illustrate this in Figure 3.5(a). In the figure, the substring length R is ten, and the Hamming threshold h is one. Since the Hamming distance between the top node and right node is one, an edge connects them; however, since the Hamming distance between the top node and left node is seven, no edge connects them. We then locate the connected components of this graph and discard any singletons, which we show in Figure 3.5(b). We will further discuss this part of the algorithm below.

Next, we transform our connected components to a set of similar regions. We first map the substrings corresponding to nodes that belong to connected components back to the reference. If one or more of these substrings are found to overlap, we merge them. We refer to the longer substring that results from merging two or more length-R substrings as a *similar region*; for clarity, we call a genome substring that does not contain any similar regions a *unique region*. In Figure 3.5(c), the top line is the reference genome, and the blue, orange, and green line segments are length-R substrings that belong to connected components of the corresponding color, from part (b) of the figure. In the first and fourth group of line segments, the segments overlap, so each set of segments is merged to create a similar region. The red segments in the reference genome represent the similar regions; the intervening blue segments are the unique regions. The second and third similar regions come from a single segment each since no other segments overlap them.

We further describe this process in Algorithm 1. For simplicity's sake, the algorithm states that the similar substrings are stored in a set S_S; however, we actually maintain information about the similarity structure of the genome rather than collapsing all the similar substrings into a single set. In what follows, we will provide more concrete information about how we not only determine the edges but also locate the connected components of the genome similarity graph with a variation of the basic union-find algorithm. While the algorithm itself is simple, naïve implementations are intractable due to computation and storage costs. Thus, we will also discuss optimizations based on indexing and parallelization that are necessary to handle the scale of this problem.

Index-Augmented Union-Find Algorithm

Given the nature of our problem, we identified the union-find algorithm as a promising approach. Union-find is a simple, well-known algorithm for finding the connected components in a graph [23]. Initially, each node is assigned to its own component, or cluster. Given a list of all the edges in the graph, we traverse each edge, and if the nodes connected by a given edge are not in the same cluster, we merge them. The basic algorithm is depicted in

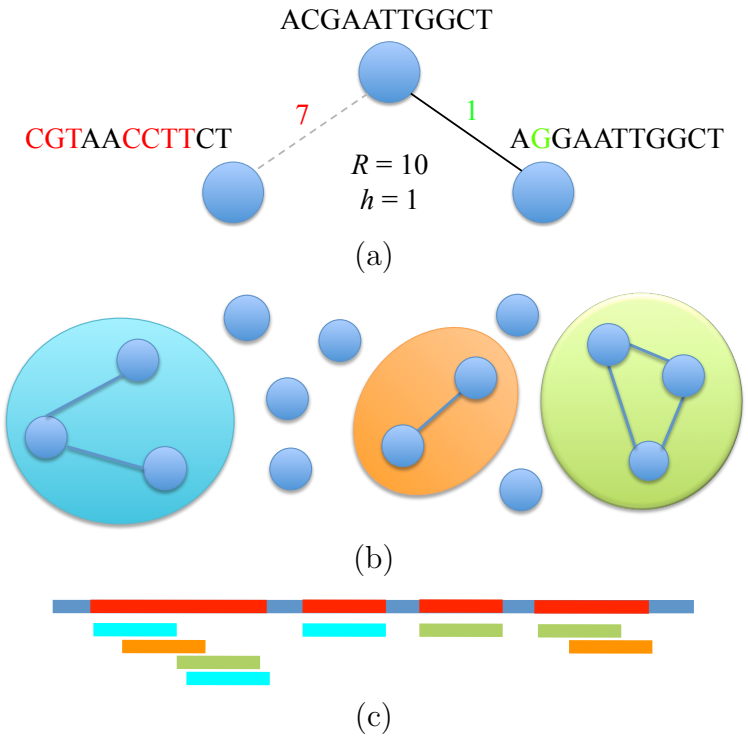


Figure 3.5: Schematic of SiRen algorithm. (a) Graph representation. (b) Locating the connected components. (c) Merging substrings to create the similar regions.

Algorithm 1 SiRen algorithm

R = read length, h = Hamming distance threshold
Initialize the set of similar substrings $S_S = \emptyset$
Initialize the set of similar regions $S_R = \emptyset$
for all substrings s of length R **do**
  if $\min_{s'} Hamming\,(s, s') \leq h$ **then**
    $S_S = S_S \cup \{s\}$
  end if
end for
Partition S_S into disjoint subsets S', where the members of S' are overlapping
for all subsets S' **do**
  Merge the contents of S' to create a *similar region* s_r
  $S_R = S_R \cup \{s_r\}$
end for
return S_R

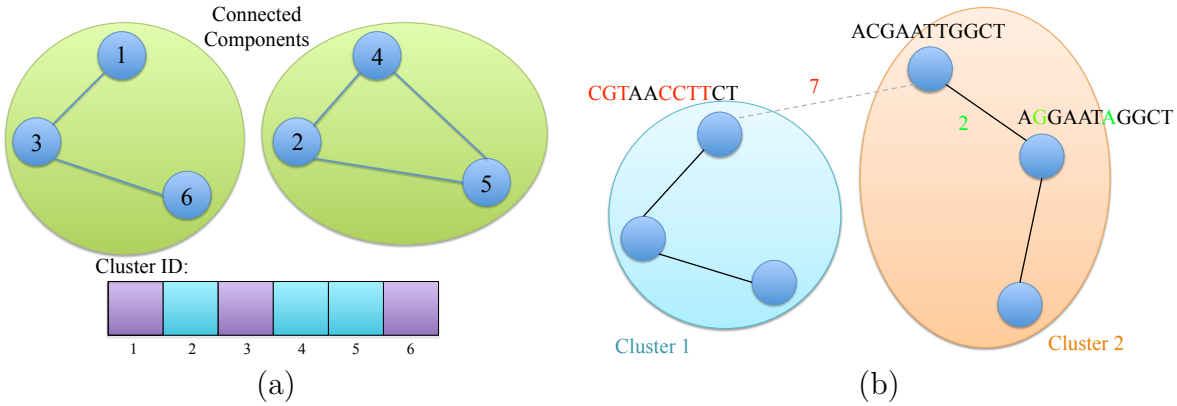


Figure 3.6: (a) Basic union-find. (b) Applying union-find to genome substrings.

Figure 3.6(a). To find the clusters in the graph, we traverse the edges, and whenever two nodes are connected by an edge, we merge them to be in the same cluster, updating the "Cluster ID" data structure as we go. After the algorithm terminates, the "Cluster ID" data structure reflects the set of connected components. In our example, nodes 1, 3, and 6 are in one component, colored purple in the "Cluster ID" array, and nodes 2, 4, and 5 are in another component, colored blue in the array. To apply union-find to our problem of clustering similar substrings, we represent each substring of length R as a node. We say there is an edge between any two substrings whose Hamming distance is less than or equal to a given threshold. Figure 3.6(b) illustrates this process.

Of course, we must first identify the edges in our graph before applying the union-find algorithm. A naïve approach is to compare each substring to each of the other three billion substrings. This comparison would be prohibitively expensive, however. Shrinking the runtime of this process is important to simplify development, manage computational costs, facilitate parameter selection, and find new similar regions whenever read lengths change or the reference is updated. Furthermore, even if it were computationally feasible, we would perform many unnecessary comparisons since the adjacency matrix is likely to be sparse.

To avoid this wasted work, we build an index of the reference genome, where the key is a *seed* of length 20 bases, and the value is a list of locations in the reference at which that seed occurs. The seed length must be shorter than a read so that even if the read contains mismatches from the reference, we will still have one or more seeds that match the reference exactly. We set the seed length equal to 20 because according to empirical observation, it achieves a good tradeoff in the space of seed length versus number of hits; shorter seeds lead to excessive hits, while longer seeds lead to too few hits [139].

After building the index, we then scan over the substrings. For each substring s, we look up its non-overlapping seeds of length 20 in the index; since our read length is 100, we have five seeds. Each seed gives us a list of positions in the genome whose corresponding substrings may be similar to s. After looking up each of s's five seeds in the index, we

obtain a list of all potential neighbors of s. The list length is orders of magnitude less than three billion, which is the number of comparisons we would have to perform under a naïve approach. Algorithm 2 describes this process.

Algorithm 2 Union-find with index optimization

R = read length, h = merge threshold, N = number of substrings in genome
Input: an index I, where the key is a 20-base seed, and the value is a list of all positions at which the seed occurs
Initialize: *parent*, an array of length N, where $parent(i) = i$ = cluster ID of the ith substring
for all genome substrings s of length R **do**
  for all seeds s' in s **do**
    $hits = I(s')$
    for all $s_h \in hits$ **do**
      if $Hamming(s_h, s) \leq h$ **then**
        Update $parent(s_h) = parent(s)$
      end if
    end for
  end for
end for
return *parent*

In Figure 3.7, we illustrate how we execute the union-find algorithm with the index optimization. The long string at the top of the figure represents the reference genome. We build an index of the reference genome, which we show on the figure's left. Given a query string "ACCTAAAA" from the reference, we look up its seeds in the index to produce a list of candidate matches; here, the possible neighbors are at positions 4, 16, and 24. For each candidate neighbor, we compute the Hamming distance, which we illustrate at the top of the figure, and update the adjacency matrix, which appears on the figure's right. If the Hamming distance is sufficiently low, we consider those two substrings to be similar. The green boxes illustrate comparisons with a low Hamming distance, while the red box shows a high Hamming distance between the substring and the reference. For each pair of similar substrings, we update the "Cluster ID" data structure, shown at the bottom of the figure, by merging the substrings' clusters. Therefore, at the end of this round of comparisons, substrings 4 and 24 belong to the same cluster, while we do not update substring 16's cluster membership.

Our approach for edge identification is a parametric one, where the parameter is the merge distance h. Moreover, since we require exact matches via seed lookups, our index-based algorithm may miss some potential edges if our merge threshold is greater than or equal to the number of seeds. In practice, we observe that a merge distance of 1 performed best in our experiments, and thus we avoid the issue of missed edges. We will address the parameter selection in detail in Chapter 4.

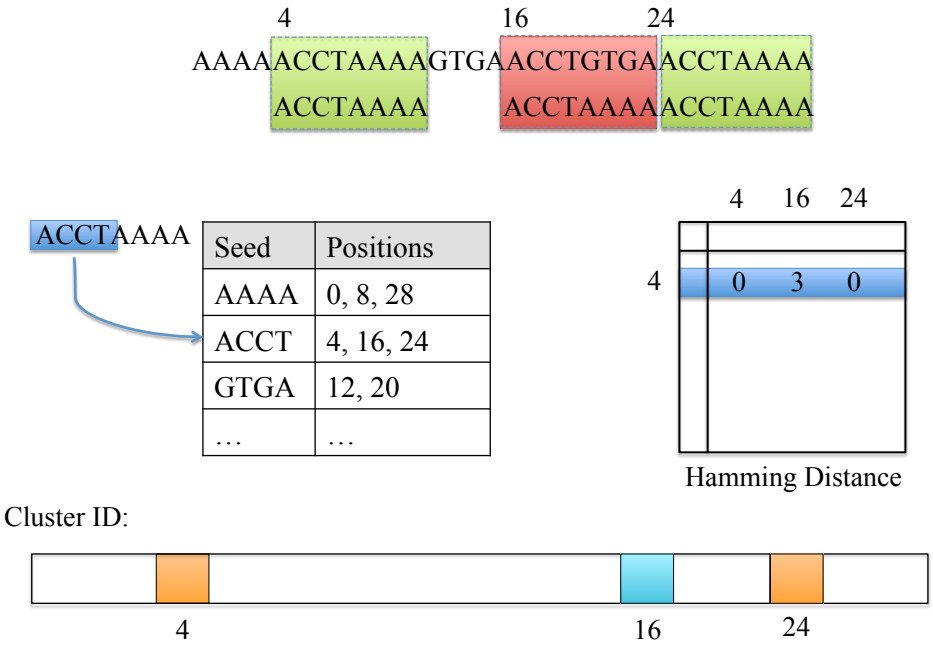


Figure 3.7: Index optimization to SiRen algorithm.

Parallelism Strategy

Despite the dramatic reduction in comparisons, our index-based algorithm remains quite expensive given the scale of our data. Our goal was therefore to develop a parallel implementation of our approach. To facilitate parallelization, we use Spark, a MapReduce-like framework for distributed computing [140]. Moreover, we divide our index-based approach into subproblems, each of which solves an index-based union-find problem on a partition of the complete dataset, and merge the results of the subproblems on-the-fly as they complete.

The first step of our parallel, distributed algorithm involves partitioning the adjacency matrix so that each task is allocated one section. In our study, we use grid partitioning based on exploratory analysis with various partitioning schemes. For example, we also tried what we called stripe partitioning, where each partition had only a portion of the rows but all the columns. However, we abandoned this approach because its memory requirements were so high that we could not run a per-partition task to completion.

In our grid partitioning scheme, each cell has two ranges: one for the rows and one for the columns. The row range determines which positions should be included in the index as seeds. The column range determines which positions should be scanned as substrings. We scan each substring in the column range, and whenever we locate an edge, *i.e.,* we find that two substrings are similar, we merge the two substrings' clusters. Therefore, we are able to combine the edge location and merging into a single pass, avoiding the need to store the edges. This one-pass approach is important since this is already a memory-intensive process. We also use the standard union-find optimizations of union by rank and path compression;

we will discuss these further in our presentation of implementation details.

The cluster results from each partition are merged on the master node. Once a worker node returns the clusters for a given partition to the master node, we must update the global cluster definitions for the entire genome that are stored on the master node. Initially, the master node stores each substring as its own cluster. Once a worker node computes the clusters for a single partition, the master updates the global clusters for the genome by merging any clusters whose points appear together in the partition's clusters. Since each partition is independent, our approach is embarrassingly parallel and therefore exhibits *strong scaling*, since adding more machines to a fixed-size problem shrinks the runtime. See Algorithm 3 for a formal description of this process.

Algorithm 3 Parallelization of union-find

```
Initialize: parent(i) = i, where length(parent) = N
Partition the adjacency matrix according to a grid scheme, with dimension G
for all grid cells g with row range r and column range c do
  Initialize: parent_g(i) = i, where length(parent) = |r| + |c|
  Create a seed index I with the portion of the genome indicated by r
  for all genome substrings s beginning with positions in c do
    Find the neighbors of s using index I
    Update parent_g
  end for
  return parent_g
end for
if task corresponding to grid cell g completes then
  for all pairs (s_1, s_2) s.t. parent_g(s_1) = parent_g(s_2) do
    if parent(s_1) ≠ parent(s_2) then
      Update parent(s_1) = parent(s_2)
    end if
  end for
end if
```

We illustrate how we can find genome-wide clusters even with a partitioning scheme in Figure 3.8. In the global clusters, the red and purple points are initially in their own clusters; this appears in part (a) of the figure. We partition the genome in a grid fashion, and worker tasks process different partitions in parallel, identifying the clusters within each partition. The green partition shown in part (b) of the figure yields three clusters, with the red and purple points grouped together. Part (c) of the figure shows that the master updates the global state so that the red and purple points are in the same cluster. For each such pair of points in a partition cluster, the master merges their corresponding global clusters.

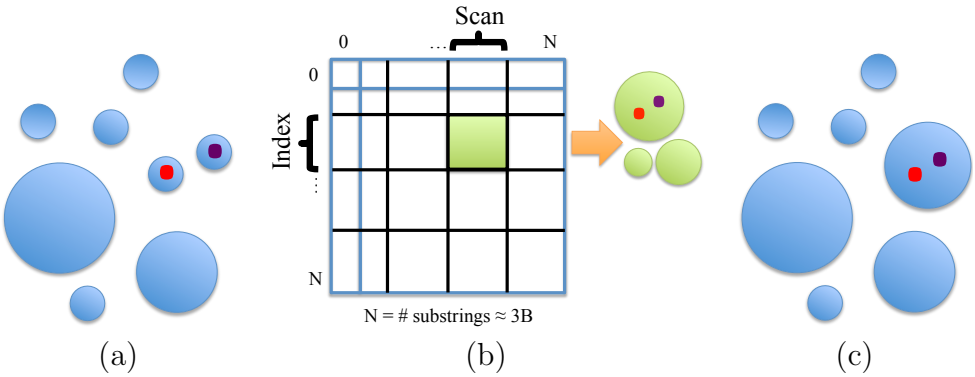


Figure 3.8: Parallelization of union-find. (a) Cluster initialization. (b) Grid partitioning. (c) Updates to the global state based on the results of clustering on one partition.

Implementation Details

In this section, we will describe some of the details involved in implementing our approach. Beyond the optimizations we describe above, we needed several implementation tricks in order to make this approach actually scale to our huge genome-sized application. We began by implementing the union-find algorithm in Scala. Besides its inherent desirable properties such as being concise and functional, Scala is a good choice for using with Spark. The important data structures include the parent array and the rank array; each array has one entry per substring. The parent array, which corresponds to the "Cluster ID" array that we mentioned above, indicates cluster membership; $parent(i)$ is the cluster to which the ith element belongs. We call it the *parent* because clusters can be thought of as trees. The two main functions that give the algorithm its name are the *union* and $find$ functions. Calling $find(i)$ returns the parent of the ith element, while $union(i, j)$ merges the clusters to which the ith and jth elements belong.

We illustrate union-find in Figure 3.9. In part (a) of the figure, we depict a tree corresponding to a cluster. Node 1 is the root of the cluster and is therefore its main parent. However, $parent(5)$ could point to either node 1 or node 2. We do not want the find function to return 2, since that is an internal cluster node; rather, we want it to return 1. Therefore, the find function is recursive; it will continue calling find on the internal nodes until it reaches the root of the tree. The find function also implements the *path compression* optimization, in which we update the parent data structure with the intermediate results obtained by calling find. We can think of this as flattening the cluster's tree, since more nodes are connected directly to the root. Say that $parent(5) = 2$ before we call $find(5)$. The find function will not only return 1 through its recursion; it will also update the parent data structure so that $parent(5) = 1$. Then, subsequent calls to $parent(5)$ will execute faster. Part (b) of the figure shows the updated tree.

The other main component of the algorithm is the union function. Suppose we call union on two elements, i and j. First, we call $find(i)$ and $find(j)$ to determine the cluster to

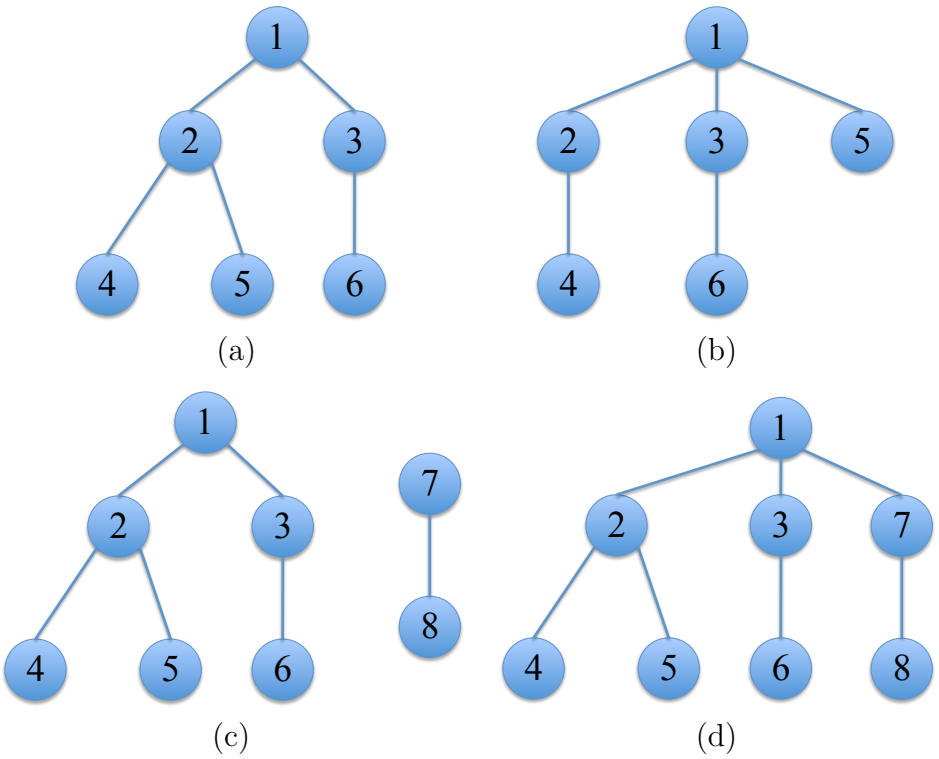


Figure 3.9: Details of implementing the union-find algorithm. (a) is before and (b) is after path compression; (c) is before and (d) is after union by rank.

which each belongs. Then, we merge those clusters so that i and j, as well as all the other nodes in their respective clusters, belong to the same cluster. While doing so, we perform the *union by rank* optimization; when we merge two clusters, *i.e.,* trees, we want to attach the smaller one to the larger one. We use the rank data structure, mentioned above, to keep track of which cluster is larger. We can think of $rank(i)$ as the depth of the cluster of which i is the root; however, when the path compression optimization is used, it may not be strictly true, as the tree will get flattened without reducing the rank value. In Figure 3.9(c), we show two trees, of which 1 and 7 are the roots. Assuming no path compression has taken place, $rank(1) = 3$, while $rank(7) = 2$. Therefore, upon calling $union(1, 7)$, we attach the tree with root 7 to the tree with root 1. To do so, we update $parent(7)$ to be 1 instead of 7 as it was initially. Note that we would get the same result if we called union on any pair of vertices where one vertex in the pair belongs to the tree with root 1 and the other belongs to the tree with root 7. We show the updated tree in part (d) of the figure. Note that both the path compression and union by rank optimizations are well-known components of the union-find algorithm.

As we mentioned earlier in this section, in order to run the algorithm, we had to parallelize our implementation. Thus, we had to modify our implementation to support our grid partitioning scheme as well as to run in a master-slave configuration via Spark. We created

```
val clusterSize = Array.fill(numBases)(0)
var firstMember = Array.fill(numBases)(0)
var nonTrivialClusters = List[Int]()
var nonTrivialMembers = scala.collection.mutable.Map[Long, LongArrayList]()
```

Figure 3.10: Data structures that we use to locate the clusters in a union-find object.

a version of our union-find implementation for grid cells on the diagonal and another version for grid cells that are off-diagonal. The version for the diagonal is simpler because we only have to keep track of one range, since we index and scan the same range for diagonal cells; therefore, we need the parent and rank data structures to keep track of just the positions in the given range. When we call find or union on a position, we convert it from an absolute position in the genome to a within-range relative position by subtracting the range's initial position. For the off-diagonal partitions, the parent and rank arrays must keep track of two ranges, one for indexing and one for scanning. In this case, in order to convert from absolute to relative coordinates, we must first determine the range to which a given position belongs; then, we can subtract the proper range's initial position to obtain the range-relative position.

We had to do some extra work to coordinate the per-partition union-find tasks. We used Spark's parallelize method to create one task per grid cell, and we indicated which grid cell each task represented by passing in the index and scan range initial positions and the range length, which is the same for both ranges, to the task. Within each task, we either create a UnionFindGrid task or a UnionFindGridDiagonal task, depending on whether or not the task's grid cell lies on the diagonal, which we can tell based on whether the index and scan ranges are equal. We then execute our union-find algorithm on the grid cell to locate the clusters for that grid cell. To report the results back to the master task, we use Spark accumulators [122]. The benefit of this approach is that instead of keeping all the intermediate results in memory for job's duration, which would cause us to quickly exceed our memory limits, we can incrementally update the master clusters with a task's results via the accumulator and then discard those results.

After we finish executing the union-find algorithm, we obtain the parent array indicating the cluster to which each point belongs. However, we want to be able to go the other way, from a cluster identifier to the members of that cluster. Therefore, we must process the parent array to find the clusters. Note that most clusters are singletons; in locating the clusters in the parent array, we want to ignore the singleton clusters. Therefore, we are only looking for clusters that have at least two members.

We have several data structures that are involved in locating the clusters, and we list these data structures in Figure 3.10. To locate the clusters, we begin by iterating over the parent array. For a given position p, the parent array indicates the cluster to which p belongs, which we will call c; then, we increment the corresponding location in the clusterSize array, or $clusterSize(c)$. Based on the value of $clusterSize(c)$, we will determine whether we should

promote the cluster to being non-trivial. If $clusterSize(c) = 1$, *i.e.,* the current position is the first member of this cluster that we have encountered, then we store the current position in our firstMember array; that is, we set $firstMember(c) = p$. If $clusterSize(c) = 2$, we have just discovered a new non-trivial cluster. We therefore add c to the nonTrivialClusters list. We also create a new list to store the members of c, to which we will add both the current position p as well as the cluster's first member, which is stored in $firstMember(c)$. We then add a tuple to the nonTrivialMembers map, where the key is the current cluster c and the value is the list that we just created. If the cluster size is greater than 2, we have already recognized that c is non-trivial; therefore, to update c, we just add p to its list of members. Note that we perform the cluster identification step at the end of each grid cell task as well as after the job finishes.

Deployment

We deployed the SiRen tool on our lab cluster. Each node in our cluster has a dual-socket Intel E5-2670 2.6 GHz processor with 16 cores and 128 GB of DDR3-1600 MHz RAM. We had to run multiple SiRen jobs so that we could explore how to set its parameter, the threshold on the Hamming Distance between two substrings, which we call h. We ran each job on a single node in our cluster, with 16 threads to take advantage of each node's 16 cores. This facilitates Spark deployment a great deal, since we do not have to configure several nodes. At the time we performed these experiments, Spark deployment was mostly through Mesos, a cluster manager [41]; the Spark developers had only recently started developing a Spark standalone deployment option. Since deployment through Mesos was rather inconvenient, and since we had the option of waiting for our results, running each job on a single node achieved the right tradeoff for us between extra hassle deploying the jobs and reducing the runtime. However, it is important to note that if results are desired with less latency, it is possible to run a SiRen job on multiple nodes, especially now that Spark standalone deployment is much more mature.

As we have mentioned throughout this section, our SiRen algorithm is memory-intensive. SiRen's memory requirements are largely attributable to having to maintain in memory the object that keeps track of the genome's global clusters. We found that we needed to allocate 100 GB of memory to our job in order to get it to run. We used the same memory allocation for each job, regardless of the value of h.

In this section, we stated that the SiRen algorithm is computationally expensive, thus inspiring us to optimize it. When we set $h = 4$, the SiRen job took 126 hours; since it was running on 16 cores, this translates to 2016 CPU hours. We could have run our jobs on AWS instead, in order to achieve isolation and get more reliable timing information. The r3.4xlarge instance type[5], which has 122 GB of memory and 16 cores, costs $1.40 per hour, so the total to run SiRen on a single instance of this type for $h = 4$ would be $176.40. Now

[5]http://aws.amazon.com/ec2/instance-types/

that we have finished developing SiRen, the cost to run it on AWS is quite reasonable, should any further runs of SiRen be necessary or desirable.

3.5 Related Work

The problem of finding repeats in the human genome has been well-studied; *e.g.,* see [2, 63, 132, 7, 103, 126]. Their goal is often to understand the evolutionary biology of the genome; thus, they have a broad scope, seeking repeats of heterogeneous scales and complexities. Some also allow strings of different lengths or with widely varying degrees of similarity to belong to a single cluster. Because the problem statement is so open-ended, their approaches are complicated and therefore computationally expensive. In addition, the figure that is usually stated for the portion of the genome comprised of repeats is 50%; recall that we discuss this in detail in Section 3.2 and especially Figure 3.1. As we will discuss in Chapter 4, we identify a much smaller fraction of the genome as belonging to similar regions, and we empirically verify that aligners struggle with these regions.

The approach for finding duplication in the genome that is most closely related to our approach is the GEM suite [28]. It also focuses on finding near duplication in the reference genome. They define a quantity called *mappability*, which is related to the inverse of how often a given substring appears in the reference, up to a certain number of mismatches that can be provided as a parameter. They compute the mappability for each position in the genome so that for any position, one may assess how feasible it will be to align reads at that position. The output of the SiRen algorithm, *i.e.,* the list of similar regions, is related to a mappability score, in that for any given position in the reference genome, we can say whether or not it belongs to a similar region, and we can also say how many substrings to which it is similar. Being similar to many other substrings corresponds to having a low mappability score. In Chapter 4, we provide a quantitative comparison of the similar regions and the GEM mappability scores. However, our work goes beyond mappability, since we compute the full similarity structure of the genome. That is, while GEM can say whether a given substring has many neighbors, it cannot identify those neighbors, and we can. This information will become crucial when we discuss our work on leveraging our knowledge of the similarity structure to improve the accuracy of downstream data processing in Chapter 5.

The problem of finding the similar regions in the genome can naturally be viewed as a clustering problem over a massive dataset, *i.e.,* three billion data points with dimensionality equal to the read length, with an unknown number of clusters. Though we do not know the number of clusters in advance, we expect it to be large; therefore, K-means does not apply since we do not know K, and determining the right K will be intractable. More sophisticated clustering approaches, *e.g.,* spectral clustering [116] and non-parametric Bayesian clustering techniques [42], do not naturally scale to our problem size, due to their computational complexity and/or inability to parallelize. Locality-sensitive hashing (LSH) [33] also appears to be a natural fit to our problem. With LSH, a family of hash functions is chosen such that

similar objects hash together while unrelated items hash to different buckets. In fact, we initially experimented with an approach like LSH, as we described in Section 3.3. However, the communication and storage requirements made this technique infeasible.

A natural question, given our use of Spark, would be why we did not also use GraphX [36], the graph processing framework built on top of Spark, to locate the connected components in our similarity graph. The primary reason is that at the time this work was completed, which was from 2011-2012, GraphX was not yet available. Another issue is that had we used GraphX to find the connected components, we would still have needed to produce the edge set E of our similarity graph. Doing so is non-trivial, as we discussed in Section 3.4; we required an index-based optimization to reduce the number of comparisons we performed. We still would have needed this optimization to use GraphX, and we also would have incurred a substantial storage cost, since we would have needed to save the edges to be input to GraphX. Therefore, while GraphX would have expedited our search for the connected components, the preprocessing needed to produce the genome similarity graph would still have been computationally expensive.

Another literature that pertains to clustering DNA sequences is metagenomics, which tackles the following problem: given a collection of short reads, identify the different species present in the sample. An interesting application of metagenomics is to sequence the human microbiome, which is the collection of the various species of microbes that live on or in the human body, to determine which species of microbes are present. One tool used to do this is UCLUST [30]. Since its goal is to come up with a clustering where each cluster corresponds to a single species, it does not focus on ensuring that all the strings within a cluster are similar to each other. Therefore, since its goal is different from ours, which is to find clusters with a high degree of intra-cluster similarity, the clusters that this algorithm produces are not useful for our purposes.

The use of MapReduce or Spark in bioinformatics algorithms and tools is still relatively new; we will discuss a few interesting examples here. ADAM is an open-source, special-purpose genomics processing engine built on top of Spark with an accompanying specialized file format for storing short-read data [89, 99]. As an example of its capabilities and features, it reimplements many commonly-used tools and pipeline stages from suites like PicardTools[6] and the GATK [27] in a distributed setting to improve throughput and scalability. Another project that uses Spark is eXpress, which is an EM algorithm for computing relative abundances of target sequences for applications using RNAseq, chromatin immunoprecipitation sequencing (ChIP-Seq), and metagenomic data [108, 107].

An example of a tool that uses MapReduce is CloudBurst [113], which is a short-read aligner that can report not only the unambiguous best alignment but also all alignment locations for a given read, without restricting the number of differences from the reference genome allowed. The use of Hadoop MapReduce[7] is crucial for providing this functionality without incurring so much latency that the tool becomes unusable. Another tool of interest

[6] `http://broadinstitute.github.io/picard/`

[7] http://hadoop.apache.org/

is Crossbow [38], which performs short-read alignment and variant calling. Crossbow leverages existing tools, relying on Bowtie [67] to align reads and SOAPsnp [78] to call variants; it achieves scalability by using Hadoop to distribute these tasks. Clearly, the bioinformatics community is recognizing the value of MapReduce-style frameworks to achieve greater performance for their data processing tasks.

3.6 Conclusion

As the price of short-read sequencing continues to drop, the need to efficiently and accurately process the resulting genome data becomes ever greater. In order to produce accurate variant calls, it is crucial to align short reads without making errors. As part of our efforts working on the SNAP alignment algorithm [139], we gained valuable insight into what makes alignment difficult, and we learned that the main factor is the repetitive structure of the genome.

In this chapter, we presented detailed information about the repeats in the genome, which comprise about 50% of the genome's sequence. They can be microsatellites of just a few bases, or they can be large structural variants ranging up to hundreds of thousands of bases. They can occur right after each other, and they can be spread throughout the genome. Therefore, it is unsurprising that they would present different degrees of difficulty for alignment algorithms.

In order to improve alignment accuracy, we focus on the subset of repeats that make alignment difficult by taking the perspective of the short reads in order to locate repeats in the genome. We represent the genome as a graph, where each node is a substring of the same length as the reads and an edge connects each pair of similar substrings, and our goal was to find the connected components of this similarity graph. Due to the large size of the graph, our initial approach based on hashing proved unwieldy.

We therefore developed the SiRen algorithm, which is based on the well-known union-find algorithm. In order to apply the union-find algorithm to our problem, we had to augment the algorithm with two important optimizations: leveraging an index to reduce the number of comparisons we had to perform to locate the graph's edges, and partitioning the graph so we could parallelize the execution. We also had to employ several implementation tricks to reduce the runtime and memory usage, such as using Spark accumulators to incrementally update the global clusters with per-partition results. Ultimately, we were able to develop a tool that locates the similar regions in a practical amount of time and can be deployed in a variety of computational settings.

Chapter 4

Validating the Similar Regions

4.1 Introduction

In Chapter 3, we presented our method for locating the similar regions. Since our technique is completely unsupervised, it is important to evaluate the quality of the clustering. One approach is to consider cluster metrics such as similarity within clusters and separation between clusters. We begin this chapter, in Section 4.2, by presenting some basic cluster metrics. We also computed more sophisticated values that are not included in Section 4.2. However, just by looking at the metrics, it is difficult to know whether the clustering is really capturing an interesting signal in the data.

Since we obtain little insight from the metrics of Section 4.2, we use a more application-oriented strategy for judging the quality of the clustering. Recall that our main argument in the previous chapter, detailed in Section 3.2 when we discussed our experiences working with alignment algorithms, is that the similar regions make alignment difficult. This in turn leads to alignment errors, which then cause variant calling errors. We reasoned, therefore, that a clustering validation that demonstrates this chain of events, from alignment to variant calling, would provide better insight into the clustering "signal" than cluster metrics alone.

The first goal of our validation is to show that the similar regions make alignment difficult. Thus, we begin the discussion of our validation in Section 4.3 by exploring whether alignment errors are primarily located in similar regions. As part of these experiments, we develop a quantitative notion of alignments being easy or hard based on consensus among four alignment algorithms. We show that the vast majority of hard alignments are in similar regions: over 90% of such alignments from simulated data and over 80% of such alignments from real data.

Ultimately, though, we only care about alignment errors if they cause mistakes in variant calling. That is, if variant calling tools managed to produce the correct results despite the presence of alignment errors, the faulty alignments would not be a cause for concern. Therefore, we also designed a set of experiments leveraging the SMaSH benchmarking suite [125] to show that variant calling is more error-prone in the similar regions than in the unique

regions. We use three datasets, including two human datasets and one mouse dataset, which comprise both simulated and real data.

Section 4.4 presents the variant calling experiments. We consider both precision and recall for two popular variant callers, mpileup and GATK, with calls separated into SNPs and indels. We find that both tools performed significantly worse in the similar regions, particularly regarding the recall, where the accuracy in the similar regions is typically at least twenty percentage points lower than in the unique regions. This discrepancy in accuracy not only confirms our hypothesis that the similar regions complicate variant calling; since it persists in both the human and mouse genome experiments, it also suggests that our findings will hold in non-human genomes as well.

We can also leverage our validation in order to choose our algorithm's parameter, the merge distance h. Looking at the results from both the alignment and variant calling portions of our validation, as well as the fraction of the genome in similar regions at each merge distance, helps us choose the merge distance. Our criteria include minimizing the amount of the genome in similar regions while maximizing the amount of alignment and variant calling errors that they can explain. To put it another way, we want to achieve a strong correlation between the similar regions and the regions that make alignment and variant calling difficult, so the similar regions are not bloated with easy regions. The two criteria are related, because smaller similar regions will likely correlate better with the hard regions, but having small similar regions does not necessarily imply that they are hard. Thus, we conclude our discussion of the validation with comments about our selection of h. In Section 4.5, as a segue to Chapter 5, we also argue that the similar regions are amenable to specialized techniques for improving variant calling accuracy.

We conclude the chapter in Section 4.6 with related work. In the previous chapter, we mentioned some existing efforts to locate redundancy in the genome. The output of these efforts is often called a *blacklist*, since the regions identified as redundant are often excluded from genome analyses. In this chapter, we perform a quantitative comparison between the similar regions and some of the other blacklists. SiRen and GEM have the most significant overlap since they are closest in size, but no existing blacklist finds the same regions that we do. SiRen also is unique, to the best of our knowledge, in that it provides structure rather than just a binary indicator for each position about whether or not it is on the blacklist. As we will discuss in the next chapter, this information is critical when it comes to improving variant calling accuracy in these regions.

4.2 Basic Cluster Metrics

Before we begin our discussion of the metrics, Figure 4.1 describes our output format. The file begins with a one-line header that provides the number of substrings clustered, the number of clusters found (excluding singletons), the substring length, and the merge distance. Of course, the number of substrings is equal to $N - R$, where N is the size of the genome and R is the read length, which we use for the substring length. Since $R \ll N$, there are roughly

```
numSubstrings numClusters substringLength mergeDistance
clusterSize clusterMember1 clusterMember2 ...
```

Figure 4.1: SiRen output format.

N substrings, or about 3.1 billion. Following the header, there is one line per cluster. Each cluster's line begins with the number of substrings in the cluster, followed by each member of the cluster. We refer to cluster members by their initial position. For example, if a substring's endpoints are p and $p + R$, we would refer to the substring as p in the file. We sort the clusters from largest to smallest.

Recall that we refer to the merge distance, or Hamming distance threshold, as h. Since it is an important element of the following discussion, we will provide more intuition about what the Hamming distance represents. Because it is Hamming distance, which allows only substitutions, rather than full edit distance, which also allows insertions and deletions (indels), it is somewhat like a SNP between two substrings. (See Chapter 2 for an explanation of SNPs and indels.) This characterization is imperfect, however, in that a SNP is a difference between the reference and the sample DNA, whereas when we measure the Hamming distance between two substrings, we are comparing portions of the reference with other portions of itself. However, despite being inexact, the mental model of a SNP is useful for thinking about the Hamming distance. For example, $h = 2$ is akin to allowing two SNPs between the two substrings being considered.

Based on our experiences with alignment, which we describe in Section 3.2 of the previous chapter, we chose to let h vary between one and ten. Recall that we want to find regions that are similar to each other from the perspective of an aligner, since we are targeting alignment-based read processing pipelines. Based on our experience with SNAP, the vast majority of reads are aligned with less than five edits from the reference genome. To provide sufficient flexibility, we chose ten as a generous upper bound for h. In Section 4.4, we provide empirical evidence of the appropriateness of this decision.

Figure 4.2 shows the file size of the output of SiRen for different merge distances between one and ten. The file size increases with the merge distance, starting with 1.6 GB for $h = 1$ and ranging up to 4.7 GB for $h = 10$. We are aware that a difference of a few gigabytes in file size may not seem like a compelling matter to report. However, the reason this is interesting will become clear when we compare this with the pattern of increase for the number of clusters given each merge distance.

Figure 4.3 gives us insight into why the file sizes differ as h increases. When $h = 1$, the number of clusters is approximately 33 million, and the number of clusters increases with h, ultimately reaching 43 million for $h = 10$. This trend is due to the fact that when h is large, more points get merged together, so more clusters are created. Thus, the file size increases with h.

An interesting observation from Figures 4.2 and 4.3 concerns the pattern of the increase

Figure 4.2: File sizes for the output of SiRen, by merge distance. Larger merge distances yield larger files.

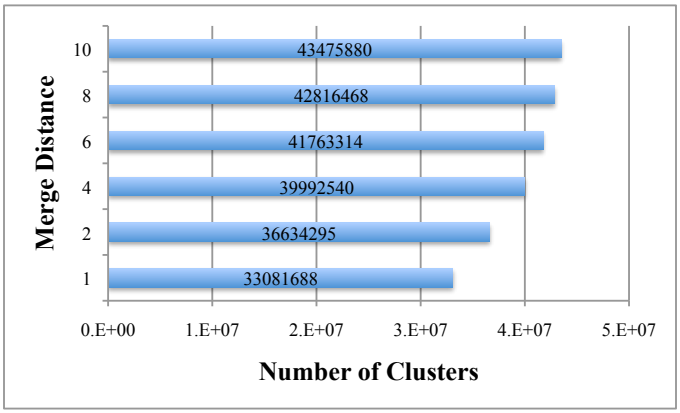


Figure 4.3: Number of clusters by merge distance. Larger merge distances yield more clusters.

in file size and number of clusters, respectively, with h. Figure 4.2 shows that the file size increases in a fairly linear manner with h, while Figure 4.3 shows that the number of clusters increases more with smaller values of h and then tapers off when h gets large. For large values of h, while an incremental increase in h has little impact on the number of clusters, it does cause more substrings to join those clusters, thus resulting in larger file sizes overall. We therefore infer that when we increase the merge distance to very large values, we stop creating new clusters, while the existing clusters become more bloated as we add on only slightly-related substrings. This outcome is undesirable since we end up with some large, diffuse clusters. Therefore, we can tell from this figure that we definitely should not increase h beyond 10, and in fact, we should prefer a small value of h to avoid cluster bloat.

In Figure 4.4, we provide a histogram of the cluster sizes obtained when $h = 1$. In Sections 4.3 and 4.4, we will discuss our reasons for selecting one as the merge distance. Note that the y-axis is on a log scale. The histogram shows that the vast majority of the clusters are

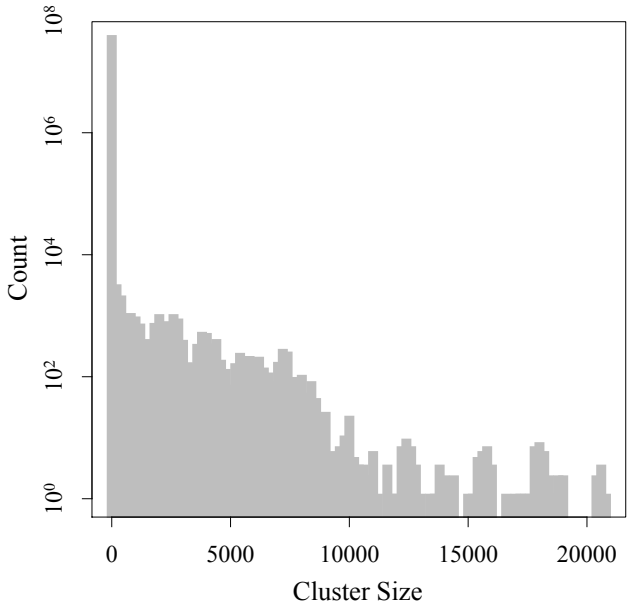


Figure 4.4: Histogram of cluster sizes for merge distance 1. Many clusters are small, while there is a long tail of large clusters.

small; in fact, many clusters are of size two. We will take advantage of this property of the clustering when we work to leverage the similar regions, which we will discuss in Chapter 5.

The figure also shows that there is a long tail, since there is a small number of large clusters. It would be possible to break up the large clusters by representing each cluster as a graph (in fact, it is a connected component of the original genome-wide similarity graph) and performing graph partitioning on the cluster's graph. We did not do this because, as we explain in Chapter 5, we primarily focus on small clusters, since they are the vast majority of clusters that we identify. However, graph partitioning would be an interesting direction to explore in the future.

We also examined some more sophisticated cluster metrics such as cluster diameter and inter-cluster distance. We gained little insight into the quality of the clustering from these metrics, however. In fact, the challenge inherent in using clustering, an unsupervised machine learning technique, is that by just using cluster metrics like these, it is very difficult to gauge whether or not the clusters identified truly correspond to important patterns in the underlying dataset.

Therefore, we decided to turn to a more application-oriented methodology to judge our clustering, as this is the recommended approach when metrics fail to provide sufficient insight [87]. Recalling our original argument that the similar regions are the reason for difficulties with alignment and therefore variant calling, our goal is to show this chain of events empirically. In Section 4.3, we discuss how we devised a microbenchmark based on alignment to evaluate whether our clusters correspond to the parts of the genome that complicate align-

ment. Since our main interest is variant calling, we present in Section 4.4 an end-to-end evaluation of variant calling accuracy in the similar regions vs. the unique regions, with the goal of demonstrating that variant calling is much more challenging in the similar regions.

4.3 Impact on Alignment

We begin by focusing on alignment, which is the first step of most variant calling pipelines. Our goal is to use alignment to evaluate the similar regions that we find via SiRen. To do so, we devised a method of generating labels for aligned data whereby we categorize each read on a spectrum from easy to hard. Our method is based on consensus across four alignment algorithms, or aligners, and yields labels with four gradations of easier or harder to align; *i.e.,* if all four aligners agree on a read's alignment location, we count it as easy, while if no aligners agree on a read's alignment location, we count it as hard. The other labels are obtained when two or three aligners agree. We selected this ensemble-style approach of label generation to avoid overfitting to any particular aligner; we picked these aligners because together, they are a good representation of the accuracy and efficiency spectrum.

Given a labeling scheme, we can now evaluate whether the regions we have obtained can explain our labeled data. We will judge the similar regions to be correlated with read difficulty if a high fraction of hard reads align to similar regions, while a low fraction of easy reads align to similar regions.

In what follows, we will show the correlation between aligner agreement and similar regions for both simulated and real data. The simulated data consists of 5 million pairs of reads generated from hg19 by the Mason simulator v.1.3.1 [44].[1] The real data consists of 2 million pairs of reads sampled from the Illumina Platinum Genome Project's NA12878 genome.[2] The length of both the simulated and real reads is 100 bases. We aligned each dataset with four aligners: Bowtie2 v.2.1.0 [66], BWA v.0.7.4 [74], NovoAlign v.3.0.3[3], and SNAP v.0.16alpha.6 [139]. We ran the aligners on EC2, using cc2.8xlarge instances, which have 16 cores and 60.5 GB of memory, and ami-c136bca8, which runs the CentOS distribution of Linux.

We first describe how we evaluated the aligners on the simulated data. After aligning the reads with all four aligners, we assigned a label to each read based on the number of aligners that agreed on its location, where we determine agreement within a tolerance of 100 bases. As this is a full read length, this is a rather generous measure of agreement, which fits our goal of avoiding unnecessarily calling reads hard if aligners differ by only a few bases. We then segmented the reads according to their labels, and then for each label, we assessed how many of the reads hit similar regions.

[1] Downloaded from `http://www.seqan.de/projects/mason/` and run with command line
`mason illumina -hn 2 -sq -mp -ll 375 -le 100 -n 100 -N 5000000 -o out.fq -s 0 hg19.fa`

[2] `http://www.ebi.ac.uk/ena/data/view/ERP001229`

[3] `http://www.novocraft.com/products/novoalign/`

Figure 4.5(a) shows the results. We observe that almost all the hard reads are in similar regions, while very few of the easy reads are in similar regions; therefore, we have achieved a close correlation between our labeled data and our similar regions (recall that the labels are defined above). We provide these results for two merge distances, 1 and 10. We also obtained results for 2, 4, 6, and 8; as they follow the same pattern, we omit them here. Since more of the genome is contained in similar regions when the merge distance is large (see Figure 4.6), it is logical that more reads of each label would fall in clusters with a larger merge distance. However, since we still obtain a high fraction of the harder reads in clusters using 1 for the merge distance, while including much less of the genome in similar regions, we prefer 1. We provide further discussion of selecting the merge distance in Section 4.4.

We carried out an equivalent analysis for the real data, to avoid overfitting to simulated data. The results, which are shown in Figure 4.5(b), are comparable to the results for simulated data. Again, reads that are hard to align are overwhelmingly in similar regions, while easy reads are not. We also observe the same trends regarding the merge distance.

These results demonstrate that one of the main reasons for alignment difficulty is the similar regions. We have shown that this holds for different aligners and for both simulated and real data, so it is unlikely to be an artifact of overfitting.

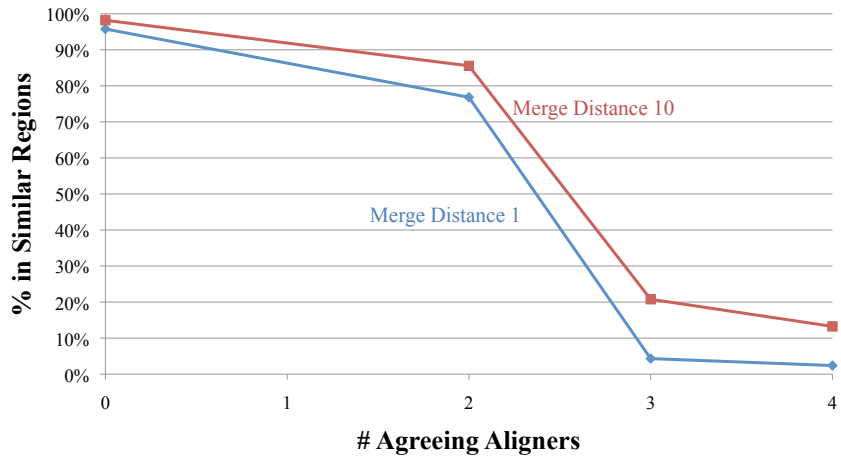
4.4 End-to-End Analysis: Variant Calling

We have shown that alignment, which is a key stage in the genomic processing pipeline, is strongly affected by the similar regions. However, this is an intermediate stage; ultimately, we care about variant calling, so in this section, we explore the impact of similar regions on variant calling accuracy. Our hypothesis is that due to alignment errors, variant calling will be more difficult in the similar regions than in the unique regions. In what follows, we validate this hypothesis using both simulated and real data.

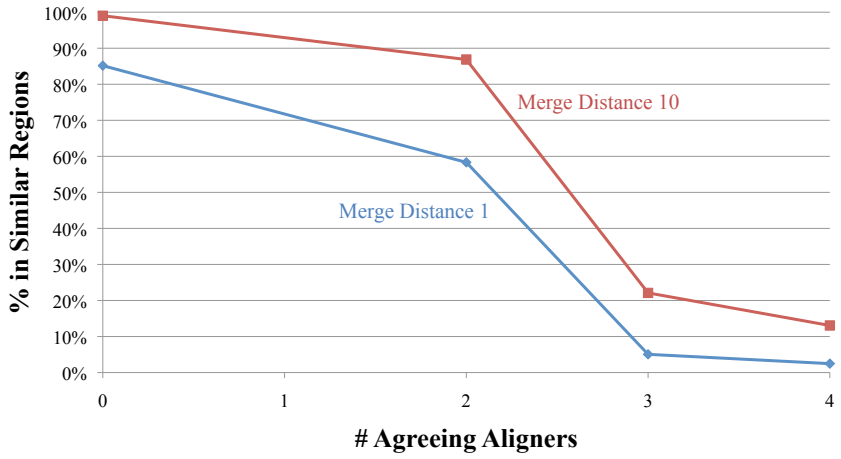
Experimental Data

For this analysis, we used the SMaSH benchmarking suite [124]. SMaSH provides read datasets for both synthetic and real data, as well as labeled variants obtained from orthogonal sequencing technologies. Leveraging this labeled data, SMaSH also offers an evaluation tool to determine the accuracy, measured in both precision and recall, for each dataset's variant calls. For both datasets, we consider both single nucleotide polymorphisms (SNPs) and short indels.

The synthetic human data from SMaSH is based on the variants detected in the Venter dataset [70]. These variants are inserted into the human reference genome hg19, and then simulated reads of length 100 bases with a coverage of 30× are produced via the simNGS tool [90]. Since all the variants are known, a comprehensive evaluation is possible; *i.e.,* we can report both precision and recall for the variant calls on this dataset.



(a) Simulated



(b) Real

Figure 4.5: Aligner agreement, correlated with cluster membership. For both (a) simulated and (b) real data, hard reads overwhelmingly coincide with similar regions, while easy reads fall in unique regions.

The real human data from SMaSH is 64× coverage reads for the NA12878 sample[4]. The reads are 101 bases long. Since this is real data, we do not know the types or locations of the true variants. However, SMaSH provides labeled variants from two SNP chips for this dataset. Note, however, that these labeled variants are not comprehensive; they are available only for the sections of the genome that the SNP chips were designed to interrogate. Therefore, we cannot determine the precision of variant calls, since precision depends on knowing the number of true positives. Hence, we report only recall for this dataset.

Experimental Setup

Our procedure for both datasets is as follows. Again using EC2, we begin by aligning each set of reads with BWA v. 0.6.1. We chose BWA due to its popularity among practitioners. Then, given the aligned reads, we produced variant calls with both mpileup, from SAMtools v.0.1.19 [76, 71], and GATK v.2.6 [27]. We chose these tools because mpileup is known to be a very simple variant caller, while GATK, considered the state-of-the-art, is widely used. We wanted to see how both a simple and a sophisticated tool would handle the similar regions.

Once we had obtained variant calls from both tools, we used bedtools,[5] a popular tool suite for executing set operations on the genome, to segment all three variant call format (VCF) files for each dataset (one for the true variants, from SMaSH; one from mpileup; and one from GATK) according to which variants appeared in similar regions and which appeared in unique regions. Then, we used SMaSH's evaluation tool to determine the accuracy of the called variants both in and out of the similar regions.

Results

We display the results separately for the synthetic and real datasets. We computed the precision and recall for each tool in the similar regions, the unique regions (the complement of the similar regions), and overall, *i.e.,* for the entire genome. Table 4.1 shows the results for the Venter dataset. For this dataset, which is synthetic, both mpileup and GATK perform worse in the similar regions than in the unique regions; this is more dramatic for the recall. Table 4.2 gives the results for the NA12878 dataset, which is the real dataset. For the real dataset, the difference in recall between the similar regions and the unique regions is much more pronounced. One likely cause that the accuracy suffers in the simiar regions is alignment errors, leading to reads being aligned to the wrong similar region, as we illustrated in the previous chapter, in Figure 3.2.

Therefore, these results validate our hypothesis that variant calling is much harder in the similar regions than in the unique regions. Though the overall precision and recall are high, we see that there is still a lot of work to do to achieve satisfactory accuracy in the similar regions. We suggest that developers of new variant callers concentrate on the similar regions rather than the unique regions, as the similar regions are clearly where the challenges lie.

[4]Available from `http://smash.cs.berkeley.edu/datasets.html`.

[5]`http://bedtools.readthedocs.org/en/latest/`

Tool	Variant	Merge Distance	Precision			Recall			Runtime (h)
			Similar	Unique	Overall	Similar	Unique	Overall	
mpileup	SNPs	1	93.8	99.0	98.7	78.6	98.5	97.0	2
		10	96.4	99.5		91.6	99.0		
	Indels	1	83.8	88.7	88.5	54.2	73.9	73.1	
		10	86.8	89.4		64.1	78.6		
GATK	SNPs	1	97.9	99.3	99.3	55.7	94.6	91.7	57
		10	98.4	99.5		80.0	96.0		
	Indels	1	86.1	91.6	91.5	67.5	90.5	89.5	
		10	90.0	92.1		85.9	91.6		

Table 4.1: Variant calling accuracy (precision and recall) for simulated data from the Venter genome. Note that the precision and especially the recall are worse in the similar regions than in the unique regions. This effect is more evident when the merge distance is 1.

Tool	Variant	Merge Distance	Recall			Runtime (h)
			Similar	Unique	Overall	
mpileup	SNPs	1	22.5	99.1	98.8	5
		10	82.8	99.3		
GATK	SNPs	1	22.9	99.2	98.8	86
		10	83.0	99.4		

Table 4.2: Variant calling accuracy for real data from the NA12878 genome. Precision cannot be reported because the dataset is sampled, and indels are omitted because SMaSH lacks indel validation data for this dataset. As with the simulated dataset, the recall is much worse in the similar regions than in the unique regions, and even more so when the merge distance is 1.

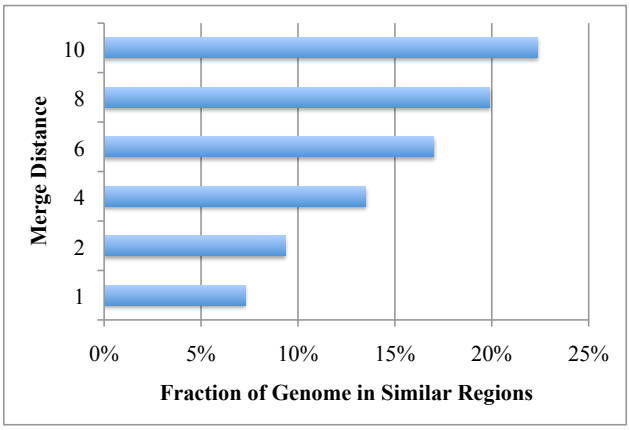


Figure 4.6: Percent of the genome in similar regions for different values of the merge distance. As we increase the merge distance, a greater fraction of the genome lies in similar regions.

Tables 4.1 and 4.2 also give us insight into how to choose the merge distance. The table presents results for merge distances 1 and 10; we obtained results for merge distances 2, 4, 6, and 8 as well, but we omitted them for the sake of brevity. We observe that when we increase the merge distance from 1 to 10, the accuracy in the similar regions increases substantially, while the accuracy in the unique regions is barely impacted. The reason for this is that when you increase the merge distance and therefore also increase the fraction of the genome that lies in similar regions, you include some easier regions. Likewise, when you restrict the merge distance, you achieve a tighter correspondence between the hard regions and the similar regions. Therefore, we prefer a smaller merge distance, in this case 1, since we greatly reduce the size of the similar regions—from 22% at merge distance 10 to 7% at distance 1 (see Figure 4.6)—without significantly impacting the performance of the variant calling algorithms in the unique regions. We will further discuss the desirability of limiting the size of the similar regions in Section 4.5.

Figure 4.7 illustrates a more subtle point. This figure compares, for our selected merge distance of 1, the fraction of variants (true, called by mpileup, and called by GATK) in similar regions. We observe that while Venter's true variants are distributed fairly evenly, NA12878 has few true variants in the similar regions. Since the Venter validation data is comprehensive while the NA12878 data is partial, we conjecture that the SNP chips used to obtain the NA12878 variants are biased against the similar regions, potentially because this data is more difficult to gather. Since SNP chips provide few variants in these regions, variant databases, which are largely based on SNP chips, likely also have relatively few variants in these regions.

Tables 4.1 and 4.2 make more sense in light of the distribution of SNPs in the two datasets. GATK is a very sophisticated tool that incorporates prior knowledge about where variants are probable based on databases of previously-detected variants such as dbSNP[6],

[6] `http://www.ncbi.nlm.nih.gov/SNP/`

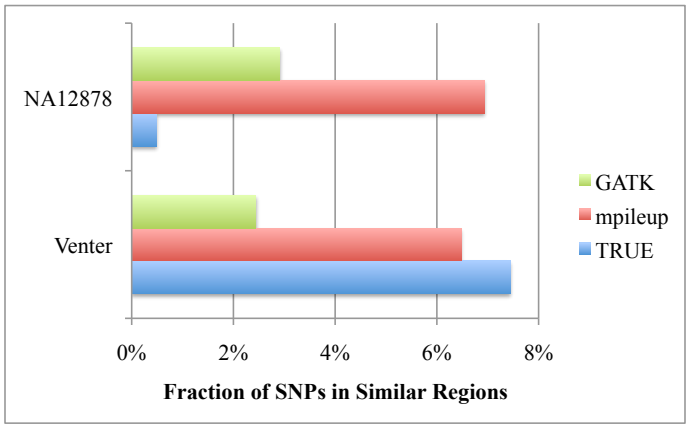


Figure 4.7: Fraction of SNPs in similar regions, for merge distance 1. For Venter, GATK vastly under-calls variants in similar regions. For NA12878, the true variants are underrepresented in similar regions.

while mpileup is relatively simple, relying only on the reads at hand. In the first table, we encounter the surprising result that with respect to recall, GATK underperforms mpileup on the Venter dataset. However, in the second table, we see that, in line with our expectations, GATK matches or exceeds mpileup's performance on the NA12878 dataset. Given that the similar regions are underrepresented in our validation data, and likely in other sources of variant data as well, GATK may be discounting SNPs that it identifies in these regions. This approach adversely impacts its accuracy in the comprehensive Venter dataset, while in the partial NA12878 dataset, its relatively few predictions are in line with the available validation data.

Variant Calling on Mouse Data

Since the SMaSH benchmark [124] also contains mouse data, we investigated whether our techniques would apply beyond human data. The mouse data has an attractive property in that it provides access to real short-read data along with a comprehensive set of validated variants. Thus, we are able to state accuracy in both precision and recall.

To repeat the variant calling experiment with mouse data, we had to run the similar region detection tool on the mouse reference genome. As with the human data, we tried a variety of merge distances to identify similar regions, aligned the mouse reads with BWA, and ran mpileup and GATK on the aligned reads. We then segmented the true, mpileup, and GATK VCF files according to the similar regions.

Table 4.3 shows the results for merge distance 1. The results for the other merge distances, as in the case of the human data, showed improved performance in the similar regions without significant change in the unique regions. However, as this improved performance comes at the cost of increasing the percentage of the genome in similar regions, we again identify 1

Tool	Variant	Precision			Recall			Runtime
		Similar	Unique	Overall	Similar	Unique	Overall	(h)
mpileup	SNPs	87.8	98.5	98.4	61.4	87.4	87.3	6
	Indels	60.4	83.2	83.1	39.3	81.3	81.1	
GATK	SNPs	80.9	98.0	97.8	67.7	95.2	94.9	107
	Indels	41.2	86.3	86.0	49.7	93.8	93.6	

Table 4.3: Variant calling accuracy (in both precision and recall) for mouse dataset, with merge distance 1.

as the optimal merge distance. Given our choice of merge distance, we observe that both GATK and mpileup perform worse, in both precision and recall, in the similar regions than in the unique regions. Therefore, we conjecture that our techniques apply not only to the human genome but to other genomes as well.

4.5 Discussion

Recall that our overall goal is not only to locate the similar regions but also to improve the accuracy of variant calling in those regions. To do so, we must deploy specialized techniques to run on the similar regions; we will discuss this in great detail in Chapter 5. In this section, we reflect on how the similar regions we identified facilitate this goal.

A good approach to partitioning the genome into similar and unique regions will have two characteristics. First, we should identify a small portion of the genome as similar, and second, the similar regions should contain many short segments, so that we can launch parallel tasks, where no task is prohibitively expensive. Regarding the first characteristic, consider that if we identify a large portion of the genome as similar, and we run the specialized algorithm on all the similar regions, we will not achieve much savings over running it on the entire genome. The second characteristic is important because consider that if we identify just a few, long similar regions, it will limit the degree of parallelism that we can achieve.

Let us consider whether a partitioning scheme based on the similar regions achieves our two criteria. First, regarding the fraction of the genome in similar regions, we see in Figure 4.6 that with merge distance 1, which is our selected merge distance, only 7% of the genome lies in similar regions. Thus, we could potentially achieve an order of magnitude savings over running a specialized technique on the whole genome.

Second, regarding the size and number of the partitions, consider Figure 4.8. We observe that for our desired merge distance of 1, even at the 99th percentile, the similar regions are only a few thousand bases long. In addition, Figure 4.9 shows that at merge distance 1,

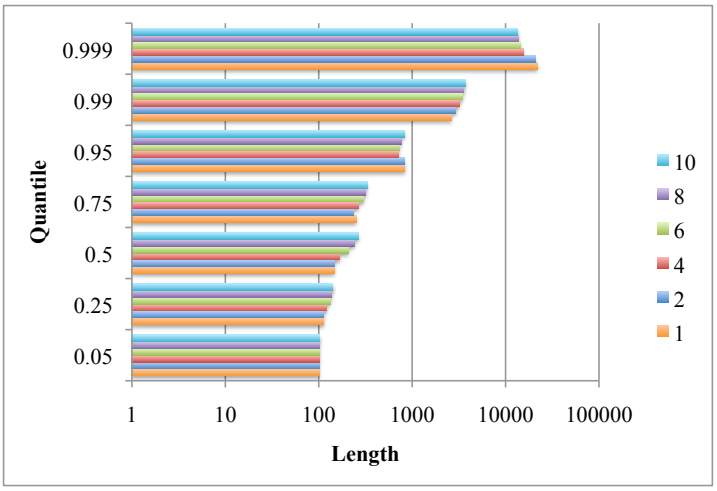


Figure 4.8: Quantiles of interest for lengths of similar regions, for different merge distances.

there are approximately 600,000 of these partitions. Since we have identified many short partitions, we can obtain a high degree of parallelism.

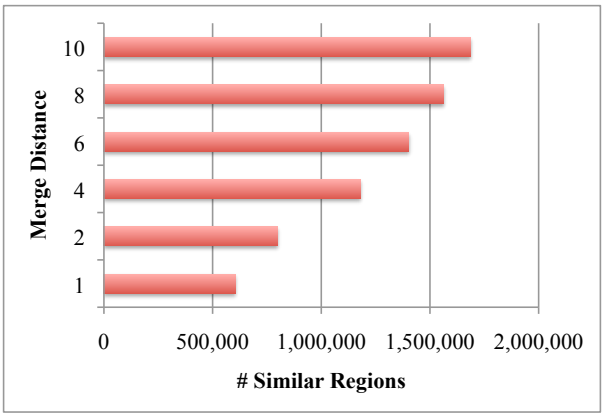


Figure 4.9: The number of similar regions, for different merge distances.

Therefore, we determine that the similar regions we have identified are amenable to our goal of developing parallelizable techniques for specialized processing to improve the accuracy of the similar regions. In the next chapter, we will build on this analysis to describe our techniques and demonstrate their efficacy.

4.6 Related Work

In this section, we describe three categories of related efforts to divide the genome into easy and hard regions, which often result in the "blacklisting" of a particular genome subset.

The partitioning could be based either on the genome itself or on the short reads from each sample. Moreover, we provide a quantitative comparison between various published genome blacklists and our proposed similar regions in Table 4.4. We also illustrate the relative sizes of the blacklists as well as the sizes of their overlaps in Figure 4.10.

The first class of efforts is based on the study of repeats in the human genome, often with the goal of understanding the evolutionary biology of the genome [2, 63, 132, 7, 103]. One of the most well-known repeat databases is Repbase [55], an inclusive set of repeats with sequences that span a broad range of lengths and allow a significant degree of divergence among members of the same family. Since Repbase[7] comprises 47% of the genome (see Table 4.4), however, it is ill-suited for the application of a specialized variant calling approach.

The second class of algorithms focuses on the task of identifying regions of the genome that are particularly challenging for variant calling. However, these approaches all employ an orthogonal strategy of relying on the short reads themselves to identify these difficult regions. Many of these strategies are complementary to our approach, identifying regions enriched with sequencing errors, *e.g.,* `GC`-rich regions, and/or sample-specific variation that could lead to low variant calling quality.

For instance, the accessible genome [1] aims to identify "accessible" regions for variant calling by aligning reads from many samples and developing heuristics primarily based on coverage to identify anomalous regions. The blacklist developed by the Pritchard Lab at Stanford[8] leverages a similar strategy to identify collapsed repeats that may hinder variant calling accuracy. The sizes of the inaccessible genome (!Accessible Genome) and the Pritchard blacklist (Pritchard) are given in Table 4.4.

The GEM suite [28] falls into the third class of methods and is the most closely related to our similar regions. GEM assesses the mappability of short reads by identifying how often each substring of length k occurs in the reference genome with up to m mismatches, and defining mappability as the inverse of this count. Hence, as in our work, GEM relies on the reference itself to determine the mappability of various regions, and the size of the GEM blacklist is comparable to the similar regions, as noted in Table 4.4.[9]

We build upon the preliminary studies in the GEM work, providing a thorough investigation of the impact of similar regions on variant calling and clearly demonstrating the need for alternative variant calling algorithms in these regions. Moreover, we provide more information than mere counts of a substring's neighbors; we provide the full set of connected components in the genome. Indeed, as we will discuss in Chapter 5, information about the structure of similar region families is crucial in order to disentangle the reads stemming from similar regions.

A notion of read quality scores, relevant to the alignment microbenchmark, was introduced in [75]; the authors named their metric mapping quality, or MAPQ. The MAPQ score, assigned to reads during the alignment phase, is meant to express the probability that the

[7]We used the version of Repbase distributed via RepeatMasker open-3.3.0 - RepeatLibrary 20120124, available from `http://www.repeatmasker.org/`.

[8]`http://eqtl.uchicago.edu/Masking/readme`

[9]We used $k = 100$, $m = 1$, and disabled the approximation.

Blacklist	Overlap Size (% of Genome)	Total Size (% of Genome)
Repbase	169 million (5.4%)	1.48 billion (47.3%)
!Accessible Genome	156 million (5.0%)	472 million (15.1%)
GEM Mappability	190 million (6.1%)	266 million (8.5%)
Pritchard	1.46 million (0.046%)	6.47 million (0.21%)

Table 4.4: Overlap between the similar regions and various blacklists, listed in descending order of size. Shown is both the size of the overlap and the size of the blacklist, in bp and as a fraction of the genome.

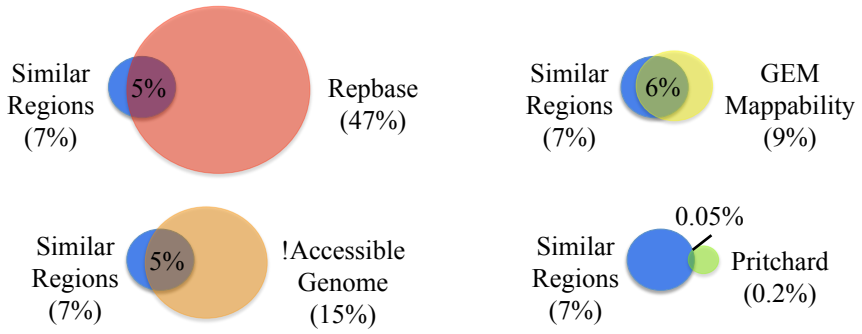


Figure 4.10: Venn diagrams comparing the similar regions to various other blacklists.

alignment is correct. Thus, a high score indicates that the aligner is confident about its placement of the read, and a low score indicates that it is not. MAPQ is a useful way for the aligner to propagate its uncertainty to downstream stages of the pipeline. Since it is a probability, though, some high MAPQ reads will be errors, while some low MAPQ reads will be correct. Thus, any downstream stages that attempt to leverage MAPQ as a signal of alignment correctness need to treat it as a probability rather than as an absolute threshold. However, this probability is potentially noisy since MAPQ is often poorly calibrated; *i.e.,* the probability expressed by the MAPQ that the alignment will be correct does not necessarily correspond to the actual percentage of correct reads with that MAPQ. Many practitioners, therefore, do not fully trust it.

Low MAPQ could be considered a proxy for hard reads; however, it is not as useful as the similar regions. First, low MAPQ values could result from other reasons besides a read mapping to multiple locations in the reference; they could also result from sequencing errors that lower the quality of the read itself. Furthermore, MAPQ is on a per-read basis, whereas our similar regions give insight about the genome itself. Therefore, we only need to compute them once rather than for each sample.

Other work in the literature pertains to our proposed strategy of developing specialized

techniques for variant calling in the similar regions. The GATK [27] has also introduced a notion of treating different regions of the genome in different ways. They do this with their ReduceReads module.[10] The idea is to compress the BAM file, which is the file containing the reads and their aligned locations, in areas that are redundant and keep only information necessary for variant calling. This approach is mostly focused on reducing storage needs rather than computational costs. In addition, it is sample-specific, *i.e.,* it must be redone for each processed genome, whereas our similar regions are functions only of the reference and can therefore be computed once and for all before processing any samples. This compression technique is orthogonal and complementary to our work, in that we may also benefit from compressing the reads, which can take a few hundred gigabytes to store.

Another project in the same spirit as GATK's ReduceReads is CAGe, which stands for Changepoint Analysis of Genomes [10]. They also process the aligned read set rather than the reference genome; hence, like ReduceReads, CAGe is sample-specific, so it must be re-run on each new sample genome. This approach goes beyond ReduceReads, however, in that it not only aims to reduce storage costs but also to reduce the runtime of variant calling. Their key idea is to use changepoint detection to recognize regions in the genome that are variant-dense and therefore more complicated for variant calling. Then, they deploy a simple variant caller in the regions with few variants, while reserving a more accurate and therefore more computationally expensive variant caller for the regions with many variants.

While CAGe designates a small portion of the genome as variant-dense, the speedups are quite modest. For example, in one single-chromosome experiment, while they designated 4% of the genome as variant-dense, therefore requiring the sophisticated caller, they achieved only a 1.4× speedup over running the tool on the whole chromosome. Therefore, more work is necessary to realize the project's potential for runtime reduction. Should this be accomplished, this approach would be complementary to ours; an efficient system combining SiRen and CAGe would first use SiRen's similar regions to partition the genome into similar and unique regions. Then, we would deploy CAGe on the unique regions, while using special techniques on the similar regions.

4.7 Conclusion

In Chapter 3, we discussed our efforts to identify the similar regions. Of course, since we employed an unsupervised algorithm, a natural question is whether the similar regions that we found are actually interesting. Our motivation was to find similar regions that correspond to difficult alignments and therefore cause variant calling errors. We developed an algorithm and build a tool to find the similar regions, so now we want to know if they meet our goal.

The two approaches that are usually used to validate a clustering result are computing metrics of the clustering itself or considering an application that leverages the clustering.

[10] `http://www.broadinstitute.org/gatk/gatkdocs/org_broadinstitute_sting_gatk_walkers_compression_reducereads_ReduceReads.html`

Looking at metrics alone was somewhat interesting, but it did not yield much insight regarding our motivation. Therefore, we decided to turn to an application-oriented validation that directly answers the questions we had in mind.

First, we designed a set of experiments to determine whether alignment is affected by the similar regions. We developed a method to label each read as easy or hard to align, and we found that hard reads are almost always in the similar regions. We also performed a set of end-to-end experiments to determine whether these alignment difficulties percolated through the pipeline to cause variant calling errors. We found that, indeed, the two popular variant calling tools we considered struggled much more in the similar regions than in the unique regions. We were thus able to quantify the phenomenon that we talked about earlier, in Chapter 3, where similar regions cause difficulties in processing short reads accurately. Hence, a contribution of this chapter is that we have provided motivation that future work on variant calling should be focused on the similar regions, since they present a significant obstacle to accuracy, and we will build upon this finding in Chapter 5.

We ended the chapter by comparing the similar regions with several existing blacklists. Unsurprisingly, the similar regions have the most in common with GEM, which has a similar size. The main advantage of the similar regions over the existing blacklists, though, is that they capture the structure of which regions are related to which other regions. In the next chapter, we will discuss our efforts to leverage this structure to improve the accuracy of variant calling in the similar regions.

Chapter 5

Leveraging the Similar Regions

5.1 Introduction

This chapter is the culmination of our discussion about the usefulness of our similar regions. We began by introducing the SiRen algorithm by which we located the similar regions (Chapter 3). Then, we demonstrated that the similar regions are correlated with alignment and variant calling (VC) errors (Chapter 4). We will now present evidence that the similar regions not only identify difficult regions but also provide a platform for improving VC accuracy in these regions.

In Chapter 4, we treated the similar regions as a binary list, where the only information the list provides is whether a region is similar to another or unique. Then, we analyzed in detail how alignment and variant calling suffered in the regions indicated as similar. This exercise was useful for validating that the results of our unsupervised algorithm captured the information about alignment difficulty that we expected them to reflect. However, GEM [28] is also a binary list of similar sequence. Therefore, our goal is to show that we provide additional value over GEM's list with our similar regions.

In this chapter, we demonstrate the usefulness of our similar regions' structure, in addition to their binary list, through a case study. First, we argue that the difficulties we encounter in alignment and variant calling in the similar regions are not solely due to those regions having more complex variants that are more difficult to call. In fact, the prevalence of alignment errors in the similar regions is a significant cause of the difficulty variant callers face in those regions. Then, we show that for many alignment errors in the similar regions, the correct locations can be found elsewhere in the corresponding families. When these errors are restored to their correct locations, the VC accuracy improves.

We begin the chapter in Section 5.2 by discussing how we form groups of similar regions. Then, in Section 5.3, we show that, modulo some effects from structural variants, VC in the similar regions is of comparable difficulty to VC in the unique regions. It is therefore not true that the poor VC accuracy we saw in Section 4.4 results solely from the similar regions being inherently more difficult to call; in fact, alignment is a major reason for the

poor VC accuracy in the similar regions. We continue in Section 5.4, demonstrating that the similar region families we created in Section 5.2 contain the information needed to repair many of the alignment errors we encounter in these regions. In Section 5.5, we provide some initial exploration of how to build a classifier to recognize alignment errors automatically, so our case study can be extended to create a fully-functioning system. While building a classifier remains future work, we have identified some intuitive (though so far unsuccessful) approaches, and we also give our perspectives on promising directions to explore next. We conclude in Section 5.6 with related work on variant calling in the face of genome redundancy.

This chapter thus completes our argument about the importance of the contribution we have made by identifying the similar regions. Not only do our similar regions indicate difficult regions in the genome, but they also have a structure that contains valuable information for correcting alignment errors. Recall that in Section 4.5, we discussed how the similar regions are suitable in terms of their number and size for the development of specialized techniques that apply to these regions. In this chapter, we give a more concrete description of what those specialized techniques should be. Our similar regions, plus our case study, lay the groundwork for future efforts to build an end-to-end system that can be inserted into a VC pipeline.

5.2 Similar Region Families

In this section, we provide more detail about the structure of our similar regions. After briefly summarizing how the read-length substrings we clustered relate to the similar regions, we present our algorithm for determining which similar regions should be grouped together. We also discuss the characteristics of the similar region groups that we identify via our algorithm.

Figure 5.1 illustrates the relationship between substrings and similar regions. Recall that each similar region is created by merging overlapping substrings. In Figure 5.1(a), we show the output of the SiRen algorithm, which we presented in Chapter 3. The output is the connected components of a graph, where each node is represented by a substring whose length is the same as the reads that we intend to process, and edges connect two substrings if the Hamming distance between them is below a threshold. Each connected component receives a unique ID.

Figure 5.1(b) shows how the substrings identified by SiRen relate to the similar regions. The long blue line represents the reference genome, and the short colored lines that appear underneath the reference represent the substrings. Note that the substrings all have the same length. Each substring appears below its original location in the genome, and its color corresponds to the component to which it belongs, shown in Figure 5.1(a). The similar regions are the red lines that lie on the reference genome and are created by merging overlapping substrings. The list above each similar region gives the component IDs that represent its underlying substrings. For example, the first similar region contains one substring from component 1, one from component 2, and one from component 6.

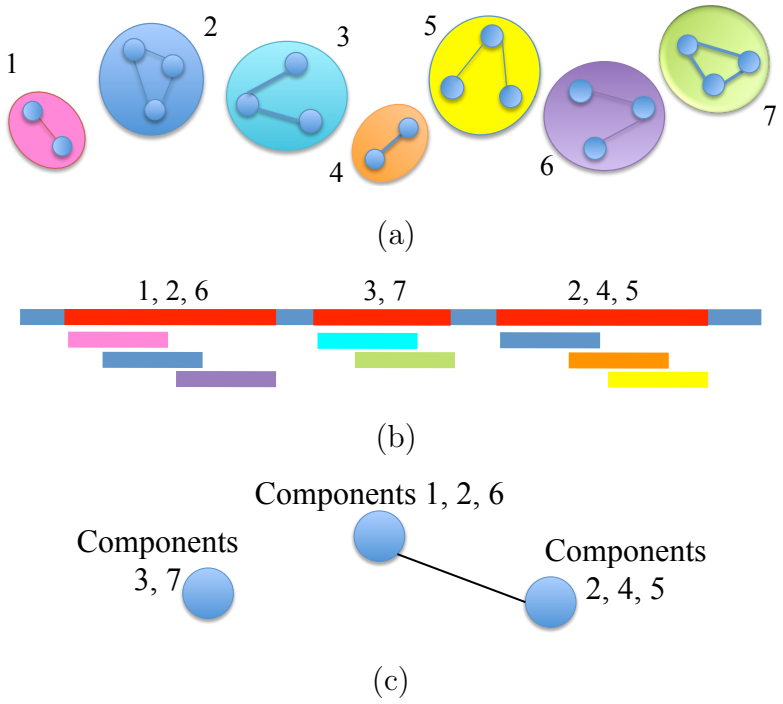


Figure 5.1: Similar region families. (a) Each group of substrings has a unique ID. (b) Each similar region is made up of substrings, which in turn belong to connected components. (c) Each similar region is represented as a list of its substrings' component IDs.

As we described in Chapter 3, SiRen tells us which substrings should be clustered, based on whether they belong to the same connected component of the genome similarity graph. What is missing, at this point, is knowing which similar regions should be grouped together. We define a *similar region family* as a group of similar regions, where for each region in the family, the family contains at least one other region with a shared substring component ID. Looking again at Figure 5.1(b), we see that the first similar region contains a dark blue substring from component 2, as does the third similar region. Thus, by our definition, the first and third similar regions should belong to a family.

To algorithmically locate the similar region families, we use an approach very similar to SiRen. We start with a new graph, in which the nodes are similar regions, represented as a list of the component IDs to which its substrings belong, and the edges connect similar regions that share at least one component ID. Figure 5.1(c) illustrates our graph representation. As mentioned above, the first similar region (represented as "Components 1, 2, 6") and the third similar region (represented as "Components 2, 4, 5") are connected by an edge since they share component 2. The second region ("Components 3, 7") is not connected to either of the other two nodes, since it does not share a component ID with either one.

Like in SiRen, our goal is to identify connected components of the graph just described; each such component will correspond to a similar region family. To identify the components,

we again employ the union-find algorithm. As a preprocessing step, we convert the output of SiRen, which we describe in Section 4.2, to similar regions. To do so, we first merge the overlapping substrings to create similar regions; then, we represent each similar region as a list of the component IDs corresponding to its substrings. Algorithm 4 describes this step.

Algorithm 4 Merging substrings to create similar regions

R = read length, G = genome size
Initialize: *posToComponentID*, map from substrings' initial positions to component ID, empty
Initialize: genome bitmap g, binary array of length G, all 0
Initialize: *similarRegions*, list of sets, empty
for all components c **do**
  for all positions $p \in c$ **do**
    $posToComponentID \mathrel{+}= (p, c)$
  end for
end for
for all positions $p \in G$ **do**
  if $posToComponentID.get(p)$ **then**
    Set $g[p, p + R)$ to 1
  end if
end for
for all $b \in g$, where b is a block of 1s **do**
  $similarRegions \mathrel{+}= \{\forall p \in b : posToComponentID.get(p)\}$
end for
return *similarRegions*

Once we have the similar regions in this representation, we again use a Spark-based implementation of the union-find algorithm, adapted to our new data format and similarity function. The new similarity function computes the number of components in common between two similar regions, which is analogous to our use of Hamming distance to compare two substrings in SiRen.

Initially, each similar region is assigned to its own cluster. As in SiRen, we partition the genome in a grid fashion. We also maintain one global union-find data structure, which can keep track of all the clusters in the genome. Via Spark, we execute in parallel on the grid cells. Algorithm 5 gives details about how we find the clusters within a single grid cell. We then update the master with each grid cell's results in the same manner as for clustering substrings, which we explained in Chapter 3, in Algorithm 3.

For each grid cell, we index the similar regions in its row range. The index is a map from component IDs to the similar region IDs that contain substrings from those components. Then, we scan over each similar region in the column range. Given a similar region, for each of its component IDs, we look up the ID in the index and then merge the current similar region with any regions that share that ID, if they do not already belong to the same

```
numSimilarRegions numFamilies substringLength minIntersectSize
familySize (region1Start,region1End) (region2Start,region2End) ...
```

Figure 5.2: Output format for similar region families.

cluster. Then, after we finish executing on the grid cell, we use Spark accumulators [122] to update the master union-find data structure with the cell's incremental results. When we have executed a task for each grid cell, the master has the results for the genome.

Algorithm 5 Union-find on similar regions, for a grid cell

```
Input: grid cell g with row range r and column range c
Initialize: parent_g(i) = i = family of the ith similar region, where length(parent) = |r|+|c|
Create index I, from component IDs to similar regions in r that contain those components
for all similar regions S_R in r do
  for all component IDs j in S_R do
    hits = I(j)
    for all k ∈ hits do
      Update parent_g(k) = parent_g(j)
    end for
  end for
end for
return parent_g
```

Figure 5.2 describes the format in which we output the similar region families. The header line gives the number of similar regions that we grouped, the number of families we created, as well as the substring length and the number of component IDs that we required in common in order to group two regions. Then, we have a line for each family. First, we output the family size, and then we give the endpoints of each region in the family. Note that we must give the regions' endpoints, rather than only their start positions, since similar region lengths can vary. We sort the family lines by number of members, from largest to smallest.

For this case study, we chose to run our union-find algorithm on the similar regions for a single chromosome, chr22. We made this decision because one chromosome's family data is sufficient to demonstrate the points we wanted to make, which we will explain in detail in the rest of the chapter. Since our implementation is distributed, however, it would be straightforward to generalize it to the whole genome.

Figure 5.3 is a histogram of the family sizes obtained via this algorithm. We observe a similar pattern as for the sizes of clusters of substrings, which we presented earlier in Figure 4.4. The vast majority of families are small, and a small number of families are large. We will exploit this pattern when we discuss our automatic realignment technique in Section 5.4.

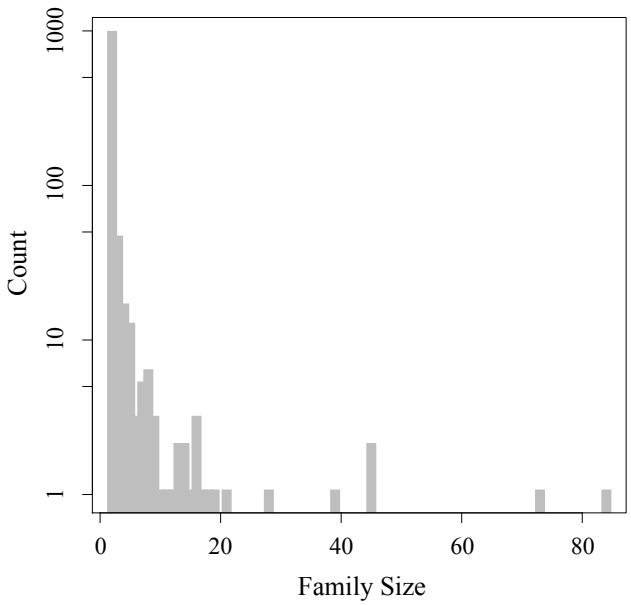


Figure 5.3: Histogram of family sizes. Given our parameter selection, most families are small.

In our discussion above, we mentioned that we draw an edge between similar regions having at least one component ID in common. Of course, we could require similar regions to share more components. However, since even with our simple similarity function, a large majority of our families were small, we elected not to explore a stricter function.

5.3 Oracle Alignment

The goal of this section is to establish whether alignment errors are the cause of the difficulty variant callers face in the similar regions, which we detailed in Section 4.4, or whether variant calling is inherently more difficult in the similar regions than in the unique regions. In Section 3.2, we presented a hypothesis that alignment errors are the culprit, rather than inherent difficulty. In what follows, we will present our findings showing that the cause of difficulty is a combination of the two factors. Since structural variants are more prevalent in the similar regions, they do make variant calling more difficult there. However, even if structural variants' effects are removed, variant calling accuracy is still worse in the similar regions. If we correct all alignment errors, variant calling accuracy is almost the same in the similar and unique regions.

In order to test our hypothesis, we chose to work with simulated data, so that we could identify the alignment errors and locate the true variants. As in our evaluation in Chapter 4, we obtained simulated data, including reads and a true VCF file, for the Venter genome from the SMaSH benchmarking suite [125]. We used BWA v.0.6.1 [74] to align the reads,

Region Type	Error Rate
Similar	18.3%
Unique	1.2%
Overall	3.7%

Table 5.1: Alignment error rates, in the similar regions, unique regions, and overall.

and we used mpileup [71] from SAMtools v.0.1.19 to perform variant calling. In this section, we focused solely on SNPs, rather than on both SNPs and indels as in Chapter 4. Thus, we will use the terms "variant calling" and "SNP calling" interchangeably throughout the rest of the chapter.

In order to remove the effects of alignment errors, we created a simple tool that we call the *oracle aligner* to align the reads. Since we used simulated data, we know the true location of each read; this information is encoded in each read's ID. The oracle aligner takes a read from a FASTQ file as input and outputs a SAM line for the read, where the position is set to be the read's true position. Recall that we discussed the FASTQ and SAM file formats in Section 2.4.

Table 5.1 gives the alignment error rates we encountered in the similar regions, unique regions, and overall. Note that we only count an alignment as an error if its aligned location is more than 100 bp away from the simulator's reported location. We also used this definition in the alignment section of our evaluation, described in Section 4.3. Since the error rate in the similar regions is much higher than in the unique regions (and overall), we would expect a large improvement in SNP calling accuracy when using the oracle aligner.

To evaluate the impact of the oracle alignment on SNP calling accuracy, we used mpileup to call SNPs on the oracle aligner's SAM file. Table 5.2 shows the results. For convenience, the Δ columns show the difference in accuracy between the similar and unique regions, for both precision and recall. The line with the alignment type "Original" shows the baseline, which is obtained using the BWA alignment. As we showed in Section 4.4, the accuracy is much worse in the similar regions than in the unique regions, with respect to both precision and recall. The line with the alignment type "Oracle" gives the SNP calling accuracy obtained using the oracle alignment. While the recall in the similar regions dramatically improves, from 75.2 to 95.5, the precision improves only modestly, from 81.3 to 84.0. We also note that even with the oracle alignment, the difference in precision between the similar and unique regions is substantial: 84.0 vs. 97.3. Therefore, the alignment errors are not the only cause of the difference in accuracy.

We noticed that many of the erroneous SNP calls that mpileup made in the similar regions, even with the oracle alignment, were around structural variants (SVs), which we defined in Section 2.7. Recall that we know the locations of Venter's structural variants, since this is simulated data. A natural question is whether SVs affect similar regions more

Alignment Type	Precision			Recall		
	Similar	Unique	Δ	Similar	Unique	Δ
Original	81.3	97.4	16.1	75.2	95.3	20.1
Original - SVs	89.2	98.5	9.3	76.0	95.4	19.4
Original - (SVs + flank)	93.6	99.0	5.4	80.2	95.9	15.7
Oracle	84.0	97.3	13.3	95.5	96.7	1.2
Oracle - SVs	95.0	98.6	3.6	96.0	96.8	0.8
Oracle - (SVs + flank)	98.1	99.3	1.2	96.9	97.1	0.2

Table 5.2: SNP calling accuracy (precision and recall) in the similar and unique regions, varying the alignment type. By masking out the SVs and their flanks, the accuracy between the similar and unique regions converges.

Region Type	Overlap Size	Region Size	Overlap Fraction
Similar	25,800	1,680,000	1.5%
Unique	107,000	49,600,000	0.2%

Table 5.3: SV overlap with similar and unique regions. SVs are more prevalent in similar regions.

than unique regions. Therefore, we decided to investigate their prevalence in each region type. Table 5.3 shows our findings. The SVs are indeed more prevalent in the similar regions, where they account for 1.5% of the sequence, compared with making up only 0.2% of the sequence in the unique regions. Therefore, it would make sense if the SNP calling accuracy in the similar regions were adversely effected to a greater extent than in the unique regions.

Based on these findings, we decided to isolate the influence of the structural variants in order to determine if they were having a disproportionate effect on SNP calling in the similar regions. To do so, we used bedtools[1] (also mentioned in Section 4.4) to mask out the structural variants from the similar and unique regions. We also created a mask where we not only excluded the structural variants but also a read-length flank on either side; recall that in our experiments, the read length is 100 bp. We chose to explore masking the flanking sequence because of our observations that mpileup called SNPs incorrectly around structural variants. Then, we analyzed the SNP calling accuracy in the remainder.

Table 5.2 above displays the results. We will comment primarily on the lines where both the SVs and their flanks are masked; the effects we will explain are also present when only

[1] `http://bedtools.readthedocs.org/`

the SVs are masked, but to a lesser extent. First, we focus on the line where the alignment type is "Original - (SVs + flank)." Comparing this line to the "Original" alignment type, we note that the precision has increased from 81.3 to 93.6 in the similar regions, while the recall has increased from 75.2 to 80.2. This result confirms our intuition that the structural variants cause trouble for mpileup in the similar regions. However, if we look at the difference between mpileup's accuracy on the "Original - (SVs + flank)" line between the similar and unique regions, given in the Δ columns, we note that a large difference in accuracy persists (especially with respect to the recall), even with the SVs and their flanks removed. Therefore, the difference in accuracy between the similar and unique regions is not solely due to the presence of SVs.

Next, we focus on the line where the alignment type is "Oracle - (SVs + flank)." Looking at the Δ columns, we note that the difference in the SNP calling accuracy between the similar and unique regions is quite small. Thus, by correcting the alignment errors, we are able to raise the SNP calling accuracy in the similar regions to the level obtained in the unique regions.

Thus, both alignment errors and structural variants contribute to making variant calling more difficult in the similar regions. We showed that alignment errors and structural variants disproportionately affect the similar regions. We also showed that solely repairing alignment errors, or solely masking out the SVs, cannot yield comparable SNP calling accuracy between similar and unique regions. However, recall that what we wanted to repudiate was the assertion that SNP calling is inherently more difficult in the similar regions. If this assertion were true, masking out the SVs should have been enough to eliminate the accuracy differential. We were only able to achieve comparable accuracy by both masking the SVs and correcting the alignment errors. Therefore, we conclude that modulo SVs, alignment errors are the culprit for reduced SNP calling accuracy in the similar regions.

5.4 Repairing Alignment Errors

Given that alignment errors in the similar regions result in poor VC accuracy, we will now explore the advantage that the structure of our similar regions provides over simply having the similar regions as a binary list, like GEM [28]. In fact, because of the structure that we have identified, we have developed an algorithm for realigning alignment errors. We show that this method is successful the majority of the time: we can correct 55% of errors in size-2 families, and in families with at least five variants, we can correct 77% of errors.

As we mentioned in Section 5.2, we restrict our attention to families of two similar regions, since that focus simplifies the realignment algorithm (discussed below). Table 5.4 provides several quantities of interest for size-2 families in chr22. As we mentioned earlier, the vast majority (81%) of similar region families are pairs. By number of bases, 32% of the sequence found in similar regions belongs to size-2 families. As 3.3% of chr22 belongs to similar regions, 1.0% of the chromosome belongs to size-2 families. If we extrapolate these rates to the whole genome, given that the similar regions make up about 7% of the genome

Quantity	Fraction
Families that are size-2	81%
Similar regions in size-2 families	32%
Similar regions in chr22	3.3%
Size-2 families in chr22	1.0%
Size-2 families in hg19	2.2%

Table 5.4: Quantities of interest for size-2 families, in chr22 and hg19 (extrapolation).

(see Figure 4.6), and given that 32% of the sequence in similar regions is in size-2 families, we would expect that 2.2% of the genome overall lies in size-2 families.

Since our simple family creation algorithm described in Section 5.2 yields such a large majority of families as being size-2, we decided to use it. However, in the future, another avenue to explore would be to adjust the algorithm's parameters so that it yields even more size-2 families. One way to achieve this would be to require similar regions to have more than one component ID in common before drawing an edge between them. Another approach would be to use graph decomposition techniques to break up big families. Still another interesting direction would be generalizing our realignment techniques to size-N families.

Figure 5.4 illustrates our *automatic realignment* algorithm for repairing alignment errors. The input to our algorithm, shown in the top band of the figure, is a size-2 family, with the reads that have been aligned to each region via a standard alignment and classified according to whether the reads have been correctly aligned. Since we are focusing on whether we can use the similar region structure to correctly realign alignment errors, we leverage our simulated data, which gives us the correct alignment location of each read, and therefore rely on an oracle classifier to recognize errors. In Section 5.5, we discuss our plans to build a classifier to recognize alignment errors.

The second band of the figure shows the result of the classification; alignment errors in Region 1 have been marked orange, and alignment errors in Region 2 have been marked purple. Then, to realign the reads, we assume that all alignment errors can be corrected by simply aligning the erroneous read to the other region in the family. In the case of the figure, this means that we would choose to align the purple reads to Region 1 and the orange reads to Region 2. In order to align to a region, we index the region's sequence, along with a read-length flank on each end to account for reads that align at the boundaries of the region.

Table 5.5 gives the results for SNP calling accuracy, using the automatic realignment as input. Since we only perform our realignment algorithm in the size-2 similar region families, this table does not include results for the unique regions, which we provided in Table 5.2. Instead, Figure 5.5 facilitates comparison by displaying the results graphically for the unique regions and size-2 similar region families, for the case where the SVs and their flanks have

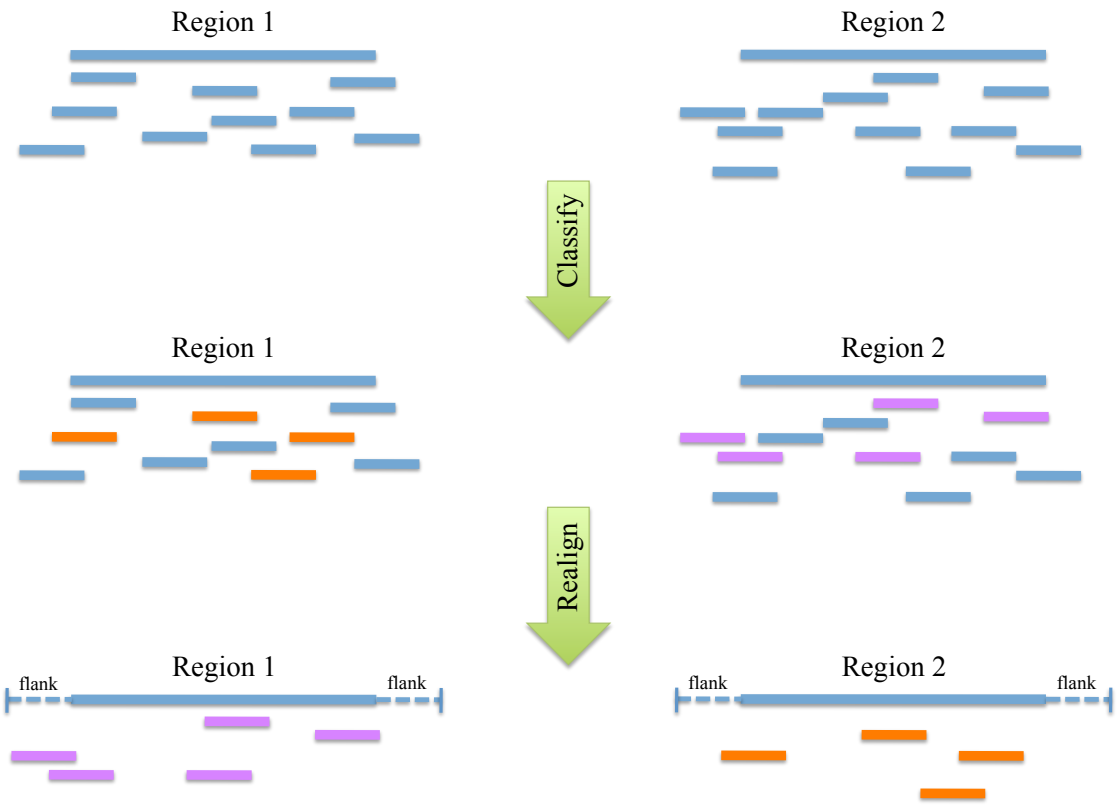


Figure 5.4: Automatic realignment. Apply a classifier to reads aligned to a size-2 family to obtain errors (shown in orange and purple). Then, align all purple reads to Region 1 and all orange reads to Region 2, where the regions are augmented with a read-length flank on each side.

been masked out. We provide the results in terms of *1−Precision* and *1−Recall* to better portray the differences between the approaches; thus, smaller is better.

Figure 5.5(a) provides the results for the baseline and the oracle alignment in the unique regions. As presented earlier, we note that the oracle alignment yields slightly better accuracy than the baseline alignment. As we indicated in Table 5.1, the alignment error rate is already quite small in the unique regions; thus, repairing the alignment errors only slightly improves the SNP calling accuracy. Figure 5.5(b) shows the accuracy for the size-2 families, with the baseline alignment, the oracle alignment, and the automatic realignment, which we just described. The realignment yields much better accuracy according to both of our metrics (and especially 1−Recall) than the baseline, but the accuracy is worse than that of the oracle aligner. Note that the realignment (the green bar) does not appear in Figure 5.5(a) since we only perform realignment in the similar regions.

In order to determine why this difference in accuracy persisted, despite our realignment technique, we manually examined some of the SNP calling errors. We call errors that our

Alignment Type	Precision	Recall
Automatic Realignment	92.3	92.0
Automatic Realignment - SVs	94.6	92.2
Automatic Realignment - (SVs + flank)	96.8	95.1

Table 5.5: SNP calling accuracy (precision and recall) in the size-2 families, using an oracle classifier with automatic realignment.

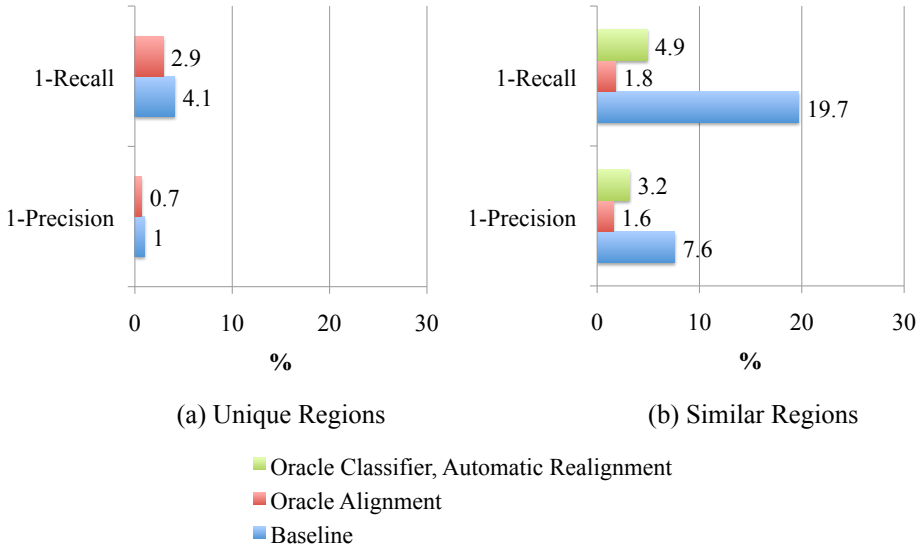
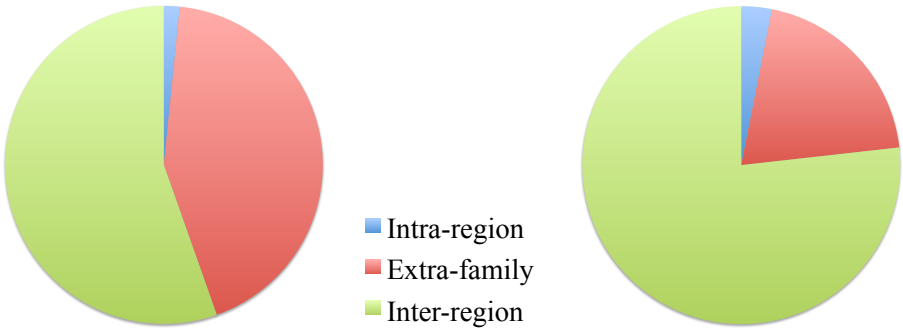


Figure 5.5: SNP calling accuracy (1 − precision, 1 − recall) in the (a) unique regions and (b) similar regions (size-2 families only), obtained by masking out the SVs and their flanks.

realignment technique targets *inter-region errors*, because they were aligned to the wrong region in the family. However, some of the errors have other causes. We call reads that are aligned to the wrong location in the correct region *intra-region errors*. Another error type, which we call *extra-family errors*, occurs when the correct location is not within the family at all. Our realignment algorithm applies to neither intra-region errors nor extra-family errors.

Figure 5.6(a) shows the prevalence of each error type for all size-2 families. Again, the inter-region errors are the error type that our realignment algorithm can repair. We see that although our algorithm targets the majority of errors, many errors are excluded. We were surprised that there were so many errors that our algorithm could not correct, given the dramatic improvement in SNP calling accuracy we presented in Figure 5.5. Figure 5.6(b) shows why: in families with at least five variants, a much greater percentage of the alignment errors are targeted by our realignment algorithm, which is likely the reason that we are able to greatly improve the VC accuracy in size-2 families despite leaving some errors uncorrected.



(a) All size-2 families

(b) Size-2 families with ≥ 5 variants

Figure 5.6: Breakdown of error types.

As we notice from Figure 5.6(a) and (b), intra-region errors contribute a small fraction of alignment errors and are therefore likely insignificant when it comes to causing SNP calling errors. However, particularly in Figure 5.6(a), extra-family errors are more prevalent and therefore worth addressing. An approach to repair these errors would involve changing how we construct the families so that more of these errors would become inter-region errors, *i.e.,* to change the families so that for each extra-family error, the read's correct location belongs to the family that contains the read's aligned location. One way we could achieve this change is to use a larger merge distance when we run the original SiRen algorithm to cluster the substrings; this would result in less-related substrings being grouped together in clusters. Another option would be to not only consider the forward strand but also the reverse strand when executing SiRen.

The tradeoff to having more inclusive families and therefore reducing the number of extra-family errors is that more of the genome would be classified as being part of similar regions. To quantify how much the similar regions would need to grow in order to eliminate extra-family errors, we computed the length of the sequence in size-2 families when they were augmented with the true locations of all extra-family errors. We found that the size-2 families are 17% larger when they include the extra-family errors' correct locations. If we extrapolate that amount of increase to the similar regions from the whole genome, the similar regions would grow from 7% of the genome (Figure 4.6) to 8.2%. As we mentioned in Section 4.5, our goal was to identify similar regions that make up less than 10% of the genome. Since the augmented similar regions could potentially still allow us to meet our goal, we believe that, as we will discuss in Chapter 6, refining our families to reduce extra-family errors is an important future direction.

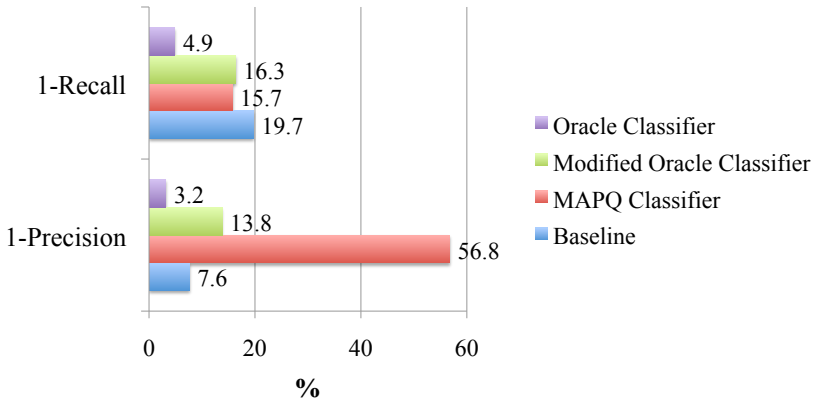


Figure 5.7: SNP calling accuracy in the size-2 families, with the SVs and their flanks masked out.

5.5 Perspectives on an Alignment Classifier

In this section, we will discuss our promising but so far unsuccessful efforts to build a classifier to recognize alignment errors. Our goal is to report on our explorations so that other researchers attempting a similar effort will avoid pursuing the approaches that we identified as ill-suited to the task and hopefully find a solution.

We decided to start with a threshold on the MAPQ score (Section 4.6), because MAPQ is how the aligner informs downstream processing stages about its confidence in where it chose to align the read. We chose a threshold of 10, which resulted in correctly classifying 67.2% of the reads. Given that the max MAPQ value is 70 and that many practitioners use 35 as a cutoff for high-quality reads, our choice of 10 is low in order to avoid counting correctly-aligned reads as incorrect.

However, the main obstacle to correct classification was that many low MAPQ reads were actually correctly aligned; even with a lower threshold, the results were very similar. To see how this would affect the VC accuracy, we used the MAPQ classifier to identify alignment errors, and then we used the realignment process we described in Section 5.4 to attempt to correct the errors. Then, we ran mpileup on the final edited alignment. The VC precision was only 43.2%, and the recall was 84.3%.

We then investigated what would happen if we realigned all the low-MAPQ errors. This approach removes the problem of changing the alignments of the many low-MAPQ reads that were correctly aligned in the baseline alignment. To do so, we modified the oracle classifier (Section 5.4) to only mark an alignment as an error if its MAPQ was below a threshold. As a result of using our modified oracle classifier, we did not realign any errors with higher MAPQ values or any low-MAPQ reads that were correctly aligned. Then, we ran mpileup on our edited alignment. The VC accuracy we obtained using a threshold of 10 for our classifier was 86.2% for the precision and 83.7% for the recall.

Figure 5.7 summarizes the SNP calling results for both the MAPQ threshold classifier and

the modified oracle classifier. We also include the baseline and the oracle classifier (Section 5.4) and again show the accuracy as 1−Precision and 1−Recall to facilitate comparison, so that smaller values are better. The MAPQ classifier achieves much worse precision and only slightly better recall than the baseline. The modified oracle classifier does much better but still gets worse precision and only slightly better recall than the baseline. Also, both MAPQ-based results are much worse than the oracle classifier, suggesting that many errors are going unrecognized.

From these results, we conclude that merely realigning low-MAPQ errors is insufficient to usefully improve VC accuracy in the size-2 families. This finding was contrary to our intuition, since the vast majority of errors have low MAPQ. However, upon further exploration of the data, we observed that the high-MAPQ errors were much more likely to overlap variants than the low-MAPQ errors. Therefore, not every error has the same importance to variant calling, and missing a small number of important errors, as we do in the case of the modified oracle classifier, has a large impact on VC accuracy.

In Section 4.6, we mentioned that though MAPQ is a valuable source of information, many practitioners do not trust it. Based on our experiences reported in this section, we agree that algorithm developers should use caution when using it as a signal about alignment results. Though MAPQ can be helpful, the fact that many low-MAPQ reads are correctly aligned and that some high-MAPQ reads are errors makes it difficult to work with MAPQ.

Another direction we explored was building a classifier using features from several different aligners. We varied the classifier type, using Spark's MLlib [86] to try an SVM, logistic regression, a decision tree, and a random forest. For features, we recorded for BWA [74], SNAP [139], and Novoalign[2] the region in the size-2 family to which the aligner assigned the read. We also recorded for each of these aligners the MAPQ value associated with its alignment. However, none of the combinations of classifier type and features that we tried made a significant change to the classification accuracy obtained with a MAPQ threshold; therefore, we obtained similarly bad VC accuracy.

So far, we have examined a single read at a time to come up with features. Going forward, we believe that it will be necessary to use features obtained by considering multiple reads at once. We believe that the ability to devise multi-read features is the main advantage of using the similar regions rather than using the original alignment. For example, Figure 3.3 showed that by looking at the reads from both regions at once, one can identify the errors in the second region, since they clearly correspond to a variant in the first region. We anticipate that multi-read features like this one will be necessary to achieve good classifier accuracy in order to fully automate this system.

5.6 Related Work

An excellent survey of how various genomic processing algorithms handle repetitive regions is available in [130]. While it is well known that the genome is comprised of a large number

[2]http://www.novocraft.com/

of repeats of many scales, variant calling algorithms such as GATK [27] do not do any special processing to address this fact.

Sniper [117] is a Bayesian method for SNP calling that accounts for genome redundancy by allowing reads to map multiple times, to different locations. Rather than forcing a read to map to a single location, Sniper preserves the aligner's uncertainty about the read's correct location. The main advantage of our work over Sniper, which would replace other variant callers, is that we have provided a foundation for building a lightweight pipeline stage that enables practitioners to continue to use their preferred variant caller. As we mentioned when we introduced variant calling in Section 2.7, practitioners are often hesitant to abandon GATK in favor of other variant callers.

Coval [62] performs alignment error correction via local realignment. This technique is related to our efforts to correct alignment errors using the similar region families, which we discuss in Section 5.4. Coval also discards some low-quality reads, where quality is determined with respect to the number of mismatches, weighted by base quality scores. We briefly discussed base quality scores in Section 2.5. The advantage of our work over Coval is that instead of performing only local error correction, our similar regions provide the foundation for correcting alignment errors in a more global manner, among the regions in a family. In addition, we do not discard any reads.

Another relevant effort is dbDNV [43], which provides an annotation of SNPs that indicates that if the SNP is called, it may be because the SNP appears in duplicated sequence, where reads from one version of the sequence align to the other. This approach is related to our finding that reads from one similar region often get mistakenly aligned to the other region in the family. The dbDNV project assumes that if SNPs are called in the duplicated sequence, they are likely to be false positives, so practitioners would want to filter them out of their call sets. A limitation of dbDNV's approach is that it only works on the exome.

Another weakness of dbDNV's approach is that it only tells practitioners which SNPs not to trust; it does not make any effort to correct any read alignments in the duplicated regions. Furthermore, reducing false positives would mostly impact the precision; however, based on our experiments in Section 4.4, as well as in Sections 5.3 and 5.4, recall is a bigger problem. An advantage of our work over dbDNV is that we lay a foundation for repairing alignment errors in size-2 families rather than requiring users to throw out SNPs called there, which could be true SNPs. However, the dbDNV SNP annotations could still be useful in combination with our work; one way we could use them is as an additional feature to recognize alignment errors, which pertains to Section 5.5.

Another related project [51] has to do with variant calling for a gene and its pseudogene, which is similar in sequence to the gene but is non-functional [137]. The goal of this project is to reduce false negative calls in the region associated with a gene, which the authors assume are being caused by reads that should align to the gene actually aligning to the pseudogene. A gene-pseudogene pair is related to our size-2 families. To avoid false negatives, this approach starts by aligning all of the reads from both sequences in the gene-pseudogene pair to the gene and then calls variants on this double-coverage alignment. If any variants are called, the authors use Sanger sequencing to verify them. The tradeoff to reducing false negatives

is that false positives are likely to result from including in their consideration of the gene many reads that should have gone to the pseudogene. Using Sanger verification can blunt the effects of false positives, though relying on Sanger may become prohibitively costly if too many false positive calls occur.

Another class of approaches that is related in its goal but not in its technique is that of using the expectation-maximization (EM) algorithm to capture the uncertainty that results when reads have multiple possible alignment locations. One example of this is a recent paper that uses an EM algorithm to allow long reads (Section 2.3) to align multiple times and then performs SNP calling on the result [53]. Like Sniper, this is a variant caller in itself, so practitioners must switch from their existing variant calling pipelines.

Another example is eXpress [108, 107], which is targeted at RNA-seq data instead of DNA and uses the EM algorithm to select the RNA-seq reads' corresponding transcript. A transcipt is an intermediate product of protein synthesis that results when DNA is transcribed to RNA and non-coding sequences are spliced out. RNA-seq data is obtained by sequencing a group of transcripts, some of which may be very similar to each other. A group of transcripts is analogous to a similar region family, and eXpress's goal of finding each read's transcript is like our goal of finding the region from which a read came. However, due to various assumptions that eXpress makes, such as preferring to assign reads to the longer transcript in a group if lengths are unequal, it is not well-suited to our problem.

The idea of a hybrid approach to genomic data processing was also discussed in [136]. The authors had begun experimenting with an easy/hard partitioning scheme, but they had not yet finalized this crucial aspect of the problem. We therefore extend their work by establishing a rigorous partitioning scheme based on the similar regions and show that our scheme is well-suited to the hybrid approach.

5.7 Conclusion

In this chapter, we went beyond our demonstration that the similar regions are associated with alignment and variant calling errors, in which we treated the similar regions like a binary indicator of whether or not a region is similar. To do so, we presented a case study to show that the structure inherent in the similar regions gives us valuable information that could be used to improve variant calling accuracy in those regions.

First, in Section 5.2, we described our algorithm for going from similar read-length substrings, which we clustered in Chapter 3, to groups of similar regions, which we call families. Also based on union-find, the algorithm is similar to SiRen, which we developed to cluster substrings. We also characterized the sizes of the families we identified.

In Section 5.3, we repudiated the possible objection that the similar regions are more difficult for variant callers solely because they have more difficult variants. While they do have more SVs than the unique regions, when we mask out the SVs, we observe that VC errors persist, due to alignment errors. Using an oracle alignment to eliminate these errors, we can approach the accuracy in the unique regions.

Then, in Section 5.4, we leverage the structure in our similar regions to reassign erroneously aligned reads to their proper alignment locations, thereby improving the VC accuracy in those regions. We showed that if we used an oracle classifier to recognize alignment errors, we can correct the majority of the errors using a straightforward realignment technique that relies on the family structure we identified. Realigning errors led to a dramatic improvement in VC accuracy compared to the baseline alignment, though not quite as good as with the oracle alignment. We discussed our thoughts about how to further improve these results by adjusting the parameters to SiRen and the tradeoffs involved in doing so.

Now that we have laid the foundation for building a complete system for read error correction, future work is necessary to realize the system. Namely, we must learn to recognize alignment errors automatically. We presented some initial work towards that end in Section 5.5. Our main finding was that MAPQ is not a good basis for building a classifier. In fact, even though we explored other per-read features, we were unable to build an accurate classifier. We therefore believe that it is necessary to use features that are developed considering multiple reads at once, thus exploiting the information we have that aligners do not, as well as leveraging the reads that aligned elsewhere in the family.

We concluded the chapter in Section 5.6 by discussing other projects that are related to the idea of what to do with alignment errors that are caused by repetition in the genome. In the next chapter, we will expand on our thoughts presented in this chapter on necessary future work to extend the platform we have built to a fully-functioning VC preprocessing step.

Chapter 6

Conclusion and Future Work

In this chapter, we present our conclusions on the work we performed (Section 6.1) and reflect on future directions for further study. In Section 6.2, we describe how, building on the foundation of the similar regions, we would build an alignment repair pipeline stage. Then, Section 6.3 outlines a number of analyses that our pipeline stage would facilitate, focusing on how we could use our system to enable advances in biological knowledge.

6.1 Summary

Variant calling accuracy is crucial for compelling applications such as personalized medicine and targeted drug development. Because of duplication in the genome, however, alignment and therefore variant calling are difficult and error-prone. In this thesis, we have developed the notion of *similar regions* to describe the genome's redundant structure. We presented the SiRen algorithm for locating similar regions, which are important due to their *identity* (*i.e.,* which regions are marked similar) and their *structure* (*i.e.,* which regions are similar to which other regions). To confirm the value of our similar regions, we performed an application-oriented validation in which we verified that the similar regions present challenges to both alignment and variant calling. We also demonstrated that the structure of the regions SiRen locates is useful as a foundation for improving variant calling accuracy therein. In what follows, we will summarize each chapter's contribution to this argument.

The context of the work in this thesis is short-read sequencing and the data processing pipeline that is used to reconstruct its reads; we described both sequencing and the pipeline in Chapter 2. Sequencing has greatly improved since the Human Genome Project produced the first draft of the complete human genome in 2003, yielding reads with a higher throughput and with dramatically lower cost. However, due to their shorter length, reconstructing the genome from reads produced by current sequencers is difficult; thus, many steps are involved in the analysis pipeline. The main steps that are relevant to this thesis are *alignment*, the problem of deciding where in the genome a read originated, and *variant calling* (VC), the problem of inferring where the sample DNA differs from the reference genome.

Chapter 3 explained in more detail that alignment and variant calling are hard because of both exact and near duplication in the genome. In particular, we described the difficulties that near duplication presents to alignment, since we were inspired to perform the work in this thesis during our previous efforts working on an alignment algorithm. We then presented the SiRen algorithm by which we locate the similar regions. SiRen is based on the union-find algorithm for detecting connected components in a graph; however, in order to produce an implementation that would scale to the entire genome, we had to modify the algorithm with several optimizations, including using an index and partitioning the similarity space. We ended the chapter with details about our distributed implementation and deployment strategies.

Since the SiRen algorithm is unsupervised, evaluating whether it captures a true pattern in the genome is complicated. Thus, Chapter 4 employs an application-oriented validation strategy for demonstrating the usefulness of SiRen's output. After delving into some high-level metrics related to SiRen's output, we evaluated our results in two phases. First, we identified reads that were hard for alignment algorithms and found that these reads were highly correlated with the similar regions. More importantly, since we ultimately care about variant calling accuracy, we showed that variant calling is more difficult in the similar regions, for both human and mouse data. We also used this evaluation to perform parameter selection for the SiRen algorithm, which is always a challenge for unsupervised algorithms.

In Chapter 5, we go beyond using the similar regions as a binary list, as previous approaches have done, and leverage the structure in the similar regions. We begin the chapter by explaining our algorithm for detecting the relationships among similar regions. We then determine that the similar regions are challenging for variant calling not only because the variants are more complex in those regions, but in large part because of alignment errors. We also show that if we isolate the effects of the structural variants and correct the alignment errors, we can approach the variant calling accuracy in the unique regions. We continue by demonstrating that the structure in the similar regions can be used to realign alignment errors, thereby improving variant calling accuracy. The structure is valuable because in the similar region families we identify, for the majority of alignment errors in those families, the correct location belongs to the family. Thus, once we know which reads are errors, we can repair many alignment errors with a simple realignment algorithm.

More work remains to build a complete system to improve variant calling. In the rest of this chapter, we outline the efforts that will allow us to complete our system. We also discuss some potential directions for exploration after the system is built.

6.2 Future Work for Pipeline

In this section, we discuss the work that remains for building the pipeline stage that repairs alignment errors in the similar regions.

Relax Assumptions

Our first line of future work would involve relaxing some of the assumptions in Chapter 5 so we can fully automate our pipeline stage. First, the realignment algorithm that we presented in Section 5.4 only applies to size-2 families. While over 80% of families are pairs, a large fraction of the sequence making up the similar regions in chr22 lies in larger families. Therefore, it may be desirable for our realignment algorithm to apply to larger families. Currently, the algorithm automatically aligns a read that has been erroneously aligned to one region in the family to the other region. We could generalize the algorithm to size-N families by aligning the read to each of the other $N - 1$ regions in the family and then selecting the region that yields the best alignment.

Currently, we can only realign reads once an oracle classifier has indicated that they are errors. Another important line of work to enable an automated system would be to build a classifier to recognize alignment errors. We outlined some initial efforts in this direction in Section 5.5. So far, the simple, intuitive features that we attempted to use yielded low classification accuracy. To build a better classifier, it will be necessary to leverage the fact that the similar regions allow us to consider all the reads from the regions in a family at once. For example, inconsistencies among reads in a single region, and relationships between reads across regions, are signals that reads have been incorrectly aligned. We demonstrated this pattern in Figure 3.3. We expect that it will be crucial to develop more features that leverage this type of cross-read information.

Refine Similar Region Families

In Section 5.4 and above, we noted that our realignment algorithm only applies to size-2 families of similiar regions. Above, we explained that one of our goals is to generalize the algorithm to size-N families. Another way to proceed would be modifying our algorithm to create similar region families, described in Section 5.2, so that we create more size-2 families. Recall that we merge two regions to create a family if they share at least one component ID. An approach that would potentially yield more small families would be to require that two regions share more component IDs; that is, we would require them to be more similar before merging them. We could also explore using graph decomposition techniques on large families to break them up into multiple smaller families.

However, achieving the above would impede one of our other important goals, which is to create families such that for most alignment errors, the read's correct location is in another region in the family. That is, we want to shift errors from being extra-family errors to being inter-region errors. If families are more restrictive, we would have more extra-family errors, which is detrimental since our realignment algorithm does not target that error type. Thus, we would expect to make a smaller improvement to variant calling accuracy.

Therefore, an important direction to explore would be to adjust our family creation algorithm to reduce the number of extra-family errors. We could increase the merge distance when we run the SiRen algorithm (Section 3.4) that clusters the read-length substrings, so

more substrings get clustered together. Intuitively, one reason for an extra-family error is that the read's true location differed from its aligned location by more than the merge distance used in running SiRen; hence, the true location would not be in a family with the aligned location. Increasing the merge distance would ameliorate this issue. We could also consider the reverse strand in addition to the forward strand when executing SiRen.

As we mentioned in Section 5.4, the best-case scenario of augmenting the size-2 families by only the extra-family errors' correct locations (with no additional sequence) results in a 17% increase to their length. The downside of larger similar regions is that our pipeline stage would have to be applied more often, leading to a greater computational expense. If the amount of increase we observed in chr22's size-2 families holds for the whole genome, however, the genome-wide similar regions would increase from 7% of the genome to 8.2%, which would still be a reasonable size. Therefore, we believe that modifying the family creation algorithm to reduce extra-family errors would be a worthwhile future direction.

From the discussion in this section, it is clear that several different tradeoffs are involved in choosing the optimal similar region family structure, where the criteria for optimization include small families (to use our existing realignment algorithm), inclusive families (to reduce extra-family errors and thereby increase variant calling accuracy), and small similar regions (to reduce latency). Thus, an important task is exploring how these criteria interact, potentially leaving it to the user to decide how to set the parameters of SiRen and the family creation algorithm according to the user's priorities. Through exploring the criteria's interactions, we could provide parameter selection guidance. In addition, since user-selected parameters would require users to run SiRen and the family creation algorithm themselves, it is crucial to choose a default setting for our tools that achieves a good balance among our criteria.

Implementation Issues

An important prerequisite to building a pipeline stage is finding the similar region families in the whole genome, rather than only one chromosome, by running the tool we described in Section 5.2. We expect this step to be straightforward since our implementation is already distributed via Spark [140]. After obtaining families for the genome, we would assess whether the extrapolation we made in Section 5.4 about size-2 families genome-wide is accurate and whether the family sizes follow a similar distribution as that of the families in chr22 (see Figure 5.3).

As we mentioned in Section 4.5, in the future we plan to build a distributed pipeline stage that can operate on similar region families in parallel. We plan to leverage ADAM [99], which facilitates the creation of scalable and efficient software to operate on read data, as a foundation for building our system. Though in this thesis we have mostly focused on VC accuracy, VC latency is also crucial, especially as genome analysis becomes more integrated into real-time decision making processes such as patient diagnosis. It is also important to reduce variant calling cost, which happens as latency decreases. Thus, an important element of building this pipeline stage would be a focus on efficiency.

We envision that this system would reside between alignment and variant calling in the pipeline diagram we presented in Figure 2.5. We would experiment to determine whether it would be best to position our new tool before or after the intermediate steps (defined in Section 2.7). We would also explore the interaction between our alignment error repair phase and the intermediate steps. It is possible that our pipeline stage would help the intermediate steps such as local realignment; if we repair alignment errors, they might not be near indels in their new locations. The intermediate steps could also help us identify alignment errors; *e.g.,* duplicate removal would correct regions of erroneously high coverage, and if coverage were used as a feature to our algorithm, adjusting coverage before creating our features would be useful.

6.3 Potential Biological Applications

In this section, we discuss other directions that are possible for us to explore once our pipeline stage is built. The focus in this section is on biologically relevant problems.

Real Data

Once we have built our pipeline stage, including an accurate classifier that recognizes alignment errors, we would apply the system to real data, rather than simulated data, as we did in Chapter 5. We would then assess whether the patterns that we saw in simulated data hold for real data; *e.g.,* using realignment, do we see the same amount of increase in VC accuracy (Section 5.4)? Recall that we can use SMaSH [125] to evaluate VC accuracy of callers like mpileup and GATK on real data as well as on simulated data, as we did in Section 4.4.

Long Reads

As we mentioned in Section 2.3, efforts are underway to introduce long reads as an alternative to second-generation sequencing. As long reads gain adoption, their use will have implications for the work presented in this thesis. First, we will need to re-run the SiRen tool to find new similar regions, using the new read length as the length of the substrings that SiRen clusters. The result may be that the fraction of the genome in similar regions decreases. Figure 3.1 shows, however, that even at a larger scale, duplication will continue to present challenges to variant calling. Thus, we expect that our work will continue to be highly relevant, even with the growing use of long reads.

Existing Databases

Another avenue would be performing further comparisons between the similar regions and some of the existing biological databases. For example, in Section 5.6, we mentioned Invitae's work on improving variant calling accuracy in gene-pseudogene pairs. We could compare

the similar regions, and especially the size-2 families, to these gene-pseudogene pairs to determine the amount of overlap. We would also explore whether Invitae's techniques could be combined with our pipeline stage to further improve VC accuracy.

In addition, in Section 4.4, we mentioned that in the NA12878 validation data from SMaSH [125], SNPs are disproportionately rare in the similar regions. We explained our hypothesis that the similar regions have few SNPs due to capture arrays being biased against these regions, potentially due to the difficulty of building capture arrays to extract their sequence. We would further analyze variant databases like dbSNP[1] to see if, like in the NA12878 SMaSH dataset, there are fewer than expected SNPs in the similar regions. We would also seek to validate our hypothesis about biases in common capture arrays.

Structural Variants

In Section 5.3, we presented our findings that structural variants are over-represented in the similar regions compared to their presence in the unique regions. Thus, even with a perfect alignment, SNP calling accuracy in the similar regions was worse than in the unique regions. A compelling future direction would involve partnering with a SV detection tool, some of which are listed in Section 2.7, in order to improve SNP calling accuracy in the SV flanks. Once we have detected SVs, we could design a specialized local realignment algorithm specifically targeted at the flanks. Of course, this approach would be limited by the fact that existing SV callers often have poor accuracy. However, any improvement to the VC accuracy in the flanks that we could make would be useful.

Benchmarking

In Chapters 4 and 5, we utilized the SMaSH benchmarking suite [125]. SMaSH is invaluable for assessing the performance of variant calling approaches on both simulated and real data. It also seeks to standardize the metrics and methods used for presenting the results of these approaches. A compelling addition to SMaSH, which is open source,[2] would be a module that specifically computes the accuracy of variant callers on the similar regions. The goal of including this module would be encouraging researchers and algorithm developers to be mindful of the similar regions, ensuring that their algorithms are accurate in those regions as well as in the unique regions.

Biological Findings

In Section 5.6, we mentioned dbDNV [43], which annotates SNPs that appear in duplicated regions, often with the goal of encouraging practitioners to throw out the SNPs if they appear in call sets. We argued that with our pipeline stage and its resulting improvement to VC

[1] `http://www.ncbi.nlm.nih.gov/SNP/`

[2] http://smash.cs.berkeley.edu/

accuracy, some of these SNPs could potentially be retained. A possible direction would be to re-assess dbDNV in light of our proposed pipeline stage, determining whether our stage improves VC accuracy to the extent that some SNPs could be removed from their list. We would then explore whether the fact that we could now confidently call SNPs that were previously thrown out enables new analyses that are currently hobbled by having to discard data. Assuming that some new SNPs could be trusted due to our pipeline stage, it would be interesting to re-call large datasets from projects like the TCGA [22, 20, 19] and the 1000 Genomes Project [21, 1] to see what new insights could be gained.

6.4 Concluding Remarks

This chapter presents our conclusions regarding the work performed for this thesis as well as future directions. In particular, we outline the remaining steps to build an alignment repair system, based on the foundation that the similar regions provide. We also discuss potential avenues that our system enables for further study in the biological realm. We believe that a system built upon the platform we have established will enable several interesting new advances in genomics.

Bibliography

[1] G. R. Abecasis et al. "An integrated map of genetic variation from 1,092 human genomes". In: *Nature* 491 (2012), pp. 56–65.

[2] Pankaj Agarwal and David J. States. "The Repeat Pattern Toolkit (RPT): Analyzing the Structure and Evolution of the C. elegans Genome". In: *Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology (ISMB-94).* AAAI Press, 1994, pp. 1–9.

[3] Can Alkan, Bradley P. Coe, and Evan E. Eichler. "Genome structural variation discovery and genotyping". In: *Nature Reviews Genetics* (2011).

[4] Can Alkan et al. "Personalized copy number and segmental duplication maps using next-generation sequencing". In: *Nature Genetics* 41.10 (2009), pp. 1061–1067.

[5] Stephen F. Altschul et al. "Basic local alignment search tool". In: *Journal of Molecular Biology* 215.3 (1990), pp. 403–410.

[6] Yann Astier, Orit Braha, and Hagan Bayley. "Toward Single Molecule DNA Sequencing: Direct Identification of Ribonucleoside and Deoxyribonucleoside 5'-Monophosphates by Using an Engineered Protein Nanopore Equipped with a Molecular Adapter". In: *Journal of the American Chemical Society* 128.5 (2006), pp. 1705–1710.

[7] Zhirong Bao and Sean R. Eddy. "Automated De Novo Identification of Repeat Sequence Families in Sequenced Genomes". In: *Genome Research* 12 (2002), pp. 1269–1276.

[8] David R. Bentley et al. "Accurate whole human genome sequencing using reversible terminator chemistry". In: *Nature* 456 (2008), pp. 53–59.

[9] Lenny Bernstein. "What is President Obama's 'precision medicine' plan, and how might it help you?" In: *The Washington Post* (Jan. 21, 2015).

[10] Adam Bloniarz et al. "Changepoint Analysis for Efficient Variant Calling". In: *Research in Computational Molecular Biology*. Ed. by Roded Sharan. Vol. 8394. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 20–34.

[11] Ivana Bozic et al. "Accumulation of driver and passenger mutations during tumor progression". In: *PNAS* 107.43 (2010).

[12] Ma'ayan Bresler et al. "Telescoper: de novo assembly of highly repetitive regions". In: *Bioinformatics* 28 ECCB 2012 (2012), pp. i311–i317.

[13] Nicolaas V. de Bruijn. "A Combinatorial Problem". In: *Koninklijke Nederlandse Akademie v. Wetenschappen* 49 (1946), pp. 758–764.

[14] M. Burrows and D. J. Wheeler. *A Block-sorting Lossless Data Compression Algorithm*. Tech. rep. 124. Palo Alto, CA: Digital Equipment Corporation, 1994.

[15] Jonathan Butler et al. "ALLPATHS: De novo assembly of whole-genome shotgun microreads". In: *Genome Research* 18 (2008), pp. 810–820.

[16] Davide Campagna et al. "PASS: a program to align short sequences". In: *Bioinformatics* 25.7 (2009), pp. 967–968.

[17] Ken Chen et al. "BreakDancer: an algorithm for high-resolution mapping of genomic structural variation". In: *Nature Methods* 6.9 (2009), pp. 677–681.

[18] Yangho Chen, Tade Souaiaia, and Ting Chen. "PerM: efficient mapping of short sequencing reads with periodic full sensitive spaced seeds". In: *Bioinformatics* 25.19 (2009), pp. 2514–2521.

[19] Lynda Chin, Jannik N. Andersen, and P. Andrew Futreal. "Cancer genomics: from discovery science to personalized medicine". In: *Nature Medicine* 17 (2011), pp. 297–303.

[20] Lynda Chin et al. "Making sense of cancer genomic data". In: *Genes and Development* 25 (2011), pp. 534–555.

[21] The 1000 Genomes Project Consortium. "A map of human genome variation from population-scale sequencing". In: *Nature* 467 (2010).

[22] The International Cancer Genome Consortium. "International network of cancer genome projects". In: *Nature* 464 (2010), pp. 993–998.

[23] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd ed. MIT Press, 2009.

[24] A. J. Cox. "ELAND: Efficient local alignment of nucleotide data". Unpublished. 2007.

[25] Kristal Curtis et al. *SiRen: Leveraging Similar Regions for Efficient & Accurate Variant Calling*. Tech. rep. UCB/EECS-2015-159. University of California, Berkeley, 2015.

[26] Matei David et al. "SHRiMP2: Sensitive yet Practical Short Read Mapping". In: *Bioinformatics* 27.7 (2011), pp. 1011–1012.

[27] Mark A. DePristo et al. "A framework for variation discovery and genotyping using next-generation DNA sequencing data". In: *Nature Genetics* 43.5 (2011).

[28] Thomas Derrien et al. "Fast Computation and Applications of Genome Mappability". In: *PLOS ONE* 7.1 (2012).

[29] Hugh L. Eaves and Yuan Gao. "MOM: maximum oligonucleotide mapping". In: *Bioinformatics* 25.7 (2009), pp. 969–970.

[30] Robert C. Edgar. "Search and clustering orders of magnitude faster than BLAST". In: *Bioinformatics* 26.19 (2010), pp. 2460–2461.

[31] P. Ferragina and G. Manzini. "Opportunistic data structures with applications". In: *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*. IEEE, 2000, pp. 390–398.

[32] Erik Garrison and Gabor Marth. *Haplotype-based variant detection from short-read sequencing*. `http://arxiv.org/abs/1207.3907`. 2012.

[33] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. "Similarity Search in High Dimensions via Hashing". In: *Proceedings of the 25th VLDB*. 1999, pp. 519–529.

[34] Sante Gnerre et al. "High-quality draft assemblies of mammalian genomes from massively parallel sequence data". In: *PNAS* 108.4 (2011), pp. 1513–1518.

[35] Larry Gonick and Mark Wheelis. *The Cartoon Guide to Genetics*. Updated. 10 East 53rd Street, New York, NY 10022: Harper Collins Publishers, 1991.

[36] Joseph E. Gonzalez et al. "GraphX: Graph Processing in a Distributed Dataflow Framework". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014, pp. 599–613.

[37] The SAM/BAM Format Specification Working Group. *Sequence Alignment/Map Format Specification*. `http://samtools.github.io/hts-specs/SAMv1.pdf`. 2015.

[38] James Gurtowski, Michael C. Schatz, and Ben Langmead. "Genotyping in the Cloud with Crossbow". In: *Current Protocols in Bioinformatics* 39.15.3 (2012), pp. 15.3.1–15.3.15.

[39] Faraz Hach et al. "mrsFAST: a cache-oblivious algorithm for short-read mapping". In: *Nature Methods* 7.8 (2010), pp. 576–577.

[40] Douglas Hanahan and Robert A. Weinberg. "Hallmarks of Cancer: The Next Generation". In: *Cell* 144.5 (2011), pp. 646–674.

[41] Benjamin Hindman et al. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center". In: *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*. 2011, pp. 295–308.

[42] N. Hjort et al., eds. *Bayesian Nonparametrics: Principles and Practice*. Cambridge University Press, 2010.

[43] Meng-Ru Ho et al. "dbDNV: a resource of duplicated gene nucleotide variants in human genome". In: *Nucleic Acids Research* 39 (2011), pp. D920–D925.

[44] Manuel Holtgrewe. *Mason - a read simulator for second generation sequencing data*. Tech. rep. TR-B-10-06. Institut für Mathematik und Informatik, Freie Universität Berlin, 2010.

[45] Nils Homer, Barry Merriman, and Stanley F. Nelson. "BFAST: An Alignment Tool for Large Scale Genome Resequencing". In: *PLOS ONE* 4.11 (2009).

[46] Fereydoun Hormozdiari et al. "Combinatorial algorithms for structural variation detection in high-throughput sequenced genomes". In: *Genome Research* 19 (2009), pp. 1270–1278.

[47] Fereydoun Hormozdiari et al. "Next-generation VariationHunter: combinatorial algorithms for transposon insertion discovery". In: *Bioinformatics* 26 (2010), pp. i350–i357.

[48] Illumina Inc. *Sequencing by Synthesis (SBS) Technology*. `http://www.illumina.com/technology/next-generation-sequencing/sequencing-technology.html`. 2015.

[49] Illumina Inc. *Technology Spotlight: Illumina Sequencing*. `http://www.illumina.com/documents/products/techspotlights/techspotlight_sequencing.pdf`.

[50] National Human Genome Research Institute. *All About The Human Genome Project (HGP)*. `http://www.genome.gov/10001772`. 2014.

[51] Invitae. *Sequencing exons 12-15 of PMS2 using next-generation sequencing (NGS)*. `http://marketing.invitae.com/acton/attachment/7098/f-0139/1/-/-/-/-/WP103-1_PMS2\%20Sequencing\%20NGS\%20Validation\%20Summary.pdf`. 2015.

[52] Zamin Iqbal et al. "De novo assembly and genotyping of variants using colored de Bruijn graphs". In: *Nature Genetics* 44 (2012), pp. 226–232.

[53] Miten Jain et al. "Improved data analysis for the MinION nanopore sequencer". In: *Nature Methods* 12.4 (2015), pp. 351–356.

[54] Hui Jiang and Wing Hung Wong. "SeqMap: mapping massive amount of oligonucleotides to the genome". In: *Bioinformatics* 24.20 (2008), pp. 2395–2396.

[55] J. Jurka et al. "Repbase Update, a database of eukaryotic repetitive elements". In: *Cytogenic and Genome Research* (2005).

[56] Wei-Chun Kao, Kristian Stevens, and Yun S. Song. "BayesCall: A model-based base-calling algorithm for high-throughput short-read sequencing". In: *Genome Research* (2009).

[57] W. James Kent. "BLAT – The BLAST-Like Alignment Tool". In: *Genome Research* 12.4 (2002), pp. 656–664.

[58] You Jung Kim et al. "ProbeMatch: rapid alignment of oligonucleotides to genome allowing both gaps and mismatches". In: *Bioinformatics* 25.11 (2009), pp. 1424–1425.

[59] Martin Kircher, Udo Stenzel, and Janet Kelso. "Improved base calling for the Illumina Genome Analyzer using machine learning strategies". In: *Genome Biology* 10.8 (2009).

[60] Donald E. Knuth. *The Art of Computer Programming*. 2nd ed. Vol. 3: Sorting and Searching. Addison-Wesley Professional, 1998.

[61] Gina Kolata. "In Treatment for Leukemia, Glimpses of the Future". In: *The New York Times* (July 7, 2012).

[62] Shunichi Kosugi et al. "Coval: Improving Alignment Quality and Variant Calling Accuracy for Next-Generation Sequencing Data". In: *PLOS ONE* 8.10 (2013).

[63] Stefan Kurtz and Chris Schleiermacher. "REPuter: fast computation of maximal repeats in complete genomes". In: *Bioinformatics* 15.5 (1999), pp. 426–427.

[64] Laura Kurtzman. *California Launches Initiative to Advance Precision Medicine*. `http://www.ucsf.edu/news/2015/04/125111/california-launches-initiative-advance-precision-medicine`. 2015.

[65] Eric S. Lander et al. "Initial sequencing and analysis of the human genome". In: *Nature* 409 (2001), pp. 860–921.

[66] Ben Langmead and Steven L. Salzberg. "Fast gapped-read alignment with Bowtie 2". In: *Nature Methods* 9.4 (2012), pp. 357–359.

[67] Ben Langmead et al. "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome". In: *Genome Biology* 10.3 (2009), R25.1–R25.10.

[68] Christian Ledergerber and Christophe Dessimoz. "Base-calling for next-generation sequencing platforms". In: *Briefings in Bioinformatics* 12.5 (2011), pp. 489–497.

[69] Wan-Ping Lee et al. "MOSAIK: A Hash-Based Algorithm for Accurate Next-Generation Sequencing Short-Read Mapping". In: *PLOS ONE* 9.3 (2014).

[70] S. Levy et al. "The Diploid Genome Sequence of an Individual Human". In: *PLoS Biology* 5.10 (2007).

[71] Heng Li. "A statistical framework for SNP calling, mutation discovery, association mapping and population genetical parameter estimation from sequencing data". In: *Bioinformatics* 27.21 (2011), pp. 2987–2993.

[72] Heng Li. *Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM*. `http://arxiv.org/abs/1303.3997`. 2013.

[73] Heng Li. *FermiKit: assembly-based variant calling for Illumina resequencing data*. `http://arxiv.org/abs/1504.06574`. 2015.

[74] Heng Li and Richard Durbin. "Fast and accurate short read alignment with Burrows-Wheeler transform". In: *Bioinformatics* 25.14 (2009), pp. 1754–1760.

[75] Heng Li, Jue Ruan, and Richard Durbin. "Mapping short DNA sequencing reads and calling variants using mapping quality scores". In: *Genome Research* 18 (2008), pp. 1851–1858.

[76] Heng Li et al. "The Sequence Alignment/Map format and SAMtools". In: *Bioinformatics* 25.16 (2009), pp. 2078–2079.

[77] Ruiqiang Li et al. "De novo assembly of human genomes with massively parallel short read sequencing". In: *Genome Research* 20 (2010), pp. 265–272.

[78] Ruiqiang Li et al. "SNP detection for massively parallel whole-genome resequencing". In: *Genome Research* 19 (2009), pp. 1124–1132.

[79] Ruiqiang Li et al. "SOAP: short oligonucleotide alignment program". In: *Bioinformatics* 24.5 (2008), pp. 713–714.

[80] Ruiqiang Li et al. "SOAP2: an improved ultrafast tool for short read alignment". In: *Bioinformatics* 25.15 (2009), pp. 1966–1967.

[81] Shengting Li et al. "SOAPindel: Efficient identification of indels from short paired reads". In: *Genome Research* 23 (2012), pp. 195–200.

[82] Wentian Li and Jan Freudenberg. "Characterizing regions in the human genome unmappable by next-generation-sequencing at the read length of 1000 bases". In: *Computational Biology and Chemistry* 53.A (2014), pp. 108–117.

[83] Hao Lin et al. "ZOOM! Zillions of oligos mapped". In: *Bioinformatics* 24.21 (2008), pp. 2431–2437.

[84] Gerton Lunter and Martin Goodson. "Stampy: A statistical algorithm for sensitive and fast mapping of Illumina sequence reads". In: *Genome Research* 21 (2011), pp. 936–939.

[85] Ruibang Luo et al. "SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler". In: *GigaScience* 1.18 (2012).

[86] *Machine Learning Library (MLlib) Guide*. `https://spark.apache.org/docs/latest/mllib-guide.html`. 2015.

[87] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[88] Santiago Marco-Sola et al. "The GEM mapper: fast, accurate, and versatile alignment by filtration". In: *Nature Methods* 9 (2012), pp. 1185–1188.

[89] Matt Massie et al. *ADAM: Genomics Formats and Processing Patterns for Cloud Scale Computing*. Tech. rep. UCB/EECS-2013-207. EECS Department, University of California, Berkeley, 2013.

[90] Tim Massingham. *simNGS and simLibrary software for simulating next-gen sequencing data*. 2012. URL: `http://www.ebi.ac.uk/goldman-srv/simNGS`.

[91] Tim Massingham and Nick Goldman. "All Your Base: a fast and accurate probabilistic approach to base calling". In: *Genome Biology* (2012).

[92] Frazer Meacham et al. "Identification and correction of systematic error in high-throughput sequence data". In: *BMC Bioinformatics* (2011).

[93] Siddhartha Mukherjee. *The Emperor of All Maladies*. Scribner, 2011.

[94] Niranjan Nagarajan and Mihai Pop. "Sequence assembly demystified". In: *Nature Reviews Genetics* 14 (2013), pp. 157–167.

[95] Oxford Nanopore. *Nanopore DNA sequencing*. `https://nanoporetech.com/science-technology/movies\#movie-24-nanopore-dna-sequencing`.

[96] Giuseppe Narzisi et al. "Accurate de novo and transmitted indel detection in exome-capture data using microassembly". In: *Nature Methods* 11 (2014), pp. 1033–1036.

[97] Sarah B. Ng et al. "Targeted Capture and Massively Parallel Sequencing of Twelve Human Exomes". In: *Nature* 461.7261 (2009), pp. 272–276.

[98] Rasmus Nielsen et al. "Genotype and SNP calling from next-generation sequencing data". In: *Nature Reviews Genetics* 12 (2011).

[99] Frank Austin Nothaft et al. "Rethinking Data-Intensive Science Using Scalable Analytics Systems". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 2015.

[100] *NovoAlign User Guide*. `http://www.novocraft.com/documentation/novoalign-2/novoalign-user-guide/`.

[101] David Patterson. "Computer Scientists May Have What It Takes to Help Cure Cancer". In: *The New York Times* (Dec. 5, 2011).

[102] Matthew Pendleton et al. "Assembly and diploid architecture of an individual human genome via single-molecule technologies". In: *Nature Methods* (2015).

[103] Alkes L. Price, Neil C. Jones, and Pavel A. Pevzner. "De novo identification of repeat families in large genomes". In: *Bioinformatics* 21 (2005), pp. i351–i358.

[104] Come Raczy et al. "Isaac: ultra-fast whole-genome secondary analysis on Illumina sequencing platforms". In: *Bioinformatics* 29.16 (2013), pp. 2041–2043.

[105] Andy Rimmer et al. "Integrating mapping-, assembly-, and haplotype-based approaches for calling variants in clinical sequencing applications". In: *Nature Genetics* 46 (2014), pp. 912–918.

[106] John L. Rinn et al. "Functional Demarcation of Active and Silent Chromatin Domains in Human *HOX* Loci by Noncoding RNAs". In: *Cell* 129.7 (2007), pp. 1311–1323.

[107] Adam Roberts, Harvey Feng, and Lior Pachter. "Fragment assignment in the cloud with eXpress-D". In: *BMC Bioinformatics* 14.358 (2013), pp. 1–9.

[108] Adam Roberts and Lior Pachter. "Streaming fragment assignment for real-time analysis of sequencing experiments". In: *Nature Methods* 10.1 (2013), pp. 71–73.

[109] James T. Robinson et al. "Integrative genomics viewer". In: *Nature Biotechnology* 29.1 (2011), pp. 24–26.

[110] Stephen M. Rumble et al. "SHRiMP: Accurate Mapping of Short Color-space Reads". In: *PLOS Computational Biology* 5.5 (2009).

[111] F. Sanger, S. Nicklen, and A. R. Coulson. "DNA sequencing with chain-terminating inhibitors". In: *Proceedings of the National Academy of Sciences of the USA (PNAS)* 74.12 (1977), pp. 5463–5467.

[112] Eric E. Schadt, Steve Turner, and Andrew Kasarskis. "A window into third-generation sequencing". In: *Human Molecular Genetics* 19.2 (2010), R227–R240.

[113] Michael C. Schatz. "CloudBurst: highly sensitive read mapping with MapReduce". In: *Bioinformatics* 25.11 (2009), pp. 1363–1369.

[114] S. Schuster. "Next-generation sequencing transforms today's biology". In: *Nature Methods* 5 (2008), pp. 16–18.

[115] Jay Shendure and Ji Hanlee. "Next-generation DNA sequencing". In: *Nature Biotechnology* 26 (2008), pp. 1135–1145.

[116] Jianbo Shi and Jitendra Malik. "Normalized Cuts and Image Segmentation". In: *IEEE Transactions on PAMI* 22.8 (2000).

[117] Daniel F. Simola and Junhyong Kim. "Sniper: improved SNP discovery by multiply mapping deep sequenced reads". In: *Genome Biology* 12 (2011).

[118] Jared T. Simpson and Richard Durbin. "Efficient de novo assembly of large genomes using compressed data structures". In: *Genome Research* 22 (2012), pp. 549–556.

[119] Jared T. Simpson et al. "ABySS: A parallel assembler for short read sequence data". In: *Genome Research* 19 (2009), pp. 1117–1123.

[120] Andrew D. Smith, Zhenyu Xuan, and Michael Q. Zhang. "Using quality scores and longer reads improves accuracy of Solexa read mapping". In: *BMC Bioinformatics* 9.128 (2008).

[121] T. F. Smith and M. S. Waterman. "Identification of common molecular subsequences". In: *Journal of Molecular Biology* 147.1 (1981), pp. 195–197.

[122] *Spark Programming Guide*. 1.2.0. `http://spark.apache.org/docs/1.2.0/programming-guide.html`.

[123] Karyn Meltz Steinberg et al. "Single haplotype assembly of the human genome from a hydatidiform mole". In: *Genome Research* 24 (2014), pp. 2066–2076.

[124] Ameet Talwalkar et al. *SMaSH: A Benchmarking Toolkit for Human Genome Variant Calling*. `http://arxiv.org/abs/1310.8420`. 2013.

[125] Ameet Talwalkar et al. "SMaSH: a benchmarking toolkit for human genome variant calling". In: *Bioinformatics* 30.19 (2014), pp. 2787–2795.

[126] Maja Tarailo-Graovac and Nansheng Chen. "Using RepeatMasker to identify repetitive elements in genomic sequences". In: *Current Protocols in Bioinformatics* 25 (2009), pp. 4.10.1–4.10.14.

[127] *The Variant Call Format (VCF) Version 4.2 Specification*. `https://samtools.github.io/hts-specs/VCFv4.2.pdf`. 2015.

[128] John F. Thompson and Patrice M. Milos. "The properties and applications of single-molecule DNA sequencing". In: *Genome Biology* 12.2 (2011).

[129] Helga Thorvaldsdóttir, James T. Robinson, and Jill P. Mesirov. "Integrative Genomics Viewer (IGV): high-performance genomics data visualization and exploration". In: *Briefings in Bioinformatics* 14.2 (2013), pp. 178–192.

[130] Todd J. Treangen and Steven L. Salzberg. "Repetitive DNA and next-generation sequencing: computational challenges and solutions". In: *Nature* 13 (2012).

[131] *U.S. health secretary lauds promise of 'precision medicine'*. `http://universityofcalifornia.edu/news/us-health-secretary-lauds-promise-\%E2\%80\%98precision-medicine\%E2\%80\%99`. 2015.

[132] Natalia Volfovsky, Brian J. Haas, and Steven L. Salzberg. "A clustering method for repeat analysis in DNA sequences". In: *Genome Biology* 2.8 (2001).

[133] Neil I. Weisenfeld et al. "Comprehensive variation discovery in single human genomes". In: *Nature Genetics* 46 (2014), pp. 1350–1355.

[134] Kris Wetterstrand. *DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP)*. `http://www.genome.gov/sequencingcosts/`. 2015.

[135] Michael R. Wilson et al. "Actionable Diagnosis of Neuroleptospirosis by Next-Generation Sequencing". In: *The New England Journal of Medicine* 370 (2014), pp. 2408–2417.

[136] Richard Xia et al. "Distributed Pipeline for Genomic Variant Calling". In: *NIPS Workshop on BIG Learning*. 2012.

[137] Gerstein Lab at Yale. *Background Information on Pseudogenes*. `http://pseudogene.org/background.php`.

[138] Kai Ye et al. "Pindel: a pattern growth approach to detect break points of large deletions and medium sized insertions from paired-end short reads". In: *Bioinformatics* 25 (2009), pp. 2865–2871.

[139] Matei Zaharia et al. *Faster and More Accurate Sequence Alignment with SNAP*. `http://arxiv.org/abs/1111.5572`. 2011.

[140] Matei Zaharia et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *NSDI 2012*. 2012.

[141] Daniel R. Zerbino. "Using the Velvet de novo Assembler for Short-Read Sequencing Technologies". In: *Current Protocols in Bioinformatics* 31 (2010), pp. 11.5.1–11.5.12.

[142] Daniel R. Zerbino and Ewan Birney. "Velvet: Algorithms for de novo short read assembly using de Bruijn graphs". In: *Genome Research* 18 (2008), pp. 821–829.

[143] Zefeng Zhang, Hao Lin, and Bin Ma. "ZOOM Lite: next-generation sequencing data mapping and visualization software". In: *Nucleic Acids Research* 38 (2010), W743–W748.

[144] Carl Zimmer. "In a First, Test of DNA Finds Root of Illness". In: *The New York Times* (June 4, 2014).