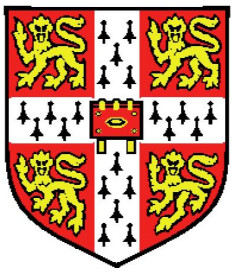
Efficient sequence assembly and variant calling using compressed data structures



Jared Thomas Simpson

Queens' College

Wellcome Trust Sanger Institute

University of Cambridge

This dissertation is submitted for the degree of

Doctor of Philosophy

September 2012

I took a lengthy path to reach this point and my family supported me the entire way. I dedicate this work to them - thank you Mom, Dad, Calley and Kim.

Declaration

This dissertation describes work carried out from May 2009 to July 2012 under the supervision of Dr Richard Durbin at the Wellcome Trust Sanger Institute, while a member of Queens' College, Cambridge. This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text. The content in Chapter 2 was published in Simpson and Durbin [2010]. The content of Chapter 3 was published in Simpson and Durbin [2012].

This thesis does not exceed the length limit of 60,000 words as specified by the Biology Degree Committee.

Jared Thomas Simpson
August 29, 2012

Acknowledgements

My decision to undertake a Ph.D. was influenced by many of my friends and colleagues. I owe thanks to my close friends Gwynn Elfring and Scott Drader, who have been a constant source of support and advice. My friend Christian Steidl introduced me to Computational Biology. This introduction led me to work on the sequence assembly problem, which directly led to this work.

My Ph.D. benefitted from many people at the Sanger Institute, most importantly my supervisor Richard Durbin. Without Richard's guidance and insight this work would be greatly diminished. I am very appreciative of the opportunity to work with Richard and proud of the work we accomplished over the last four years. I had many great discussions within Richard's group in journal clubs, group meetings, lunches and over coffee. These discussions introduced me to many areas of genetics and provided a great environment for developing as a scientist. A strength of the Ph.D. program at Sanger is the opportunity to build collaborations across many different research areas. Within Richard's group I enjoyed collaborating with Leopold Parts, Aylwyn Scally and Kees Albers. I also appreciate the time I spent working in David Adams' lab and the Cancer Genome Project under Peter Campbell. I owe thanks to the Wellcome Trust for financially supporting my Ph.D. and Annabel Smith and Christina Hedberg-Delouka for making sure the graduate program at Sanger runs smoothly.

Summary

De novo genome assembly is one of the most computationally demanding problems in genomics. In this thesis, I describe a collection of novel algorithms for performing *de novo* assembly using compressed data structures. First, I describe an algorithm to directly construct the assembly string graph - a model of overlap-based sequence assembly - using the compressed FM-index data structure. Previous algorithms for constructing the string graph required the intermediate step of building a full overlap graph, then removing transitive edges from the graph. My novel FM-index based algorithm does not require this time-consuming intermediate step. This algorithm allows fast and memory efficient overlap-based assembly. In Chapter 3, I extend my FM-index algorithms to build a space-efficient assembler for real sequencing data by designing error correction, read merging and scaffolding algorithms. Using these efficient algorithms I am able to reduce the memory requirement for assembling a human genome to 54GB.

In Chapter 4, I address the problem of detecting DNA sequence differences between two related genomes - the *variant calling* problem. Traditional approaches to variant calling align short sequence reads to a reference genome. While this approach is effective for simple differences, like isolated SNPs, it is more difficult to find complex changes like the insertion or deletion of sequence. My approach is based on analyzing the structure of an assembly graph built from the sequence data from multiple individuals. In Chapter 5, I apply this approach to real sequencing problems, including finding *de novo* mutations in the child of two parents, somatically acquired mutations in cancer and polymorphic variants present in a large human population.

Contents

List of Figures

Chapter 1

Introduction

A hallmark of modern science is the collection of large volumes of data by measuring physical processes. Experiments which collect terabytes of data are common in particle physics and astronomy. In biology, we can now sequence the DNA of previously uncharacterized organisms, entire human populations or thousands of human cancers. These applications need extremely computationally efficient algorithms to process the data. In this work, I will present efficient algorithms for processing raw DNA sequence data. The algorithms I will develop are based on the idea of querying a *compressed data structure*. These data structures become more efficient in memory use as the redundancy in the data increases, while retaining the ability to perform efficient queries. Throughout this text I will develop algorithms for building and querying these structures in the context of DNA sequence data. Using the algorithms I develop, I will address two major problems in sequence analysis. The first problem is the efficient reconstruction of a genome - the full complement of DNA within a cell - using only the output from a DNA sequencing instrument. Given the scale of the problem, often requiring hundreds of gigabytes of data, computation time and memory efficiency is a primary concern. The second problem that I will address is the detection of differences in the sequence of DNA between related genomes. The approach I will develop does not require the alignment of raw sequence reads to a reference genome - it works directly from the sequence reads themselves.

In the remainder of this chapter, I will introduce DNA sequencing and the key problem of reconstructing a genome from a set of sequence reads. I will

also discuss previous approaches to the reconstruction problem and give a brief overview of compressed data structures. At the end of this chapter, I provide an overview of the remainder of this text.

1.1 DNA Sequencing

The field of molecular biology developed rapidly in the second half of the 20th century. Watson and Crick's discovery of the double-helix structure of DNA told us how genetic information is copied within a cell and transmitted from generation to generation. The determination of the genetic code - how the sequence of nucleotides in genes encode the information necessary for constructing proteins - led to the formation of the central dogma of molecular biology, which describes how DNA encodes RNA which encodes proteins. It was thus readily apparent that the sequence of nucleotides making up an individual's genome underlies the protein complement of the cell and hence much phenotypic variation that we see between individuals. The development of methods for determining the sequence of nucleotides in DNA molecules would have great importance to the study of human health and the evolution of life. Using techniques developed by Sanger and colleagues to determine the nucleotide sequence of short RNA fragments [Sanger et al., 1965], Walter Fiers and colleagues sequenced the first gene, the coat protein of bacteriophage MS2 [Jou et al., 1972]. Later in 1976 the complete sequence of bacteriophage MS2 would be determined [Fiers et al., 1976]. Maxam and Gilbert developed a method for directly determining the sequence of nucleotides in DNA based on breaking radioactively-labelled DNA at specific positions, then size-sorting the DNA fragments using gel electrophoresis [Maxam and Gilbert, 1977]. Also in 1977 Sanger, Nicklen and Coulson developed a method that would dominate DNA sequencing for almost three decades. This method, called chain-termination sequencing and widely known as Sanger sequencing, is based on introducing specially modified nucleotides which do not allow extension of a DNA chain. When these terminating nucleotides are incorporated into a growing DNA chain, they stop the reaction from proceeding any further. The result is a mixture of partial copies of the original template DNA. This mixture can be sorted by fragment length using gel electrophoresis, and the sequence of

the DNA template can be read from the gel by the pattern of bands indicating which modified base (A, C, G or T) stopped the chain at a given position [Sanger et al., 1977]. Sanger and Gilbert's contributions to DNA sequencing would earn them the 1980 Nobel Prize in Chemistry (shared with Paul Berg).

Over the next three decades the efficiency and throughput of chain-termination sequencing was greatly improved, particularly as a result of automation of the sequencing process. Gel electrophoresis and radioactively labelled nucleotides were replaced by capillary tubes and fluorescent nucleotides, allowing the automated imaging and analysis of the sequencing reaction using a computer [Smith et al., 1986]. Multiple sequencing reactions were run in parallel. The read length - the number of contiguous bases sequenced in a single reaction - improved to 500-1000 bases. These improvements to the throughput of DNA sequencing led to the start of the genomics era, where entire genomes could be sequenced. The first complete genome sequenced of a free-living organism was the 1.8 megabase genome of *Haemophilus influenzae* published in 1995 [Fleischmann et al., 1995]. As the cost of sequencing continued to fall, larger genomes were sequenced like that of *Escherichia coli* [Blattner et al., 1997], *Saccharomyces cerevisiae* [Goffeau et al., 1996], *Caenorhabditis elegans* [C. elegans Sequencing Consortium, 1998] and the model plant *Arabidopsis thaliana* [Arabidopsis Genome Initiative, 2000].

An international consortium to sequence the human genome was formed in the early 1990s. A competing privately funded project was later started by the Celera corporation. These competing projects progressed in contrasting styles. As the human genome was already known to be highly repetitive [Schmid and Deininger, 1975], the publicly funded Human Genome Project (HGP) took a conservative approach to sequencing. They developed libraries of Bacterial Artificial Chromosomes (BACs), 150 kilobase fragments of human DNA copied in a bacterial cell. Restriction digestions of the BACs were created and ordered into a 'map' based on the overlapping patterns of restriction fragments formed by gel electrophoresis. Using the map, individual BACs were selected for direct sequencing to tile across each chromosome. The BAC map gave a scaffold on which to place the individually sequenced clones. The privately funded project opted for a more aggressive, faster approach. This method, termed *whole genome shotgun sequencing*, did not construct a map but rather sampled random sequence

reads from the entire genome. These raw shotgun reads would be augmented by 'mate-pair' reads where both ends of a long DNA fragment would be sequenced, with unknown sequence in between. This strategy relied on the development of sophisticated computational algorithms to determine the order of the sequence reads and assemble them into the genome. In 2001, the two projects published their drafts of the human genome [International Human Genome Sequencing Consortium, 2001; Venter et al., 2001]. A vigorous debate on the effectiveness and independence of Celera's approach to sequencing the human genome ensued in the literature for a number of years [Adams et al., 2003; Green, 2002; Myers et al., 2002; Waterston et al., 2002, 2003]. Subsequent to the sequencing of the human genome, the genome of a laboratory strain of mouse was sequenced using a combination of BAC-based and whole genome shotgun data [Mouse Genome Sequencing Consortium, 2002].

More recently many genomes have been sequenced including the Chimpanzee [Chimpanzee Sequencing and Analysis Consortium, 2005], the Giant Panda [Li et al., 2010a], the Gorilla [Scally et al., 2012] and the Bonobo [Prufer et al., 2012]. The default is now whole genome shotgun assembly, which can provide the complete genome sequence for small prokaryotic genomes but for larger and more complex genomes assemblies are typically incomplete.

1.1.1 High Throughput Sequencing

While the efficiency of Sanger sequencing was improved by orders of magnitude since its conception, the cost of sequencing remained too high for the routine sequencing of entire human genomes. A second generation of sequencing technology arose in the mid-2000s, based on performing and measuring millions of sequencing reactions in parallel. These technologies are collectively referred to as High Throughput Sequencing (HTS) or Next Generation Sequencing (NGS).

The first HTS technology developed is termed "pyrosequencing" and was commercialized by 454 Life Sciences[1]. In this method of sequencing, single-stranded DNA is captured by beads, amplified and loaded into picoliter reaction wells. Fluorescently labelled nucleotides are added to the reaction in a predefined or-

[1]Later acquired by Roche

der. When a labelled base is bound to the template DNA, a short pulse of light is released which can be detected with a CCD camera. After each reaction, the reagents are cleared before the next addition of bases. The captured images are analyzed in real time to determine the sequence of each template molecule [Wheeler et al., 2008]. This method of sequencing produces far more data per run than Sanger sequencing, generating up to 700 megabases of sequence per 23 hour run, with up to 1000bp read lengths[1].

A second massively parallel sequencing technology was developed from work begun by Balasubramanian and Klenerman at the University of Cambridge and subsequently commercialized by Solexa[2]. In this method, template DNA is ligated to sequences fixed on a slide. The template DNA is amplified in place to generate a cluster of molecules. The sequencing process occurs over a number of cycles. In each cycle reversibly-terminated nucleotides, each labelled with a fluorescent dye, is added to the reaction. An image is taken, then the dye and terminator are chemically removed to allow the reaction to proceed to the next base in the chain. The captured images are analyzed after the run, and the identity of which base was incorporated during each cycle is determined by the color of each cluster [Bentley et al., 2008]. This method now produces up to 600 gigabases per run, when 100bp reads are taken from both ends of a DNA fragment[3].

Other approaches include sequencing-by-ligation (SOLiD by Life Technologies, first used in Valouev et al. 2008, and Complete Genomics Drmanac et al. 2010). Recently single molecule methods requiring no DNA amplification have become available (PacBio, Eid et al. 2009). In principle these can give very long reads, beyond the effective limit of 1000bp for preceding technologies, but currently they are not cost and accuracy competitive.

The enormous volume of data generated by HTS instruments has allowed population surveys of human genome variation [1000 Genomes Project Consortium, 2010] and plans to sequence thousands of human cancers [The International Cancer Genome Consortium, 2010] and 10,000 vertebrate genomes *de novo* [Genome 10K Community of Scientists, 2009]. High throughput sequencing has also gen-

[1]This information was taken from http://454.com/ on July 16th, 2012
[2]Later acquired by Illumina, Inc
[3]This information was taken from http://www.illumina.com on July 16th, 2012

erated new analysis challenges. As whole genome sequencing is now a routine experimental measure, we need algorithms and software that can scale to match the data generated. This is particularly important for the computationally demanding *de novo* assembly problem.

1.2 Sequence Assembly

Even at the earliest stages of DNA sequencing, when the genomes sequenced were only a few kilobases in length, it was apparent that computers would be needed to help analyze the data. In [Staden, 1979] Roger Staden observes that "It became clear during the sequencing of bacteriophage ϕX174 DNA that it was necessary to use computers to handle and analyze the data". As sequencing technology has progressed over the last 30 years, computational techniques to analyze the data have developed in parallel. One of the fundamental computational problems in DNA sequence analysis is the reconstruction of a genome from a set of sequence reads. This problem is known as *de novo* assembly. For large genomes, billions of reads may be used in the assembly and the time and space efficiency of the algorithm is crucial. In this section we give an overview of assembly algorithms. These will be discussed in more technical detail in the following chapter.

Early assemblers were not fully automated but helped a user identify and merge overlapping sequence reads. The Staden package referenced above is an example of this approach. As the size of sequence data sets grew, fully automated assemblers needed to be developed. Early assemblers often used a greedy algorithm. Pairs of reads would be compared to find overlaps and each overlap would be scored based on the number of matching and mismatching bases. The reads sharing the highest scoring overlap would be merged together and the process would iterate. As this process made a greedy choice, care needed to be taken to avoid merging reads that originate from similar repeats. The Phrap program[1], designed for assemblies of BACs and used by the Human Genome Project, used base quality scores[2] to help distinguish between true overlaps and those caused by identical or nearly identical repeats.

[1] Unpublished, algorithm described here http://www.phrap.org/phredphrap/phrap.html
[2] An estimation of the probability that a given nucleotide in the read is incorrect

In a move away from greedy algorithms, Kececioglu and Myers modelled the assembly problem as a variant of the Shortest Common Superstring problem, which was known to be NP-Hard [Kececioglu and Myers, 1993]. Their formulation of the problem used the concept of an overlap graph. In an overlap graph, each sequence read is a vertex and two vertices are connected by an edge if their corresponding reads have a significant overlap. The assembly problem is thus to find walks through the graph that are consistent with the overlap relationships. A final consensus sequence can be computed from a multiple alignment constructed from the set of overlapping reads implied by the walk. The three stages of assembly - overlap computation, layout, consensus gave rise to the "OLC" acronym used to describe assemblers following this paradigm. The overlap computation stage is typically the computational bottleneck in the assembly. In the worst case this requires $O(N^2)$ time where N is the total number of bases in the sequence reads. In the Celera assembly of the human genome, they compared all reads against each other, which required 10,000 CPU hours on a cluster of 40 machines [Venter et al., 2001]. In [Myers, 2005], Gene Myers reformulated overlap graph assembly in terms of *string graphs*. Myers' string graph construction algorithm removes transitive edges from an overlap graph. As this requires the full overlap graph to be constructed it shares the same computational bottleneck as OLC assembly. Various strategies to accelerate overlap detection have been developed, including limiting overlap computation to pairs of reads that have an exact, short match [Rasmussen et al., 2006]. Despite such improvements, overlap computation remained a significant bottleneck. This became particularly important when High Throughput Sequencing instruments became widely available, as traditional overlap-based strategies could not cope with the volume of data.

In the late 1980s, a new approach to DNA sequencing was proposed in which DNA is hybridized to an array containing short oligonucleotide probes with known sequences. A signal is read from each probe, indicating whether or not the particular sequence is present in the genome. Thus, the assembly problem is to reconstruct the genome from the *spectrum* of short sequences that it contains. This approach to DNA sequencing, called Sequencing by Hybridization, did not have a significant impact on the *de novo* sequencing of genomes but in studying this problem a new class of assembly algorithms was developed, pioneered by

Pavel Pevzner [Pevzner, 1989; Pevzner et al., 2001] along with Idury and Waterman [Idury and Waterman, 1995]. The defining characteristic of these algorithms is that they break sequence reads into chains of consecutive k-mers[1], overlapping by $(k-1)$ bases, and construct a graph of the relationship between k-mers. Such assembly graphs are called *de Bruijn graphs* after the graphs used by Nicolaas de Bruijn to study combinatorial problems [de Bruijn, 1946]. As the construction of a de Bruijn graph only requires performing exact matches between k-mers, the construction can be performed in linear time using a hash table. The straightforward construction of the graph, along with the efficiency in which repetitive sequences are handled by the graph, led to the de Bruijn graph becoming the dominant method of assembly for high throughput, short read sequence data. This approach to assembly was initially applied to HTS data by Chaisson and Pevzner [Chaisson and Pevzner, 2008] and Zerbino and Birney [Zerbino and Birney, 2008].

While the de Bruijn graph approach to assembly solved the computation time problem, the amount of memory required to store the graph became a major concern. In the de Bruijn graph, there is a vertex for every unique k-mer in the genome. In addition, sequencing errors cause new, erroneous k-mers to be added. The de Bruijn graph of large genomes can have billions of vertices, requiring hundreds of gigabytes of memory. Reducing the memory requirements of de Bruijn graph assembly is a very active area of research. Simpson et al. [2009] designed a representation of the de Bruijn graph which could be distributed across a network of computers, spreading the memory load across multiple machines. Li et al. [2010c] performed error correction before assembly to reduce the number of vertices in the graph. Conway and Bromage proposed encoding the structure of the graph using sparse bit vectors [Conway and Bromage, 2011]. Recently Pell et al. have developed a probabilistic representation of a de Bruijn graph using a bloom filter [Pell et al., 2012]. Chikhi and Rizk have also recently used a bloom filter to represent a de Bruijn graph [Chikhi and Rizk, 2012].

[1] subsequences of a uniform length, k

1.2.1 A Practical Overview of Assembly

The input into a genome assembly is a set of sequence reads from the genome of interest. Often *paired-end* reads will be obtained. In paired-end sequencing, both ends of the DNA fragment will be read without reading the sequence between the ends. For example the first and last 100 bases of a 500 base fragment of DNA may be read - the 300bp sequence separating the ends is unknown. *Mate-pair* reads may also be obtained. In mate-pair sequencing a multi-kilobase fragment of DNA is circularized then cut twice, and the ends from the two cut points are read. This allows a wider separation between the sequenced pair, up to 3 to 10 kilobases. When discussing sequencing data, I will refer to the *coverage* of the genome. This is a measure of how redundantly the genome is sampled by the reads. For example sequence coverage of 40X indicates that on average each base of the genome is represented in 40 reads. This is often also called the sequencing *depth*.

The assembler will read in all available data and output a set of *contigs* and *scaffolds*. The contigs are the primary output of the assembly. These are stretches of the genome that have been completely assembled from the raw reads. They contain no gaps. When the assembler finds a repeat that it cannot resolve, or a contig cannot be extended due to a lack of reads covering the genome, the contig assembly stops. If paired-end or mate-pair reads are available the assembler can build scaffolds from the contigs. The paired reads provide long-range information that can jump over the coverage gap or unresolvable repeat. The scaffolds will contain multiple contigs separated by gaps. The gaps will be encoded using ambiguity symbols (typically runs of "N" symbols) which estimates the length of the unresolved sequence. The lengths of the scaffolds will typically be much greater than the lengths of the input contigs.

To measure the quality of assembly the *N50* length of the contigs or scaffolds can be calculated. The N50 of a set of contigs or scaffolds is the length x such that contigs of length x or greater contain half of the total length of the assembly. Misassembled contigs may inflate the N50 length of an assembly. If a reference genome is available we may calculate NG50 instead - this calculates the N50 length of segments of the genome that have been correctly assembled, which accounts

for the possibility of misassembled contigs or scaffolds [Earl et al., 2011].

1.2.2 The Topology of Assembly Graphs

The structure of the assembly graph can give important information about the underlying genome. Here I will describe common topological features of assembly graphs. The features below are common to both de Bruijn graphs and string graphs. Any distinctions will be noted.

1.2.2.1 Graph Tips

On the Illumina sequencing platform, errors are more likely to occur at the end of a read. When constructing a de Bruijn graph from the k-mers of a read containing sequencing errors, the k-mers covering the position of the error are typically unique. These k-mers will form a chain of vertices in the de Bruijn graph that terminates with the last k-mer in the read. As these branches of the graph are only connected on one end, they are typically termed *tips* of the graph. A simple tip is depicted in figure 1.1. In the string graph, these structures can also form. In this case, reads with sequencing errors will not have a suffix (or prefix) overlap. This will break the chain of overlapping reads, leading to a disconnected branch in the graph. The tips in a string graph will typically be shorter than those of a de Bruijn graph, as in the de Bruijn graph a single sequencing error may generate multiple erroneous vertices.

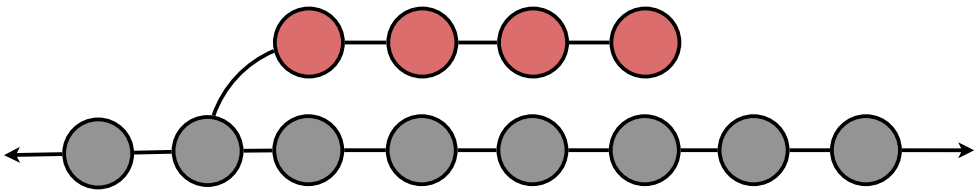


Figure 1.1: A simple tip in an assembly graph. The red vertices contain sequencing errors - due to these errors the sequence of this branch diverges from the rest of the graph (grey vertices). The arrows on the terminal grey vertices are to indicate the graph continues off-page.

A standard graph cleaning operation found in most graph-based assemblers is to identify these tips then trim them back to the point of divergence. In the example given in 1.1 this would remove the four red vertices, which would remove the ambiguity in the edge set of the second grey vertex.

1.2.2.2 Graph Bubbles

When the genome contains nearly-identical sequences structures known as *bubbles* form in the graph. When sequencing a diploid genome bubbles will form around heterozygous variants. The k-mers (or reads) covering the variants will cause a branch in the graph, with two possible paths to follow. The distinguishing feature of the bubble is that these two paths will collapse back together a short distance later. An example of a bubble is shown in figure 1.2.

Bubbles can also form around recurrent sequencing errors or inexact copies of repeats dispersed throughout a genome. As with tip removal, removing bubbles is a common operation performed on assembly graphs. Typically one of the two halves of the bubble will be deleted from the graph. A bubble removal algorithm is described in section 3.2.5. Detecting these structures in the graph will form the basis of the algorithms presented in Chapter 4.

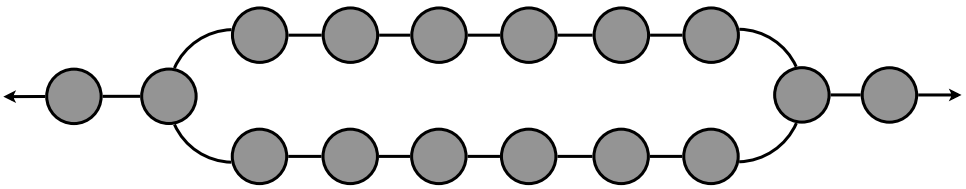


Figure 1.2: A bubble in the graph showing the distinctive divergence/collapsing signature.

1.2.2.3 Repeats

Genomes contain identical or nearly-identical *repeat* sequences. These sequences cause ambiguity in the assembly graph. Figure 1.3 depicts the simplest possible situtation where there are two exact copies of a repeat in a genome. The red nodes in this example graph indicate the repetitive sequence. The repeat segment of

the graph has two entry and exit points. This indicates there are four possible paths through this segment of the graph - *XRW*, *XRZ*, *YRW*, *YRZ* - only two of which are correct. In the absence of all other information we cannot resolve this repeat. When working with a de Bruijn graph, we may try to thread the original sequence reads through the graph in an attempt to resolve the repeat. If we can find reads that contain both X and W or X and Z it will help us determine the correct pair of paths traversing this segment of the graph. If the repeat is very long or has many copies in the genome, it is unlikely that it can be easily resolved. Paired end or mate pair data can be used to help resolve these cases.

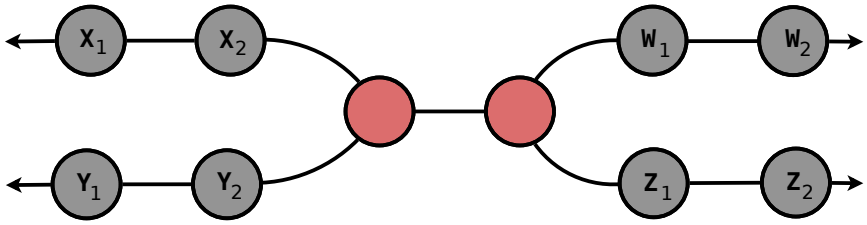


Figure 1.3: A simple repeat in the assembly graph. The red nodes represent sequences present multiple times in the genome.

1.2.2.4 Unipaths

Vertices in the graph that are connected to only one neighbor are *unambiguous*. Such vertices can be merged together into a single vertex without the loss of information or the possibility of making a misassembly. Chains of unambiguous vertices are referred to as *unipaths*. Finding unipaths is the primary method of *contig* assembly in most graph based assemblers. Figure 1.4 shows the unipath graph of the simple repeat from figure 1.3.

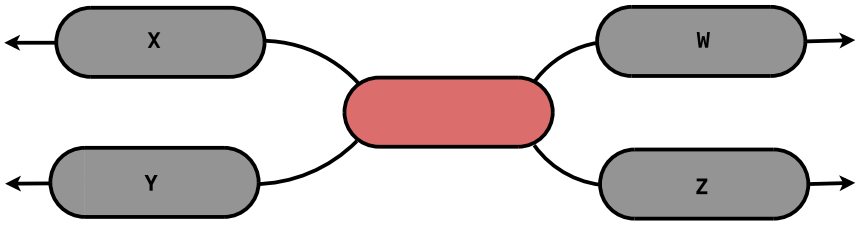


Figure 1.4: A unipath graph constructed from the graph depicted in figure 1.3. The unambiguously connected vertices have been merged together.

1.2.2.5 Assembly Software

The development of short read assemblers has been one of the most active areas of bioinformatics since the introduction of high-throughput sequencing. SSAKE was one of the first assemblers designed specifically for short reads [Warren et al., 2007]. The Velvet assembler [Zerbino and Birney, 2008] was introduced shortly afterwards and became very popular for assembling small-to-medium sized genomes. Velvet uses an explicit representation of the de Bruijn graph based on memory pointers in the C programming language. This representation of the graph requires large amounts of memory when the de Bruijn graph has many vertices, like in the case of assembling a human genome. The ABySS assembler was designed specifically for large genomes [Simpson et al., 2009]. ABySS does not use an explicit pointer-based de Bruijn graph, rather it stores a collection of k-mers in a sparse hash table from which the structure of the graph can be inferred. This representation of the graph allows the memory load to be distributed over a cluster of computers. ABySS was used in the Gorilla [Scally et al., 2012] and Tomato genome projects [Tomato Genome Consortium, 2012]. The SOAPdenovo assembler [Li et al., 2010c] was also designed for large genomes and reduces memory consumption by first performing k-mer based error correction to avoid adding erroneous k-mers to the de Bruijn graph. SOAPdenovo was one of the first short read assemblers to focus on the use of multiple large-insert mate-pair libraries for scaffold construction. This technique was used in the Giant Panda genome project [Li et al., 2010a]. The ALLPATHS series of assemblers is designed by the Broad Institute [Butler et al., 2008; Gnerre et al., 2011; Maccallum

et al., 2009]. ALLPATHS is unique in that it requires specific types of sequencing reads[1]. The algorithms are optimized for this input, allowing high-quality assemblies to be generated when the requirements are met [Gnerre et al., 2011]. In the Assemblathon competition [Earl et al., 2011], ALLPATHS-LG and SOAPdenovo were the top two entries out of seventeen groups who submitted assemblies. The assembler described in Chapter 3 of this thesis, SGA, was ranked third. ABySS was seventh.

The computational requirements for these popular assemblers varies widely. Velvet is very fast but cannot run on large data sets. For a human genome, ABySS requires 150-400GB of aggregate memory across the assembly cluster. SOAPdenovo requires a similar amount of memory after the error correction step. The authors of ALLPATHS-LG suggest the use of a multi-core 512GB server and estimate the run time to be 3.5 weeks for a human genome [Gnerre et al., 2011].

New assemblers have been developed recently with a focus on memory efficiency. Cortex [Iqbal et al., 2012] uses an approach similar to ABySS and encodes a de Bruijn graph using an efficient hash table of k-mers. Gossamer [Conway et al., 2012; Conway and Bromage, 2011] uses sparse bit arrays to represent the de Bruijn graph. Fermi [Li, 2012] uses modified versions of the algorithms described in this thesis to construct a string graph using the FM-index data structure.

1.3 Resequencing and Variant Calling

Often when we sequence an individual a reference genome for that species is already available. In the case of humans the individual is expected to match the reference at 99.9% of the bases. The *resequencing* problem is to discover the 1 in 1000 bases that differ between the individual and the reference. These differences can be *substitutions*, where the identity of a symbol is swapped with another symbol, or *indels*, where some bases have been inserted or deleted with respect to the reference. There are also larger *structural variants*. These are large deletions, copy number changes, inverted segments or chromosomal rearrangements. The process of finding the ways in which a sequenced genome differs from the reference is referred to as *variant calling*.

[1] Overlapping paired-end reads and at least one long-insert mate pair library

Standard approaches to variant calling will *map* the sequence reads to the reference genome. This is the process of finding the location on the reference genome that a sequence read (or read pair) was sampled from. The result of mapping is an *alignment* of a read against the reference, which is a one-to-one mapping from individual bases of the read to individual bases of the reference. The mapping and alignment process must take into consideration errors that occur during sequencing, for instance incorrectly identifying a base, and possible variants between the individual and the reference genome. Once all reads have been mapped to the reference, variants are called by finding consistent differences between the aligned bases and the reference genome. For example if the reference base is a T at a given position and all read bases aligned to that position are C, we may say there is a T to C substitution. The strength of the evidence for each variant will be typically assessed in a probabilistic model to distinguish between sequencing or alignment errors and true variants. A comprehensive description of the mathematics involved in the variant calling process can be found in [Li, 2011].

Variant calling by mapping reads to a reference genome is effective for isolated substitutions and small indels and has formed the basis of many resequencing projects, like the 1000 Genomes Project [1000 Genomes Project Consortium, 2010]. However, the mapping and alignment process can fail when there are significant differences between the individual and the reference genome, like multi-base substitutions or larger indels. In the worst case the sequence reads will be *misaligned* to the reference and false variant calls will be made. The primary source of misalignments (and therefore false positive variants) is polymorphic indels [Li and Homer, 2010]. For this reason, assembly-based methods of variant calling have recently been proposed [Iqbal et al., 2012; Li, 2012]. In Chapter 4, I will also address the variant calling problem with assembly-based algorithms.

1.4 Compressed Data Structures

A text *index* is a data structure which allows string queries to be performed without requiring scanning the full text. For example, given a text collection T and a pattern P, we may wish to check whether P is a substring of T, count

the number of occurrences of P in T or locate the positions of P in T. Text indices are used frequently in bioinformatics to accelerate searches over large sequence collections. Sequence alignment is a classic application of text indices. The BLAT [Kent, 2002] and SSAHA [Ning et al., 2001] algorithms use an index of short subsequences of a reference genome to find candidate mapping locations of a query sequence which are refined by dynamic programming.

Some of the most time efficient indices use suffix data structures. The suffix tree and suffix array data structures store an ordering of the suffixes of a text. The suffix tree can be searched for a pattern by matching the symbols of a pattern to labels on the edges of the tree. A suffix array uses considerably less space than a suffix tree and allows pattern matching via a binary search [Manber and Myers, 1990]. These data structures are extensively studied in Gusfield's classic sequence analysis text [Gusfield, 1997]. The suffix array is used widely in bioinformatics, for example in sequence clustering [Malde et al., 2003], repeat detection [Abouelhoda et al., 2002], microarray design [Li and Stormo, 2001] and k-mer counting [Kurtz et al., 2008].

Despite the widespread use of the suffix array, its memory requirements (8 bytes per input base for large sequence collections) has limited its use to relatively small analysis problems. In the past decade, *compressed* text indices related to the suffix array have been developed. These indices store a compressed version of the text, and define auxiliary data structures that allow the compressed text to be searched without the need for complete decompression. In 2000, Ferragina and Manzini described the FM-index (full-text, minute-space) [Ferragina and Manzini, 2000]. The FM-index is based on the Burrows-Wheeler transform [Burrows and Wheeler, 1994] and allows similar queries as the suffix array. Early applications of the FM-index in bioinformatics were for counting substrings in genomes [Healy et al., 2003] and finding local alignments between a query sequence and a reference genome [Lam et al., 2008]. The FM-index would later become the dominant data structure for mapping high throughput short reads to a reference genome [Langmead et al., 2009; Li and Durbin, 2009; Li et al., 2009]. The algorithms I develop in this thesis are based on the FM-index, and a full description of the data structures and how it is used is given in chapter 2. Compressed versions of the suffix array and suffix tree have been developed,

significantly shrinking their memory requirement [Grossi and Vitter, 2000]. As of yet, these structures have not been widely used within bioinformatics.

1.5 Overview of this work

In Chapter 2, I introduce my technical notation and definitions then describe an algorithm to efficiently construct the assembly string graph from a set of sequence reads, without the need to build the full overlap graph first. This will solve the time bottleneck for performing overlap-based assembly. The algorithm that I develop is based on the FM-index. As this is a compressed data structure, its memory footprint is much smaller than other full-text indices, like the suffix array. This will allow the reduction of the memory bottleneck of assembly. While the primary focus of this work is overlap-based assembly, in section 2.6 I will describe how the FM-index can be used as a memory efficient representation of a de Bruijn graph for all k.

In Chapter 3, I describe the implementation of these assembly algorithms into a fully functional sequence assembler called SGA. This assembler has algorithms for building the FM-index from very large sequence collections and correcting sequencing errors. I demonstrate the use of SGA on real sequence data from *Schizosaccharomyces pombe*, *Caenorhabditis elegans* and the human genome. Also, I compare the output of SGA to leading de Bruijn graph assemblers.

In chapter 4, I study the problem of finding differences between a pair of related genomes. I will address this problem by using assembly graphs to model the variation structure within the genomes. I will describe the use of both de Bruijn graphs and string graphs for this problem. Again, this work is based on the FM-index. This allows efficient access to the complete read sequences, which I will use when developing a probabilistic model to distinguish between sequence errors and true variation[1]. In Chapter 5, I study the sensitivity and accuracy of this approach with simulations. I also apply this approach to the problems of detecting newly acquired mutations in the offspring of two parents (*de novo* mutations), detecting somatically acquired mutations in cancer and detecting variation within a population of individuals.

[1]This work is performed in collaboration, details are provided at the start of Chapter 4

In Chapter 6, I offer concluding remarks including the prospects of expanding upon this work for upcoming sequencing technology.

Chapter 2

The FM-Index and Genome Assembly

2.1 Introduction

2.1.1 Publication Note

The work described in this chapter was previously published in [Simpson and Durbin, 2010]. Section 2.6 describes unpublished results. The work described is the sole work of the author, under the supervision of his PhD supervisor, Richard Durbin.

2.2 Definitions and Notation

Let X be a string of symbols $a_1, ..., a_l$ from an alphabet Σ. The length of X is denoted $|X|$. $X[i] = a_i$ is the i-th symbol of X and $X[i, j]$ is the substring $a_i, ..., a_j$. Let $X' = a_l, a_{l-i}, ..., a_1$ denote the reverse of X. When discussing lexicographically ordered alphabets, $b+1$ will refer to the next highest symbol in the alphabet after b.

If Y is a substring of X and $Y \neq X$, then we say that Y is a *proper substring* of X. A substring $X[k, |X|]$ is a *suffix* of X and a substring $X[1, k]$ is a *prefix* of

X. We will often refer to substrings of length n as a n-mers [1].

When discussing text indices, we will consider all strings to be terminated by a sentinel symbol \$ that is not in Σ and is lexicographically lower than all the symbols in Σ. In this work the DNA alphabet will be used. The lexicographic ordering of this alphabet and the sentinel is $\$ < A < C < G < T$.

2.2.1 Genomes and Sequence Reads

We define a *genome* to be a long string from the alphabet $\{A, C, G, T\}$ representing the complete DNA sequence of an individual, for simplicity ignoring potential subdivisions into chromosomes. A sequence *read* is a short substring from a genome. DNA is a double stranded molecule and sequence reads can originate from either strand. We use the notation $\overline{X}$ for the *reverse-complement* of a read X.

Reads may contain sequencing errors. These occur when the sequencing instrument incorrectly identifies a symbol (substitution error), or when a symbol is incorrectly inserted into, or deleted from, the string (indel error). We will occasionally assume that reads are error-free. It will be clearly stated when this is the case.

We say that two reads X and Y *overlap* if a prefix of X is equal to a suffix of Y or vice versa. If X and Y originate from opposite strands, they overlap if the reverse complement of one of them overlaps the other. If X (or $\overline{X}$) has the same sequence as Y then we say that the two reads are *identical*, or *duplicates*. If X (or $\overline{X}$) is a proper substring of Y, then we say that X is *contained* within Y. When two reads X and Y overlap, we can merge them into a new sequence Z which contains both X and Y. In this case we say that we have *assembled* X and Y. We will refer to the new sequences that result from assembly as *contigs*.

In a shotgun sequencing experiment a set of sequence reads is randomly sampled from a genome, G, with an unknown sequence. We will denote the indexed set of reads by $\mathcal{R}$, with the i-th read in the set denoted by $\mathcal{R}_i$. The *de novo* assembly problem is to reconstruct the sequence of G given only $\mathcal{R}$. If the sequence of

[1] Such substrings are also referred to as n-grams or n-tuples, particularly in Computer Science. We will use n-mer for consistency with most literature in the sequence assembly field.

G was drawn randomly from $\{A, C, G, T\}$ the assembly problem would be easy - even very short reads with length on the order of $\log |G|$ would suffice to assemble nearly all of G unambiguously. In reality, the problem is far more complicated. Eukaryotic genomes are shaped by the duplication and divergence of large segments, along with the proliferation of transposon elements. The difficulty of the assembly problem stems from these *repetitive* regions.

2.3 Assembly Graphs

To help reconstruct G from $\mathcal{R}$, we can build a graph of the relationships between sequence reads. We will discuss three different types of assembly graph - the overlap graph, the de Bruijn graph and the string graph. The common thread between these graphs is that the structure of the underlying genome is reflected in the structure of the graph. Walks through these graphs describe assemblies of the reads into segments of the genome. We begin with the overlap graph.

2.3.1 Overlap Graphs

In the overlap graph each sequence read in $\mathcal{R}$ is a vertex. Two vertices are joined by an edge if their corresponding reads overlap. To help distinguish true overlaps from spurious overlaps we set a threshold of τ_{min} on the minimum acceptable overlap length. When allowing for sequencing errors, a threshold on the maximum error rate of ϵ_{max} will also be set. For the remainder of this chapter, we will consider only error-free reads. This constraint will be relaxed in Chapter 3. We associate coordinates with each edge describing the matching segments of the linked reads.

The overlap graph is computationally demanding to construct. A naive algorithm (suitable only for very small sequencing projects) would compare all pairs of reads to discover overlapping pairs. Such an algorithm has $O(N^2)$ time complexity where $N = \sum_{i=1}^{|\mathcal{R}|} |\mathcal{R}_i|$. For larger sequencing projects, comparing all pairs of reads is impractical. To accelerate overlap detection, an index of all l-mer sequences appearing in the reads can first be constructed, and only reads sharing

an l-mer would be checked for an overlap, avoiding the need to compare all pairs. In Myers [2005] the use of a q-gram filter (subsequently described in Rasmussen et al. [2006]) is suggested for finding all (τ_{min}, ϵ_{max}) overlaps in $O(N^2/D)$ time where D is a function of the amount of memory available. In Gusfield [1997] an algorithm to solve the all-pairs maximal overlap problem in $O(N + |\mathcal{R}|^2)$ time using a suffix tree is described. The quadratic term is due to the requirement that an overlap between all pairs must be found - if we instead require that only τ_{min}-overlaps are found, a faster version of this algorithm is possible. However, the suffix tree requires a very large amount of memory to store [Abouelhoda et al., 2004], so in practice this data structure is not commonly used for indexing large sequence collections.

An optimal algorithm for overlap detection would require $O(N + |E|)$ time, where $|E|$ is the number edges in the resulting graph. Even with such an algorithm, overlap-based assemblers suffer from two computational problems. First, as all reads covering the same position of the genome will mutually overlap, the number of overlaps (and therefore edges in the graph) is quadratic in sequencing depth. This is a significant problem when assembling high-throughput sequence data as the genomes tend to be covered very deeply to ensure each base is covered multiple times (typically greater than 40 reads cover each base). Second, for reads originating from repetitive regions, the number of overlaps will be quadratic in the product of the sequencing depth and the number of copies of the repeat. This leads to greatly increased computational cost and a much larger graph when assembling highly repetitive genomes. For this reason, overlap assemblers occasionally take the step of masking known repeats and low-complexity sequence as a pre-processing step, as exemplified by Celera's assembly of the human genome [Venter et al., 2001].

As each read in $\mathcal{R}$ was sampled from a distinct location in G, the optimal solution will assign a linear ordering $\{1, 2, ..., |\mathcal{R}|\}$ to the elements of $\mathcal{R}$, reflecting their position along G. Such a solution requires finding a path through the overlap graph which visits each vertex exactly once. This is the Hamiltonian path problem which is known to be NP-complete. For this reason a global solution to the assembly problem is rarely sought. Instead, the assembly of reads into *contigs* typically focuses on finding local groups of reads that can be unambiguously

assembled together.

2.3.2 de Bruijn Graphs

We follow Pevzner's formulation of the de Bruijn graph [Pevzner et al., 2001]. Set a fixed value ρ and let $\mathcal{P}$ be the set of all distinct ρ-mers in $\mathcal{R}$. Let $k = \rho - 1$ and $\mathcal{V}$ be the set of all distinct k-mers in $\mathcal{R}$. $\mathcal{V}$ is the set of vertices of the graph. For each $P \in \mathcal{P}$ we create a bidirected edge $K_1 \leftrightarrow K_2$ where K_1 is the length-k prefix of P and K_2 is the length-k suffix of P. To handle the double-stranded nature of DNA, we can also introduce the reverse-complements of the ρ-mers and k-mers into the graph[1].

Unlike the overlap graph, the de Bruijn graph is computationally easy to construct. The construction of the vertex and edge set only requires iterating over distinct k or p-mers, which can easily be implemented with hash tables, sorted arrays of strings, red-black trees or any other data structure allowing efficient queries for whether a string is present in a collection. A second important property, perhaps the most important, is how repeats appear in the graph. Let R be a long repeated substring of G ($|R| > k$). By definition each instance of R contains the same sequence of k-mers. As the de Bruijn graph only contains *distinct* k-mers, there is a single vertex for each of these k-mers. Therefore unlike the overlap graph the repeat copies do not contribute extra edges to the graph - all copies are represented by a single segment of the graph (see figure 1.3). Coupled with the efficient construction algorithms, this property has made the de Bruijn graph the dominant data structure for assembly of genomes from high-throughput short read data.

The reconstruction of G from the de Bruijn graph requires a tour that visits each edge at least once. This is the route-inspection problem (also known as the Chinese Postman Problem). Pevzner proposed [2001] to introduce edge multiplicities in the graph to transform the problem into one in which each edge must be visited exactly once. This is a classic graph theory problem originally studied by Euler [Euler, 1741] and hence known as a Eulerian path problem.

[1] Some assemblers, like ABySS, represent a k-mer and its reverse complement as a single vertex

As the Eulerian path problem has a known polynomial-time algorithm [Fleury, 1883], this was a promising approach to reducing the computational complexity of genome assembly. However, with long repeats in the genome the problem is underconstrained - many possible solutions may exist with only one representing the true sequence of G [Nagarajan and Pop, 2009]. For this reason, assembling contigs from the de Bruijn graph mainly focuses on local segments of the graph that can be unambiguously assembled, like in the case of the overlap graph.

2.3.3 The String Graph

As a refinement to the overlap graph, Myers formulated the String Graph [2005]. The String Graph has a number of important differences with the overlap graph. First, we remove duplicated or contained reads from $\mathcal{R}$ to provide a new non-redundant vertex set. Second, we label each edge with the unmatched substrings of each read. Let $X = X_1X_2$ and $Y = Y_1Y_2$ be two overlapping reads. When X and Y are from the same sequencing strand, either $X_2 = Y_1$ or $X_1 = Y_2$. When X and Y are from opposite sequencing strands, either $\overline{X_1} = Y_1$ or $\overline{X_2} = Y_2$. We will call the substrings that are found in both X and Y the *matched* substrings. The other substrings are the *unmatched* substrings. We define the labels of an edge to be the unmatched substrings of the reads. Specifically:

$$L_{xy} = \begin{cases} Y_2 & \text{if } X_2 = Y_1 \\ Y_1 & \text{if } X_1 = Y_2 \\ \overline{Y_2} & \text{if } X_1 = \overline{Y_1} \\ \overline{Y_1} & \text{if } X_2 = \overline{Y_2} \end{cases}$$

The reciprocal label L_{yx} is defined similarly. Note that in the case that X and Y are from opposite sequencing strands, then the label is the reverse-complement of the unmatched substring. The concatenation of X and L_{xy} is an assembly of reads X and Y - the resulting string contains both the sequence of X and Y. As there are no duplicated or contained reads in the graph, the labels are necessarily non-empty strings.

We can also associate with each edge in the graph a *type* describing the relationship between the pair of reads it links.

$$type_{xy} = \begin{cases} S & \text{if } X_2 = Y_1 \text{ or } X_2 = \overline{Y_2} \\ P & \text{if } X_1 = Y_2 \text{ or } X_1 = \overline{Y_1} \end{cases}$$

If a suffix of X overlaps a prefix of Y, we will call the edge an SP-edge. Likewise when a prefix of X overlaps a suffix of Y, we will call the edge a PS-edge. When a reverse-complemented prefix (suffix) of X overlaps a prefix (suffix) of Y, we call the edge a PP-edge (SS-edge). We note that when $type_{xy} = type_{yx}$ X and Y are necessarily from opposite sequencing strands.

Walks through the graph must respect the edge types. For example, for the walk $X \leftrightarrow Y \leftrightarrow Z$ to be valid, if $type_{yx} = S$ then $type_{yz}$ must be P and vice versa. In other words, if we enter a vertex via a suffix overlap, we must leave the vertex using a prefix overlap. This makes the string graph *bidirected*. We can associate a string with a walk through the graph by concatenating the label of each edge in the walk to the sequence of the first vertex in the walk.

Definition 1. *Let $X_1 \leftrightarrow X_2 \leftrightarrow ... \leftrightarrow X_n$ be a valid walk through the edge-labelled graph. We assume that $X_1, X_2, ..., X_n$ are from the same sequencing strand. If not, we preprocess the walk by changing the strand of each edge label to match the strand of X_1. After such a step, we define the string corresponding to the walk to be:*

$$A_{x_1 x_2 ... x_n} = \begin{cases} X_1 L_{x_1 x_2} ... L_{x_{n-1} x_n} & \text{if } type_{x_1 x_2} = S \\ L_{x_{n-1} x_n} ... L_{x_1 x_2} X_1 & \text{if } type_{x_1 x_2} = P \end{cases}$$

The final difference between the string graph and the overlap graph is that *transitive* edges are removed.

Definition 2. *Consider a read X that overlaps reads Y and Z, which mutually overlap. The initial overlap graph will contain the edges $X \leftrightarrow Y$, $X \leftrightarrow Z$ and $Y \leftrightarrow Z$. We will say an edge $X \leftrightarrow Z$ is* transitive *when the string spelled by path $X \leftrightarrow Z$ is the same as the string spelled by path $X \leftrightarrow Y \leftrightarrow Z$.*

Transitive edges can be removed from the graph without reducing the set of strings that can be spelled by the graph. We will refer to non-transitive edges as *irreducible*. We will now define useful properties of transitive edges. For

the following properties we will assume without loss of generality that $type_{xy} = type_{xz} = S$ and all three reads are from the same strand. This implies $X \leftrightarrow Y$, $X \leftrightarrow Z$ and $Y \leftrightarrow Z$ are all SP-edges.

Property 1. *The label of the transitive edge $X \leftrightarrow Z$ is the concatenation of the edge labels of the walk $X \leftrightarrow Y \leftrightarrow Z$.*

Property 2. *L_{xy} is a prefix of L_{xz}.*

Proof. From the definition of a transitive edge $A_{xz} = A_{xyz}$. From definition 1, $A_{xz} = XL_{xz}$ and $A_{xyz} = XL_{xy}L_{yz}$ therefore $L_{xz} = L_{xy}L_{yz}$ and L_{xy} is a prefix of the transitive edge.

□

Property 3. *Let C be the matched substring between X and Z. C is also a substring of Y.*

Proof. We can write X and Z in terms of C as $X = X'C$ and $Z = CL_{xz}$. Assume C is not a substring of Y and let $C = C_1C_2$ where C_2 is the prefix of Y that matches a suffix of X and C_1 is not empty. Write Y and Z in terms of these substrings as $Y = C_2L_{xy}$ and $Z = C_1C_2L_{xz}$. From property 1 we have $L_{xz} = L_{xy}L_{yz}$ therefore $Z = C_1C_2L_{xy}L_{yz}$. This implies that Y is contained within Z which contradicts a precondition on the graph therefore C must be a substring of Y.

□

Property 4. *C is not a prefix of Y*

Proof. The proof is similar to that of property 3 except we assume the prefix of Y is C to arrive at a contradiction.

□

Property 5. *The overlap between X and Y is longer than the overlap between X and Z.*

Proof. Again let C be the matched substring between X and Z. The length of C is the length of the overlap between X and Z. As C is a substring of Y but

not a prefix of Y, the matched substring between X and Y is MC where M is non-empty. The length of MC is therefore greater than the length of C.

□

An example string graph built from three overlapping reads is given in figure 2.1.

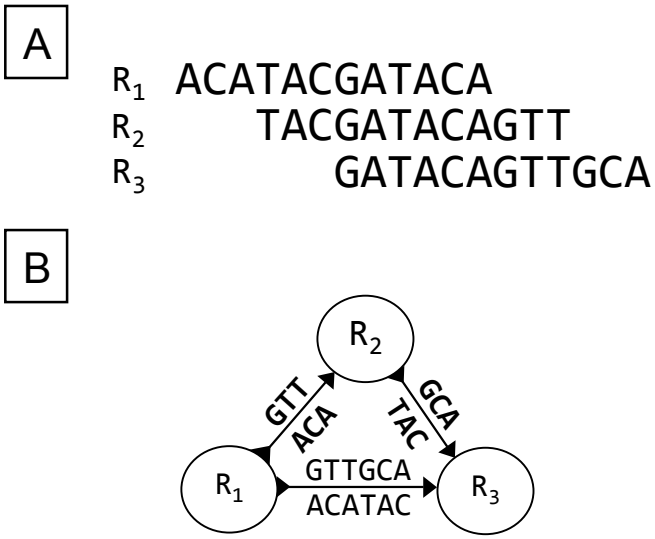


Figure 2.1: Diagram of a simple assembly graph. Three overlapping reads (R_1, R_2, R_3) are shown in panel A. Panel B shows the graph constructed from the overlaps between the reads. The arrowheads pointing into the nodes depict an edge of type P and arrowheads pointing away from the nodes depict edges of type S. For example the edge between R_1 and R_2 is a SP-edge. The edge $R_1 \leftrightarrow R_3$ is transitive. Removing this edge will turn the graph into a string graph.

Transforming an overlap graph into a string graph by removing duplicated and contained reads, along with transitive edges, avoids the quadratic expansion of edges with sequencing depth and repeat copy number. Like in the de Bruijn graph, repeats are collapsed to single segments in the graph. The string graph therefore represents an alternative to the de Bruijn graph, with the important benefit that the graph contains the full read sequences, representing the complete information present in $\mathcal{R}$.

The string graph can be built indirectly by first constructing an overlap graph then removing duplicate and contained reads, then removing transitive edges. Myers provides an $O(|E|)$ expected-time algorithm to perform transitive reduction on an existing overlap graph Myers [2005]. The fundamental problem of overlap assembly remains however, in that the computation of the overlap graph is the computational bottleneck. This is the problem that we address in this chapter by devising an algorithm to *directly* output the string graph, without the need to transitively reduce an overlap graph. This algorithm will allow us to construct the graph in linear-time, bringing the algorithmic complexity of the string graph in line of that of the de Bruijn graph. We begin the description of this algorithm with an introduction to text indices.

2.4 The Suffix Array, BWT and FM-Index

The *suffix array* data structure was introduced by Manber and Myers [1990] as a succinct representation of the lexicographic ordering of the suffixes of a string. The suffix array of a string X, denoted $\mathbf{SA}_X$, is a permutation of the integers $\{1, 2, ..., |X|\}$ such that $\mathbf{SA}_X[i] = j$ iff $X[j, |X|]$ is the i-th lexicographically lowest suffix of X. For example, if $X = \text{AAGTA\$}$ then $\mathbf{SA}_X = [6, 5, 1, 2, 3, 4]$. Since the suffix array is a sorted data structure, the start positions of all the instances of a pattern Q in X will occur in an interval in $\mathbf{SA}_X$. We refer to such an interval as a *suffix array interval* and associate with it a pair of integers $[l, u]$ denoting the first and last index in $\mathbf{SA}_X$ that correspond to a position in X of an instance of Q. Using $\mathbf{SA}_X$ and the original string X, l and u can be efficiently found with a binary search for Q. Ferragina and Manzini [2000] developed a related method of indexing text, called the FM-index, which requires considerably less memory than a suffix array and can compute l and u in $O(|Q|)$ time, independent of the size of the text being searched. Central to the FM-index is the Burrows-Wheeler transform (BWT). Originally developed for text compression [Burrows and Wheeler, 1994] the Burrows-Wheeler transform of X, denoted $\mathbf{B}_X$, is a permutation of the symbols of X such that:

$$\mathbf{B}_{\mathrm{X}}[i] = \begin{cases} X[\mathbf{SA}_{\mathrm{X}}[i] - 1] & \text{if } \mathbf{SA}_{\mathrm{X}}[i] > 1 \\ \$ & \text{if } \mathbf{SA}_{\mathrm{X}}[i] = 1 \end{cases}$$

Restated, $\mathbf{B}_{\mathrm{X}}[i]$ is the symbol preceding the first symbol of the suffix starting at position $\mathbf{SA}_{\mathrm{X}}[i]$. For the example string X from above, $\mathbf{B}_{\mathrm{X}} = AT\$AAG$.

Ferragina and Manzini extended the BWT representation of a string by adding two additional data structures to create a structure known as the FM-index. Let $\mathbf{C}_{\mathrm{X}}(a)$ be the index in $\mathbf{SA}_{\mathrm{X}}$ of the first suffix starting with symbol a. If v is the number of symbols lexicographically lower than a in X, then $\mathbf{C}_{\mathrm{X}}(a) = v + 1$. Let $\mathbf{Occ}_{\mathrm{X}}(a, i)$ be the number of occurrences of the symbol a in $\mathbf{B}_{\mathrm{X}}[1, i]$[1]. We note that $\mathbf{C}_{\mathrm{X}}$ and $\mathbf{Occ}_{\mathrm{X}}$ include counts for the sentinel symbol, \$.

$\mathbf{C}_{\mathrm{X}}(a)$ for the example string X is:

a	\$	A	C	G	T
$\mathbf{C}_{\mathrm{X}}(a)$	1	2	5	5	6

$\mathbf{Occ}_{\mathrm{X}}(a, i)$ for X is:

a	\$	A	C	G	T
$\mathbf{Occ}_{\mathrm{X}}(a, 1)$	0	1	0	0	0
$\mathbf{Occ}_{\mathrm{X}}(a, 2)$	0	1	0	0	1
$\mathbf{Occ}_{\mathrm{X}}(a, 3)$	1	1	0	0	1
$\mathbf{Occ}_{\mathrm{X}}(a, 4)$	1	2	0	0	1
$\mathbf{Occ}_{\mathrm{X}}(a, 5)$	1	3	0	0	1
$\mathbf{Occ}_{\mathrm{X}}(a, 6)$	1	3	0	1	1

Using $\mathbf{C}_{\mathrm{X}}(a)$ and $\mathbf{Occ}_{\mathrm{X}}(a, 1)$, Ferragina and Manzini provided an algorithm to search for a string Q in X. Let S be a string whose suffix array interval is

[1]These definitions use 1-based coordinates. When implementing these data structures 0-based coordinates are preferred. To allow this, we modify the definition of $\mathbf{C}_{\mathrm{X}}(a)$ to equal v and $\mathbf{Occ}_{\mathrm{X}}(a, i)$ to count over $\mathbf{B}_{\mathrm{X}}[0, i]$. The following algorithms work in either case.

known to be $[l, u]$. The interval for the string aS can be calculated from $[l, u]$ using $\mathbf{C}_{\mathrm{X}}$ and $\mathbf{Occ}_{\mathrm{X}}$ by the following:

$$l' = \mathbf{C}_{\mathrm{X}}(a) + \mathbf{Occ}_{\mathrm{X}}(a, l-1) \tag{2.1}$$

$$u' = \mathbf{C}_{\mathrm{X}}(a) + \mathbf{Occ}_{\mathrm{X}}(a, u) - 1 \tag{2.2}$$

We encapsulate equations (2.1) and (2.2) in the following algorithm, `updateBackward`.

Algorithm 1 updateBackward($[l, u]$, a)

$l \leftarrow \mathbf{C}_{\mathrm{X}}(a) + \mathbf{Occ}_{\mathrm{X}}(a, l-1)$
$u \leftarrow \mathbf{C}_{\mathrm{X}}(a) + \mathbf{Occ}_{\mathrm{X}}(a, u) - 1$
return $[l, u]$

To search for a string Q, we need to first calculate the interval for the last symbol in Q then use equations (2.1) and (2.2) to iteratively calculate the interval for the remainder of Q. The initial interval for a single symbol a is simply $[\mathbf{C}_{\mathrm{X}}(a), \mathbf{C}_{\mathrm{X}}(a{+}1){-}1]$ where $a{+}1$ denotes the next largest symbol in the alphabet[1]. The `backwardsSearch` algorithm presents the searching procedure in detail. If `backwardsSearch` returns an interval where $l > u$, Q is not contained in X otherwise $\mathbf{SA}_{\mathrm{X}}[i]$ is the position in X of each occurrence of Q for $l \leq i \leq u$.

Algorithm 2 backwardsSearch(Q) - find the interval in $\mathbf{SA}_{\mathrm{X}}$ for the pattern Q

$i \leftarrow |Q|$
$l \leftarrow \mathbf{C}_{\mathrm{X}}(Q[i])$
$u \leftarrow \mathbf{C}_{\mathrm{X}}(Q[i]+1) - 1$
$i \leftarrow i - 1$
while $l \leq u$ & $i \geq 1$ **do**
  $[l, u] \leftarrow$ `updateBackward`$([l, u], Q[i])$
  $i \leftarrow i - 1$
return $[l, u]$

The `backwardsSearch` algorithm requires updating the suffix array interval $|Q|$ times. As each update is a constant-time operation, the complexity of `backwardsSearch` is $O(|Q|)$ given that the FM-index is already constructed.

[1] If a is the largest symbol in Σ, then $\mathbf{C}_{\mathrm{X}}(a+1)$ simply returns $n+1$ where n is the highest index in $\mathbf{SA}_{\mathrm{X}}$

2.4.1 The Generalized Suffix Array

We can easily expand the definition of a suffix array to include sets of strings. Let $\mathcal{T}$ be an indexed set of strings and $\mathcal{T}_i$ be element $\mathcal{T}[i]$. We define $\mathbf{SA}_\mathcal{T}[i] = (j, k)$ iff $\mathcal{T}_j[k, |\mathcal{T}_j|]$ is the i-th lowest suffix in $\mathcal{T}$. In the generalized suffix array, unlike the suffix array of a single string, two suffixes can be lexicographically equal. We break ties in this case by comparing the indices of the strings. In other words we treat each string in $\mathcal{T}$ as if it was terminated by a unique sentinel character $\$_i$ where $\$_i < \$_j$ when $i < j$. We extend the definition of the Burrows-Wheeler transform to collections of strings as follows. Let $\mathbf{SA}_\mathcal{T}[i] = (j, k)$ then:

$$\mathbf{B}_\mathcal{T}[i] = \begin{cases} \mathcal{T}_j[k-1] & \text{if } k > 1 \\ \$ & \text{if } k = 1 \end{cases}$$

Like the BWT of a single string, $\mathbf{B}_\mathcal{T}$ is a permutation of the symbols in $\mathcal{T}$; therefore the definitions of the auxiliary data structures for the FM-index, $\mathbf{C}_\mathcal{T}(a)$ and $\mathbf{Occ}_\mathcal{T}(a, i)$, do not change.

2.5 Direct Construction of the String Graph

In this section we describe the first results of this work, string graph construction algorithms based on the FM-index of a set of reads. We will show that by using the FM-index of $\mathcal{R}$ the set of overlaps can be computed in $O(N + C)$ time for error-free reads where C is the total number of overlaps found. We then provide an algorithm which detects only the overlaps for irreducible edges - removing the need for the transitive reduction algorithm and allowing the direct construction of the string graph.

2.5.1 Building an FM-index from a set of sequence reads

To build the FM-index of $\mathcal{R}$, we can first compute the generalized suffix array of $\mathcal{R}$. We could do this by creating a string which is the concatenation of all members of $\mathcal{R}$, $S = \mathcal{R}_1\mathcal{R}_2...\mathcal{R}_m$ and then use one of the well-known efficient suffix array construction algorithms to compute $\mathbf{SA}_\mathrm{S}$ [Puglisi et al., 2007]. We have adopted a different strategy and have modified the induced-copying suffix array

construction algorithm [Nong et al., 2009] to handle an indexed set of strings $\mathcal{R}$ where each suffix array entry is a pair (j, k) as described in section 2.4.1. This suffix array construction algorithm is similar to the Ko-Aluru algorithm [2005]. A set of substrings of the text (termed LMS substrings) is sorted from which the ordering of all the suffixes in the text is induced. Our algorithm differs from the Nong-Zhang-Chan algorithm as we directly sort the LMS substrings using multikey quicksort [Bentley and Sedgewick, 1997] instead of sorting them recursively. This method of construction is fast in practice as typically only $30 - 40\%$ of the substrings must be directly sorted. Once $\mathbf{SA}_{\mathcal{R}}$ has been constructed, the Burrows-Wheeler transform of $\mathcal{R}$, and hence the FM-Index is easily computed as described above. We also compute the FM-index for the set of *reversed* reads, denoted $\mathcal{R}'$, which is necessary to compute overlaps between reverse complemented reads. We also output the *lexicographic index* of $\mathcal{R}$, which is a permutation of the indices $\{1, 2, ..., |\mathcal{R}|\}$ of $\mathcal{R}$ sorted by the lexicographic order of the strings. This can be found directly from $\mathbf{SA}_{\mathcal{R}}$ and is used to determine the identities of the reads in $\mathcal{R}$ from the suffix array interval positions once an overlap has been found.

Alternatively, when all reads in $\mathcal{R}$ are short ($\approx$ 100bp) then the Bauer-Cox-Rosone algorithm [Bauer et al., 2011] can be used to construct $\mathbf{B}_{\mathcal{R}}$. This topic will be revisited in 3.2.1 when discussing our software implementation.

2.5.2 Overlap detection using the FM-Index

We now consider the problem of computing the set of τ_{min} overlaps between reads in $\mathcal{R}$. Consider two reads X and Y. If a suffix of X matches a prefix of Y a SP-edge will be created in the initial overlap graph. We will describe a procedure to detect overlaps of this type from the FM-index of $\mathcal{R}$. Let X be an arbitrary read in $\mathcal{R}$. If we perform the `backwardsSearch` procedure on the string X, after k steps we have calculated the interval $[l, u]$ for the suffix of length k of X. The reads indicated by the suffix array entries in $[l, u]$ therefore have a substring that matches a suffix of X. Our task is to determine which of these substrings are prefixes of the reads. Recall that if a given element in the suffix array, $\mathbf{SA}_{\mathcal{R}}[i]$, is a prefix of a string then $\mathbf{SA}_{\mathcal{R}}[i] = (j, 1)$ for some j and $\mathbf{B}_{\mathcal{R}}[i] = \$$ by definition. Therefore, if we know the suffix array interval for a string Q, the interval for the

strings beginning with Q can be determined by calculating the interval for the string $\$Q$ using equations (2.1) and (2.2). This interval, denoted $[l_\$, u_\$]$, indicates that the reads with prefix Q are the $l_\$$-th to $u_\$$-th lexicographically lowest strings in $\mathcal{R}$. We can therefore recover the indices in $\mathcal{R}$ of the reads overlapping X using lexicographic index of $\mathcal{R}$. The algorithm is presented below in `findOverlaps`.

Algorithm 3 findOverlaps(X, τ) - determine the reads in $\mathcal{R}$ that overlap X by at least τ symbols

```
i ← |X|
l ← C_R(X[i])
u ← C_R(X[i] + 1) − 1
i ← i − 1
while l ≤ u & i ≥ 1 do
  if |X| − i + 1 ≥ τ then
    [l_$, u_$] ← updateBackwards([l, u], $)
    if l_$ ≤ u_$ then
      outputOverlaps(X, [l_$, u_$])
  [l, u] ← updateBackward([l, u], X[i])
  i ← i − 1
if l ≤ u then
  outputContained(X, [l, u])
```

The `findOverlaps` algorithm is similar to the backwards search procedure presented in section 2.4. It begins by initializing $[l, u]$ to the interval containing all suffixes that begin with the last symbol of X. The interval $[l, u]$ is then iteratively updated for longer suffixes of X. When the length of the suffix is at least the minimum overlap size, τ, we determine the interval for the reads that have a prefix matching the suffix of X and output an overlap record for each entry (using the subroutine `outputOverlaps`). When the update loop terminates, $[l, u]$ holds the interval corresponding to the full length of X. The `outputContained` procedure writes a containment record for X if X is contained by any read in $[l, u]$. The overlaps detected by `findOverlaps` correspond to SP-edges. We must also calculate the overlaps for SS-edges and PP-edges, which arise from overlapping reads originating from opposite strands. To calculate SS-edges we use `findOverlaps` on the complement of X (not reversed) and the FM-index of $\mathcal{R}'$. Similarly, to calculate PP-edges we use `findOverlaps` on $\overline{X}$ (the reverse

complement of X) and the FM-index of $\mathcal{R}$.

In rare cases, multiple valid overlaps may occur between a pair of reads. In this case the interval set returned by `findOverlaps` will contain intersecting intervals. To account for this, we sort the intervals and only keep the interval representing a maximal overlap when two adjacent intervals intersect.

The overlap records created by `outputOverlaps` are constructed in constant time as they only require a lookup in the lexicographic index of $\mathcal{R}$. Let c_i be the number of overlaps for read $\mathcal{R}_i$. The `findOverlaps` algorithm makes at most $|\mathcal{R}_i|$ calls to `updateBackwards` and a total of c_i iterations in `outputOverlaps` for a total complexity of $O(|\mathcal{R}_i| + c_i)$. For the entire set $\mathcal{R}$, the complexity is $O(N + C)$ where $C = \sum_{i=1}^{|\mathcal{R}|} c_i$. Note that the majority of these edges are transitive and subsequently removed. We can therefore improve this algorithm by only outputting the set of irreducible edges, allowing the direct construction of the string graph. We address this in the next section.

2.5.3 Detecting irreducible overlaps

To directly construct the string graph, we must only output irreducible edges. Recall from section 2.3.3 that the labels of the irreducible edges for a given read are prefixes of the labels of transitive edges. We use this fact to differentiate between irreducible and transitive edges during the overlap computation. Consider a read X and the set of reads that overlap a suffix of X, $\mathcal{O}$. We could devise an algorithm to find the subset consisting only of irreducible edges by calculating the edge-labels of all members of $\mathcal{O}$ and filtering out the members whose label is the extension of the label of some other read. This would require iterating over all members of $\mathcal{O}$ which can be quite large for repetitive reads or high-depth data. We will now show that the labels of the irreducible edges can be constructed directly from the suffix array intervals using the FM-index.

Consider a substring S that occurs in $\mathcal{R}$ and its suffix array interval $[l, u]$. Let a *left extension* of S be a string of length $|S| + 1$ of the form aS. We can use $\mathbf{B}_{\mathcal{R}}[l, u]$ to determine the set of left extensions of S. Let $\mathcal{B}$ be the set of symbols that appear in the substring $\mathbf{B}_{\mathcal{R}}[l, u]$. The left extensions of S are

the strings aS such that $a \in \mathcal{B}$. Note that we do not have to iterate over the range $\mathbf{B}_\mathcal{R}[l, u]$ to determine $\mathcal{B}$. Since $\mathbf{Occ}_\mathcal{R}(a, i)$ is defined to be the number of times symbol a occurs in $\mathbf{B}_\mathcal{R}[1, i]$ we can count the number of occurrences of a in $\mathbf{B}_\mathcal{R}[l, u]$ (and hence aS in $\mathcal{R}$) in constant time by taking the difference $\mathbf{Occ}_\mathcal{R}(a, u) - \mathbf{Occ}_\mathcal{R}(a, l-1)$. If the \$ symbol occurs in $\mathbf{B}_\mathcal{R}[l, u]$ we say that S is *left terminal*, in other words one of the elements of $\mathcal{R}$ has S as a prefix. We similarly define a *right extension* of S as a string of length $|S|+1$ of the form Sa. While we cannot build the right extensions of S directly from the FM-index, the right extensions of S are equivalent to left extensions of S' (the reverse of S) in $\mathcal{R}'$. Let S be *right terminal* if \$ exists in $\mathbf{B}_{\mathcal{R}'}[l', u']$, in other words S is a suffix of some string in $\mathcal{R}$.

The procedure to find all the irreducible edges of a read X and construct their labels is to find all the intervals containing the prefixes of reads that overlap a suffix of X, then iteratively extend them rightwards until a right-terminal extension is found. The terminated read forms an irreducible edge with X and the label of the edge is the sequence of bases that were used during the right-extension. All non-terminated strings with the same sequence of extensions are transitive and therefore not considered further.

The algorithm requires searching the FM-index in two directions, first backwards to determine the intervals of overlapping prefixes and then forwards to extend those prefixes and build the irreducible labels. Naively this would require first determining the intervals $[l, u]$ for each matching prefix, P, and then reversing the prefix and performing a backwards search on the FM-index of $\mathcal{R}'$ to find the interval $[l', u']$ for P'. The intervals $[l', u']$ would then be used in the extension stage to determine the labels of the irreducible edges. We can do better however by noting that the interval $[l', u']$ can be calculated directly during the backwards search without using the FM-index of $\mathcal{R}'$. We define $\mathbf{OccLT}_\mathcal{R}(a, i)$ to be the number of symbols that are lexicographically lower than a in $\mathbf{B}_\mathcal{R}[1, i]$. Let $S = X[i, |X|]$ be a suffix of X and $[l_i, u_i]$ its suffix array interval. Suppose we know the interval $[l'_i, u'_i]$ for S' in $\mathcal{R}'$. Let $a = X[i-1]$. The interval for $S'a = [l'_{i-1}, u'_{i-1}]$ is therefore:

$$l'_{i-1} = l'_i + (\mathbf{OccLT}_\mathcal{R}(a, u_i) - \mathbf{OccLT}_\mathcal{R}(a, l_i - 1)) \tag{2.3}$$

$$u'_{i-1} = l'_{i-1} + (\mathbf{Occ}_{\mathcal{R}}(a, u_i) - \mathbf{Occ}_{\mathcal{R}}(a, l_i - 1) - 1) \tag{2.4}$$

The interval for $X'[1]$ is identical to that of $X[|X|]$ since $\mathbf{B}_{\mathcal{R}}$ and $\mathbf{B}_{\mathcal{R}'}$ are both permutations of symbols in $\mathcal{R}$ therefore $\mathbf{C}_{\mathcal{R}} = \mathbf{C}_{\mathcal{R}'}$. We can therefore initialize the interval $[l', u']$ to the same initial value of $[l, u]$ and perform a forward search of X' simultaneously while performing a backward search of X using only the FM-index of $\mathcal{R}$. This does not require any additional storage as the $\mathbf{OccLT}_{\mathcal{R}}$ array can easily be computed from $\mathbf{Occ}_{\mathcal{R}}$ by summing the values for symbols less than a. This procedure is similar to the 2way-BWT search recently proposed by Lam *et al.* (2009) . The `updateFwdBwd` algorithm implements equations (2.3) and (2.4) along with `updateBackward` to calculate the pair of intervals. The $\mathcal{F}$ parameter to `updateFwdBwd` indicates the FM-index used - that of $\mathcal{R}$ or $\mathcal{R}'$.

Algorithm 4 updateFwdBwd($[l, u, l', u']$, a, $\mathcal{F}$)

$l' \leftarrow l' + (\mathbf{OccLT}_{\mathcal{F}}(a, u) - \mathbf{OccLT}_{\mathcal{F}}(a, l - 1))$
$u' \leftarrow l' + (\mathbf{Occ}_{\mathcal{F}}(a, u) - \mathbf{Occ}_{\mathcal{F}}(a, l - 1) - 1)$
$[l, u] \leftarrow$ `updateBackwards`$(l, u, a, \mathcal{F})$
return $[l, u, l', u']$

We now give the full algorithm for detecting the irreducible overlaps for a read X. The algorithm is performed in two stages, first a backwards search on X is performed to collect the set of interval pairs, denoted $\mathcal{I}$, for prefixes that match a suffix of X. This algorithm is presented in `findIntervals` below and is conceptually similar to `findOverlaps`.

The interval set found by `findIntervals` is processed by `extractIrreducible` to find the intervals corresponding to the irreducible edges of X. This algorithm has two parts. First, the set of intervals is tested to see if some read in the interval set is right terminal. If so, the intervals corresponding to the right terminal reads form irreducible edges with X and are returned. If no interval has terminated, we create a subset of intervals for each right extension of $\mathcal{I}$ and recursively call `extractIrreducible` on each subset.

The algorithm above assumes that $\mathcal{R}$ does not have any contained reads. If this is not the case, a slight modification must be made. If the set of reads overlapping X includes a read that is a proper substring of some other read it is possible that

Algorithm 5 findIntervals(X, τ)

$\mathcal{I} \leftarrow \emptyset$
$i \leftarrow |X|$
$l \leftarrow C(X[i])$
$u \leftarrow C(X[i]+1)-1$
$[l', u'] \leftarrow [l, u]$
$i \leftarrow i-1$
while $l \leq u$ & $i \geq 1$ **do**
  if $|X| - i + 1 \geq \tau$ **then**
    $[l_\$, u_\$, l'_\$, u'_\$] \leftarrow \texttt{updateFwdBwd}([l, u, l', u'], \$, \mathcal{R})$
    if $l_\$ \leq u_\$$ **then**
      $\mathcal{I} \leftarrow \mathcal{I} \cup [l_\$, u_\$, l'_\$, u'_\$]$
  $[l, u, l', u'] \leftarrow \texttt{updateFwdBwd}([l, u, l', u'], X[i], \mathcal{R})$
  $i \leftarrow i-1$
return $\mathcal{I}$

Algorithm 6 extractIrreducible($\mathcal{I}$)

if $\mathcal{I} = \emptyset$ **then**
  return $\emptyset$
$\mathcal{L} \leftarrow \emptyset$
for all $[l, u, l', u'] \in \mathcal{I}$ **do**
  $[l'_\$, u'_\$, l_\$, u_\$] \leftarrow \texttt{updateFwdBwd}([l', u', l, u], \$, \mathcal{R}')$
  if $l_\$ \leq u_\$$ **then**
    $\mathcal{L} \leftarrow \mathcal{L} \cup [l_\$, u_\$]$
if $\mathcal{L} \neq \emptyset$ **then**
  return $\mathcal{L}$
for all $a \in \Sigma$ **do**
  $\mathcal{I}_a \leftarrow \emptyset$
  for all $[l, u, l', u'] \in \mathcal{I}$ **do**
    $[l'_a, u'_a, l_a, u_a] \leftarrow \texttt{updateFwdBwd}([l', u', l, u], a, \mathcal{R}')$
    if $l_a \leq u_a$ **then**
      $\mathcal{I}_a \leftarrow \mathcal{I}_a \cup [l_a, u_a, l'_a, u'_a]$
  $\mathcal{L} \leftarrow \mathcal{L} \cup \texttt{extractIrreducible}(\mathcal{I}_a)$
return $\mathcal{L}$

the first right terminal extension found is not that of an irreducible edge but of the contained read. It is straightforward to handle this case by observing that such a read will have an overlap that is strictly shorter than that of the irreducible edge. In other words, the only acceptable right terminal extension is to the reads in $\mathcal{I}$ that have the longest overlap with X.

We can similarly modify `extractIrreducible` to handle overlaps for reads from opposite strands. To do this, we use `findIntervals` to determine the intervals for overlaps for the same strand as X and overlaps from the opposite strand of X (using the complement of X as in the previous section). When extending an interval that was found by the complement of X, we extend it by the complement of a. In other words if we are extending same-strand intervals by A, we extend opposite strand intervals by T and so on.

We now offer a sketch of the complexity of the irreducible overlap algorithm in the case where all edges in the graph are part of the walk spelling the genome sequence G. Let L_i be the label of irreducible edge i. During the construction of L_i at most k_i intervals must be updated, corresponding to the number of reads that have an edge-label containing L_i. The sum over all irreducible edges, $E = \sum_i (|L_i|k_i)$, is the total number of interval updates performed by `extractIrreducible`. Note that each read in $\mathcal{R}$ is represented by a path through the string graph. The total number of times edge i is used in the set of paths spelling all the reads in $\mathcal{R}$ is k_i and the amount of sequence in $\mathcal{R}$ contributed by edge i is $|L_i|k_i$. This implies E can be no larger than N, the total amount of sequence in $\mathcal{R}$, and `extractIrreducible` is $O(N)$. As `findIntervals` is also $O(N)$, the entire irreducible overlap detection algorithm is $O(N)$.

2.5.4 Results

The algorithms described in this chapter form the basis of the assembler I developed, SGA[1]. As a proof of concept, I profiled these algorithms on simulated error-free sequence reads. The assembly is broken into three stages: index, overlap and assemble. The index stage constructs the FM-index for a set of sequence reads, the overlap stage computes the set of overlaps between the reads and

[1]String Graph Assembler

the assemble stage loads the graph, performs transitive reduction if necessary, then compacts unambiguous paths in the graph and writes them out as a set of contigs. I performed two sets of simulations. In all simulations the faster Bauer-Cox-Rosone algorithm was used to calculate the FM-index. In both sets of simulations, I compared the exhaustive overlap algorithm (which constructs the set of all overlaps) and the direct construction algorithm (which only outputs overlaps for irreducible edges). First, I simulated *E. coli* reads with average sequence depth from 5X to 100X to investigate the computational complexity of the overlap algorithms as a function of sequence depth. After constructing the index for each data set, I ran the overlap step in exhaustive and direct mode with fixed $\tau = 27$. The running times of these simulations are shown in figure 2.2. As expected, the direct overlap algorithm scales linearly with sequence depth. The exhaustive overlap algorithm exhibits the expected above-linear scaling as the number of overlaps for a given read grows quadratically with sequence depth.

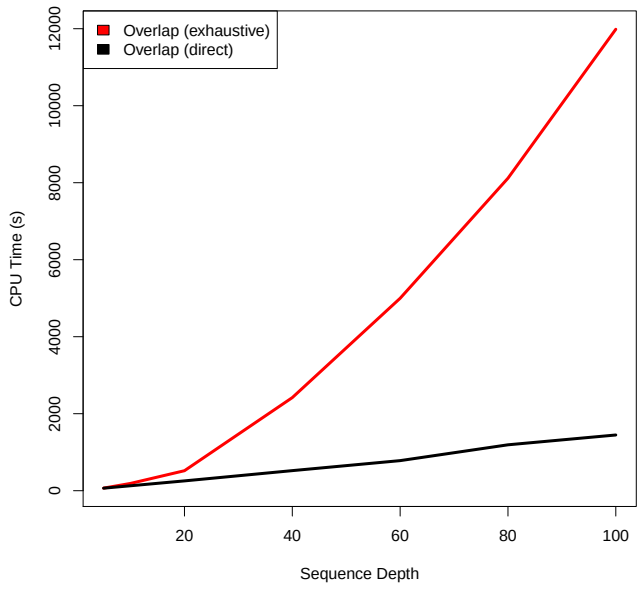


Figure 2.2: The running time of the direct and exhaustive overlap algorithms for simulated E. coli data with sequence depth from 5X to 100X.

I also simulated data from human chromosomes 22, 15, 7 and 2 to assess how the algorithms scale with the size of the genome. I pre-processed the chromosome sequences to remove sequence gaps then generated 100bp error-free reads randomly at an average coverage of 20X for each chromosome. Again I compared the direct construction algorithm to the exhaustive construction algorithm. The overlap length was set to 45. The results of these simulations are summarized in table 2.1.

	chr 22	chr 15	chr 7	chr 2	ratio
Chr. size (bp)	34.9M	81.7M	155.4M	238.2M	6.8
Number of reads	7.0M	16.3M	31.1M	47.6M	6.8
Duplicated reads	684k	1,663k	3,103k	4,709k	6.9
Duplicated %	9.8%	10.2%	10.0%	9.9%	-
Transitive edges	70.0M	176.4M	364.2M	583.8M	8.3
Irreducible edges	7.2M	17.2M	36.2M	57.4M	8.0
Assembly N50 (bp)	3.0k	4.1k	4.3k	4.8k	-
Longest contig (bp)	41.4k	51.9k	63.2k	57.9k	-
Index time	1,486s	3,652s	7,284s	11,443s	7.7
Overlap time (e)	3,595s	9,393s	27,736s	30,176s	8.4
Overlap time (d)	2,204s	7,885s	11,516s	17,596s	8.0
Assemble time (e)	1,399s	4,795s	15,287s	33,140s	23.7
Assemble time (d)	280s	694s	1,518s	2,432s	8.7
Index memory	1.0GB	2.2GB	4.2GB	6.5GB	6.5
Overlap mem. (e)	0.5GB	1.2GB	2.3GB	3.5GB	7.0
Overlap mem. (d)	0.5GB	1.2GB	2.3GB	3.5GB	7.0
Assemble mem. (e)	8.9GB	24.5GB	27.2GB	99.7GB	11.2
Assemble mem. (d)	2.1GB	5.0GB	10.0GB	15.7GB	7.5

Table 2.1: Simulation results for human chromosomes 22, 15, 7 and 2. For the overlap and assemble rows, (e) and (d) indicate the exhaustive and direct algorithms, respectively. The last column is the ratio between chromosome 2 and 22.

For all chromosomes the direct overlap computation algorithm was faster. The direct overlap calculation step required almost half the run time when com-

pared to the exhaustive overlap calculation. When the full overlap graph was constructed (exhaustive case) the assemble step required performing Myers' transitive reduction algorithm on the graph. This step was over 23 times longer for chromosome 2 than chromosome 22, as the chromosome 22 graph contained over 500 million transitive edges that needed to be removed. The vast number of transitive edges in this case caused the chromosome 2 graph to require nearly 100GB of memory. The assemble step for the direct case was over 13 times faster on chromosome 2 as the initial graph was much smaller and transitive reduction did not need to be performed. These results verify the efficiency of my direct string graph construction algorithm and the benefits when compared to building the full overlap graph first.

The memory bottleneck in these assemblies is loading the string graph into memory during the assemble step. This bottleneck will be addressed in the next chapter, where I describe an algorithm to find and compress *unipaths* in the graph without the need to load the entire string graph in memory.

2.6 Representing a de Bruijn Graph using the FM-Index

Recall from section 2.3.2 that the set of distinct k-mers of $\mathcal{R}$ defines the vertices of its de Bruijn graph. Likewise, the set of distinct ρ-mers defines the edges of the graph. This observation allows us to use the FM-index as a representation of the de Bruijn graph of $\mathcal{R}$. Here we describe queries to compute the local structure of the graph around a single vertex. To test whether a given k-mer is a vertex in the graph, we can use algorithm `isDBGVertex`. This performs a simple backwards search to check whether the k-mer, or its reverse-complement, has a non-empty suffix array interval.

Algorithm 7 isDBGVertex(K, $\mathcal{R}$)

$[l_1, u_1] \leftarrow$ `backwardsSearch`$(K, \mathcal{R})$
$[l_2, u_2] \leftarrow$ `backwardsSearch`$(\overline{K}, \mathcal{R})$
if $l_1 \leq u_1$ or $l_2 \leq u_2$ **then**
  return true
else
  return false

We can also use the FM-Index to get the neighbors of a particular vertex in the graph. For a k-mer K, there are 8 possible neighbors. To find which are actually present in $\mathcal{R}$, we can simply directly query for the 16 possible ρ-mers describing these neighbors (including their reverse complements). Pseudocode for this algorithm is shown in `getDBGNeighborsSingleIndex`. This requires $16(k+1)$ interval updates.

If the FM-index of both $\mathcal{R}$ and $\mathcal{R}'$ is available, we can implement a faster procedure based on performing extension queries like those described in our string graph construction algorithm in section 2.5.3. We start by calculating the interval pair for K, then using $\mathbf{Occ}_{\mathcal{R}}$ to calculate the possible left-extensions of K. This query provides the ρ-mers of the form xK which define prefix neighbors of K. To calculate suffix neighbors of K (of the form Kx), we can use right-extension queries. These queries must also be performed with the reverse complement of K, to cover both strands. In total, this procedure requires $4K$ interval updates plus 8 accesses of the occurrence array. As we need the FM-index of $\mathcal{R}$ and $\mathcal{R}'$ to do the right-extension queries, the memory usage is doubled when compared to `getDBGNeighborsSingleIndex`.

The description within this section uses Pevzner's ρ-mer based formulation of the de Bruijn graph. In our implementation of these algorithms we use a slight modification. Instead of querying for ρ-mers, we directly query for the neighboring k-mers (of the form $xK[1, k-1]$ and $K[2, k]x$ for $x \in \{a, c, g, t\}$). This is subtly different as connected vertices do not necessarily need to share a ρ-mer - they only need to overlap by $k-1$ bases.

Algorithm 8 getDBGNeighborsSingleIndex(K, $\mathcal{R}$)

$k \leftarrow |K|$
for all $Q \in \{aK, cK, gK, tK\}$ **do**
  $[l, u] \leftarrow$ `backwardsSearch`$(Q, \mathcal{R})$
  $[l', u'] \leftarrow$ `backwardsSearch`$(\overline{Q}, \mathcal{R})$
  if $l \leq u$ or $l' \leq u'$ **then**
    $\mathcal{I} \leftarrow \mathcal{I} \cup \{Q[1, k]\}$
for all $Q \in \{Ka, Kc, Kg, Kt\}$ **do**
  $[l, u] \leftarrow$ `backwardsSearch`$(Q, \mathcal{R})$
  $[l', u'] \leftarrow$ `backwardsSearch`$(\overline{Q}, \mathcal{R})$
  if $l \leq u$ or $l' \leq u'$ **then**
    $\mathcal{I} \leftarrow \mathcal{I} \cup \{Q[2, k+1]\}$
return $\mathcal{I}$

Chapter 3

The SGA Assembler

3.1 Introduction

In the previous chapter, I described an algorithm to construct an assembly string graph Myers [2005] for a set of error-free sequence reads using the FM-index. In this chapter, I expand upon the algorithms to build a fully-functional sequence assembly program. I will describe algorithms to correct base calling errors, remove duplicate and contained sequences and construct contigs and scaffolds for real sequencing data. These algorithms are implemented in my software called SGA (String Graph Assembler)[1]. SGA is implemented as a modular pipeline, which allows it to be easily extended as improved algorithms are developed or sequencing technology changes.

3.1.1 Publication Note

The work described in this chapter was previously published in [Simpson and Durbin, 2012]. Sections 3.2.7, 3.3.1 and 3.3.5 describe currently unpublished results. The work described is the sole work of the author, under the supervision of his PhD supervisor, Richard Durbin.

[1] Source code available at www.github.com/jts/sga

3.1.2 Algorithm Overview

The SGA algorithm is based on performing queries over an FM-index constructed from a set of sequence reads. The SGA pipeline begins by preprocessing the sequence reads to filter or trim reads with multiple low-quality or ambiguous base calls. The FM-index is constructed from the filtered set of reads and base-calling substitution errors are detected and corrected using k-mer frequencies. The corrected reads are re-indexed then duplicated and contained sequences are removed, remaining low-quality sequences are filtered out and a string graph is built. Contigs are assembled from the string graph and constructed into scaffolds if paired end or mate pair data is available. Figure 3.1 depicts the flow of data through the SGA pipeline. I discuss the major components of SGA below.

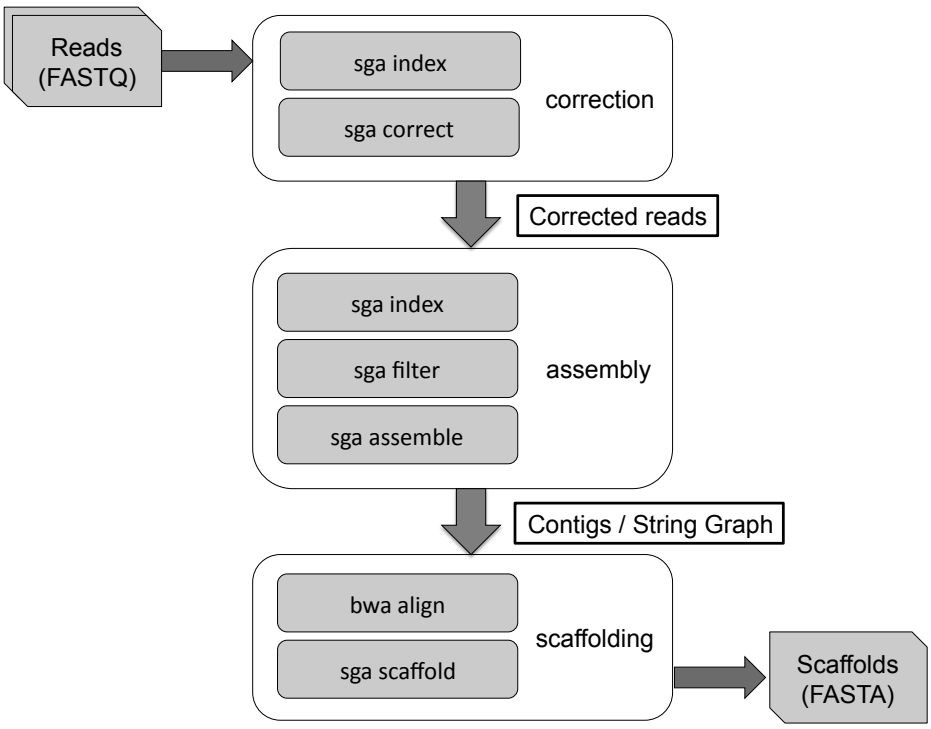


Figure 3.1: Schematic of the flow of data through SGA.

3.2 SGA Algorithms

3.2.1 Construction of the FM-index for large read sets

The algorithm begins with the construction of the FM-index for the complete set of reads. In Chapter 2, I described a modified version of the Nong-Zhang-Chan algorithm [Nong et al., 2009]. This algorithm has the drawback that to compute the Burrows-Wheeler transform of a set of reads, $\mathcal{R}$, it must first construct the full suffix array for a read set. The full suffix array requires $N \log(N)$ bits of memory, where N is the total number of bases in the read set. For a human genome sequenced to 30X coverage, this would require over 400GB of memory. As using this amount of memory during index construction would eliminate any benefit of using a compressed data structure for assembly, I have taken a different approach. I have implemented a distributed construction algorithm that builds an FM-index for subsets of $\mathcal{R}$, $\mathcal{R}_1$, $\mathcal{R}_2$,..,$\mathcal{R}_m$. I then iteratively merge pairs of the intermediate indices together using a BWT merging algorithm [Ferragina et al., 2010] until a single index of the entire data set is obtained. As the space occupancy of the FM-index is typically less than an order of magnitude smaller than that of a suffix array, this indexing strategy allows us to efficiently build the FM-index for very large sequence collections. This construction strategy can be easily parallelized as the construction of the FM-index for each read subset, and most merging operations, can be computed independently.

Recently Bauer, Cox and Rosone designed an algorithm specifically tailored to the problem of computing the BWT for a very large collection of short ($\approx$ 100bp) sequence reads [Bauer et al., 2011]. Their algorithm directly computes the BWT of a read set without the need to first construct a suffix array. Their algorithm has two variants. Let n be the number of reads and l be the read length. The first variant, named `BCR`, uses $O(n \log(nl))$ bits of working space, and $O(l\texttt{sort}(n))$ time, where `sort(n)` is the time required to sort n integers. The second variant, `BCRext`, uses external memory (disk storage) for most data structures. It requires $O(ln)$ time with constant memory usage and overall I/O volume $O(l^2 n)$. Both algorithms work by progressively building partial BWTs, starting from the last base of each read. At each iteration j, the position to insert the next base of

each read into the partial BWT can be calculated from the previous iteration, $j-1$. In both variants of the algorithm the partial BWTs are stored on disk to save memory. As `BCR` and `BCRext` use disk-based storage to store intermediate files, their performance is dependent on the I/O and seek times of the underlying harware[1]. For this reason, I implemented a variant of `BCR` within SGA that uses in-memory storage of the partial BWT files. I compare the performance of the indexing algorithms in section 3.3.1. All algorithms can be used by SGA - the Nong-Zhang-Chan and in-memory BCR algorithms are natively implemented in SGA. `BCR` and `BCRext` are available by running the authors' reference implementation (BEETL[2]) then running a script to convert the output into SGA's BWT file format.

3.2.2 k-mer based error correction algorithm

Real sequencing data contains base calling errors. SGA's error corrector is currently designed to handle substitution errors, which are the dominant error mode in the Illumina sequencing platform [Bentley et al., 2008]. I have implemented two error correction methods. The first is a k-mer frequency-based corrector, which has been successfully used in other sequence assemblers [Kelley et al., 2010; Li et al., 2010c; Pevzner et al., 2001]. The second algorithm is based on finding inexact overlaps between sequence reads. In my tests the k-mer based corrector is faster than the overlap-based corrector and is therefore the default method of correction using SGA. Both correction methods have an option to use per-base quality scores of the read being corrected to vary the coverage threshold required to support a base call.

The primary error correction algorithm in SGA is based on k-mer frequencies. Assuming base-calling errors are a random process that occur independently, k-mers covering a incorrectly-called base in a read will occur in the entire data set with low frequency (typically k-mers covering an error will form unique strings). This is illustrated in figure 3.2, which plots a histogram of k-mer frequencies for simulated error-free data and simulated data with 1% error rate. For both data

[1] The authors of BCRext found that Solid State Drives offered the fastest indexing performance

[2] www.github.com/BEETL/BEETL

sets, 30X coverage of 100bp reads from the *E. coli* genome was generated. In the perfect data, there are almost no k-mers in the read set that are seen less than 5 times. In contrast, for the 1% error rate data, there is a large proportion of k-mers are low frequency (<5 occurrences). These are k-mers that are very likely to contain sequencing errors. We can therefore use k-mer occurrence counts to distinguish between correct and erroneous k-mer strings.

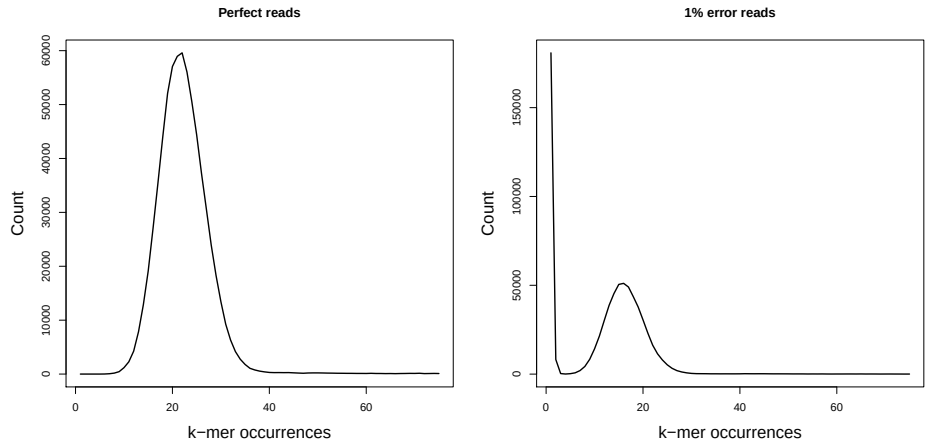


Figure 3.2: k-mer occurrence histogram for simulated perfect data (left) and simulated data with 1% uniform base calling errors (right). The y-axis records the number of times a k-mer with frequency x occurs in samples of the data set. For example, there are 57,059 k-mers seen 20 times in the perfect data set. The histogram was calculated by sampling 10,000 random reads.

Our correction algorithm follows from other k-mer based correctors [Kelley et al., 2010; Li et al., 2010c; Pevzner et al., 2001] in that it attempts to identify positions in the read that are incorrect, then searches for a suitable correction. The algorithm scans each read left-to-right to identify bases that are not present in a k-mer of frequency at least c. We iterate over the potentially incorrect bases and change the base in the left-most k-mer covering the position to the 3 other possibilities. If exactly one of the possibilities yields a k-mer with frequency at least c, the correction is made. If no correction can be found using the left-most covering k-mer, the right-most covering k-mer is tested. If this test also fails the procedure terminates and returns the original read sequence. If a set

of corrections is found that makes all bases in the read trusted (frequency $\geq c$), then the procedure terminates and returns the modified sequence.

The minimum coverage parameter c is conservatively chosen to avoid collapsing SNPs (if the genome is diploid) or distinct copies of a repeat. This parameter can either be manually provided or automatically selected by SGA by finding the trough in the k-mer frequency histogram.

Unlike previous k-mer based error correctors, the k-mer frequencies are not stored in a lookup or hash-table but rather directly calculated from the FM-index. Each k-mer frequency lookup in the FM-index only requires $O(k)$ time when using the algorithm `countOccurrences`.

Algorithm 9 countOccurrences($\mathcal{R}$, Q) - count the number of times Q and its reverse complement occurs in $\mathcal{R}$

$c \leftarrow 0$
$[l, u] \leftarrow$ `backwardsSearch`$(\mathcal{R}, Q)$
if $l \leq u$ **then**
  $c \leftarrow u - l + 1$
$[l, u] \leftarrow$ `backwardsSearch`$(\mathcal{R}, \overline{Q})$
if $l \leq u$ **then**
  $c \leftarrow c + u - l + 1$
return c

In addition to being computationally efficient, this has the advantage of using comparatively little memory and allowing greater flexibility in the parameter choices as the FM-index can support any value of k, unlike a hash table which must be reconstructed for each choice of k.

3.2.3 Overlap based error correction

The second error correction algorithm in SGA is based on finding inexact overlaps between reads. In the next section, I describe the algorithm to compute overlaps. In the following section, I describe how these overlaps are used to correct reads.

3.2.3.1 Finding Inexact Overlaps with the FM-Index

The overlap algorithm from 2.5.2 can be extended to allow mismatches in the overlaps. Let ϵ be the maximum allowed mismatch rate between two overlapping strings (for example $\epsilon = 0.05$ would allow 1 mismatch in a 20bp overlap). At each stage of the overlap extension, we can branch to each possible base A, C, G, T. If the base that we extend to is different from the current position in X, we increment a mismatch counter d. If the value of d exceeds the maximum number of mismatches for an overlap of length $|X|-1$ the current search path is terminated as a valid overlap cannot possibly be found. When an overlap of length at least τ is found and the mismatch rate is at most ϵ we output overlaps as in `findOverlaps`. We then recursively branch the search, updating the mismatch counter as needed. The pseudocode for this algorithm is presented below in `findOverlapsInexact` and `findOverlapsInexactExtend`. While this matching algorithm will return the complete set of τ, ϵ-overlaps, it is inefficient. This naive search will branch excessively at the beginning of the search (when i is close to $|X|$) as the overlap lengths are not large enough to exclude strings that are matched by chance. Once the overlap length becomes long enough (i.e. for $|X| - i \approx 16$) then most branches will not form valid matches (and hence have empty suffix array intervals) and therefore stop the recursion.

Algorithm 10 findOverlapsInexact(X, $\mathcal{R}$, τ, ϵ) - find all reads overlapping X by at least τ bases with error rate at most ϵ

```
i ← |X|
for all b ∈ [A, C, G, T] do
    l ← C_R(b)
    u ← C_R(b + 1) − 1
    if X[i] ≠ b then
        findOverlapsInexactExtend(X, i, 1, [l, u], R, τ, ε)
    else
        findOverlapsInexactExtend(X, i, 0, [l, u], R, τ, ε)
```

Algorithm 11 findOverlapsInexactExtend(X, i, d, $[l, u]$, $\mathcal{R}$, τ, ϵ) - perform one round of extension of the inexact search. The current suffix array interval is given by l and u which corresponds to a string from the end of X to base i with d mismatches.

```
// check if the number of mismatches exceeds the maximum
// possible for a valid overlap
m ← ⌊(|X| − 1) ∗ ε⌋
if d > m then
  return
// check if overlaps should be output
o ← |X| − i + 1
r ← d/o
if o ≥ τ and r ≤ ε then
  [l$, u$] ← updateBackwards(R, [l, u], $)
  if l$ ≤ u$ then
    outputOverlaps(X, [l$, u$])
// perform branched extension
if i > 1 then
  i ← i − 1
  for all b ∈ [A, C, G, T] do
    [l′, u′] ← updateBackward(R, [l, u], b, R)
    if l′ ≤ u′ then
      if X[i] ≠ b then
        findOverlapsInexactExtend(X, i, d + 1, [l′, u′], R, τ, ε)
      else
        findOverlapsInexactExtend(X, i, d, [l′, u′], R, τ, ε)
```

We can design a more efficient algorithm using the seed-and-extend method of sequence alignment. This method is based on the principle that if we want to align a string X to a text T with up to d mismatches, we can create $d + 1$ *seeds* over the sequence of X, one of which must be matched exactly to T. The seed matches can then be extended allowing for mismatches. We have adapted this method of alignment to finding inexact overlaps with the FM-index. We must

take care when choosing the seed length to ensure that at least one seed matches exactly between any two reads that have a τ, ϵ-overlap. Let $d_\tau = \lfloor \epsilon\tau \rfloor$ be the maximum number of differences between two reads that overlap by the minimum amount. We define the minimal seed region of the read as $r_{min} = \lceil d_\tau/\epsilon \rceil$ and calculate the seed length $l_{seed} = \lfloor r_{min}/(d_\tau + 1) \rfloor$. This value of l_{seed} is small enough such that for all overlap lengths $i \geq \tau$ we are guaranteed to have $\lfloor i\epsilon \rfloor + 1$ seeds covering the suffix of X of length i and hence we can find all τ, ϵ-overlaps.

Once the seed positions of X have been calculated, we can find the suffix array intervals for the seeds using an exact match with the FM-index. The seeds can then be extended with a branching algorithm similar to that of `findOverlapsInexact`. We note that in this case, the extensions are not a strict right-to-left search as in `findOverlapsInexact` as some seeds start in the middle of the read. We use the bidirectional search procedure outlined in Chapter 2 to extend the seeds both left-to-right and right-to-left. See also [Välimäki et al., 2010] for a discussion of other inexact prefix-suffix matching algorithms.

3.2.3.2 Overlap Based Error Correction Algorithm

Let X be a read in $\mathcal{R}$ that we want to correct. We use the seed-and-extend algorithm presented in the previous section to find all reads in $\mathcal{R}$ with a τ, ϵ-overlap with X. This set of reads forms a multiple alignment with respect to X. Let $C[i]$ be a 4 element vector of the counts for each base call in column i of the multiple alignment. A simple consensus-based correction algorithm would be to set $X[i]$ to the element of $C[i]$ with the highest value. However, we must take care to avoid collapsing variation (if the sequenced genome is diploid) or distinct copies of a repeat. We filter the multiple alignment by excluding reads that have consistent mismatches with respect to X. If two elements of $C[i]$ have a count of at least v, we label position i as *conflicted*. The reads that match X at all conflicted columns are kept and the remainder excluded; $C[i]$ is re-calculated from the filtered multiple alignment. We correct $X[i]$ to be the consensus base indicated by $C[i]$ if there is a single base in $C[i]$ that occurs more than c times. This condition helps avoid setting $X[i]$ to an incorrect base in the situation that multiple well-supported bases remain in the multiple alignment. The values of

v (the conflict threshold) and c (the minimum base call support required) are command line parameters (the default values are 5 and 3, respectively).

As the number of overlaps for a given read is dependent on sequence depth, the runtime of the overlap based algorithm is dependent on the depth of sequencing. The run time of k-mer based algorithm presented in the previous section does not depend on the sequence depth. For this reason, it tends to be much faster than the overlap-based algorithm presented here, and therefore the k-mer based algorithm is the default method of correction used in SGA.

3.2.4 Read filtering

To construct the string graph we require a subset of $\mathcal{R}$ consisting of unique reads. We achieve this by removing contained and duplicated reads. To compute this subset, we use the FM-index to calculate full-length matches for each read in $\mathcal{R}$. If a read $\mathcal{R}_i$ has a full length match (including reverse complements) to some other read $\mathcal{R}_j$ we keep $\mathcal{R}_i$ iff $i < j$, otherwise $\mathcal{R}_i$ is discarded. Once the unique subset $\mathcal{U}$ of $\mathcal{R}$ has been calculated, we do not need to re-compute the FM-index of $\mathcal{U}$ from scratch. The BWT of $\mathcal{U}$ can be derived from the FM-index of $\mathcal{R}$ by marking the positions in $\mathbf{B}_{\mathcal{R}}$ that correspond to reads that were discarded and exporting only the unmarked positions as $\mathbf{B}_{\mathcal{U}}$ [Sirén, 2009].

Some reads remain uncorrected after error correction. To prevent these sequences from impacting the assembly, we remove sequences with unique k-mers. By default, this filter requires all 27-mers in a read to be seen at least twice.

3.2.5 Read merging and assembly algorithm

After correction and filtering, the vast majority of the remaining reads do not contain errors. We could directly apply our string graph construction algorithm (section 2.5.3) to these, however the resulting graph would have a vertex for every read and therefore require a substantial amount of memory when assembling very large genomes (as demonstrated in table 2.1). The majority of reads in the initial graph are simply connected (that is, without branching) to two other reads - one matching a prefix of the read and one matching a suffix. Such chains of reads, referred to as *unipaths*, can be unambiguously merged to reduce the size of the

graph. We have developed an algorithm to merge unipaths by locally constructing the assembly graph around each read. For each read, we find the predecessor and successor vertices in the graph by querying the FM-index for its irreducible edge set using `findIntervals` and `extractIrreducible` from section 2.5.3. If the read connects to its neighbors without branching, we continue the search from the neighboring reads. This search stops when a branch in the graph, or no possible extension, is found. This procedure will discover all non-branching chains in the graph and allow the chain to be replaced by a single merged sequence. As the predecessor/successor queries only require the FM-index, not the complete structure of the graph, this merging step requires comparatively little memory. Once we have performed this merging step, we build an FM-index for the merged sequences and use this FM-index to construct the full string graph. We then perform the standard assembly graph post-processing step of removing tips (see section 1.2.2.1) from the graph where a vertex only has a connection in one direction [Chaisson and Pevzner, 2008; Li et al., 2010c; Simpson et al., 2009; Zerbino and Birney, 2008].

To account for heterozygosity in a diploid genome, we have developed an algorithm to find and catalog variation described by the structure of the graph, similar to the "bubble-popping" approaches taken by de Bruijn graph assemblers. Let v be a vertex in the graph which branches (the prefix or suffix of v has multiple overlaps). Following each branch, we search outwards from v for a set of walks, $\mathcal{W}$, which meets the following conditions: 1) all walks terminate at a common vertex u and 2) no vertex visited in any walk between v and u has an edge to a vertex that is not present in a walk in $\mathcal{W}$. The first condition ensures that the walks describe equivalent sequence in G - any assembly of G that visits v and u must use one of the found walks. The second condition ensures that the induced subgraph of G described by the walks is self-contained - we can remove any walk in $\mathcal{W}$ without breaking any walk in $G \setminus W$. Once a set of walks meeting these conditions has been found, we select one of the walks to remain in the graph. We align the sequence described by the other walks to the sequence of the selected walk and, if the sequence similarity is within tolerance (by default 95%) in all cases, the non-selected walks are removed from the graph. We retain the sequences of the removed walks in a FASTA file to allow the heterozygous

variation present in the genome to be analyzed after assembly.

3.2.6 Paired end reads/Scaffolding

The final stage of the assembly is to build scaffolds from the contigs using paired-end or mate-pair data. Similar to other approaches to scaffolding [Pop et al., 2004], our method is based on constructing a graph of the relationships between contigs. We begin by re-aligning the paired reads to the contigs using `bwa` [Li and Durbin, 2009]. The copy number of each contig in the source genome is estimated from the number of reads aligned to the contig using Myers' A-statistic which approximates the log-odds ratio between the contig being unique and a collapsed repeat [Myers, 2005]. By default, we classify contigs with an A-statistic ≥ 20 as unique and the remainder as repetitive. We construct a scaffold graph where each unique contig is a vertex. Contigs linked with read pairs are connected by a bidirected edge labeled with the estimated gap size separating the contigs. Paths through this scaffold graph describe layouts of the contigs into scaffolds. The gap sizes are estimated using the `DistanceEst` subprogram from ABySS [Simpson et al., 2009].

Our scaffolder first removes ambiguous or likely erroneous edges from the graph. For each contig in the graph with more than one edge in a particular direction, we test whether the linked contigs have an ordering that is consistent with each pairwise distance estimate. An ordering of contigs $C_1, C_2, ..., C_n$ is called consistent if no pair of contigs has an overlap (implied by their positions in the layout) greater than α bases ($\alpha = 400$ by default). If the contigs cannot be consistently ordered, we break the graph by removing all edges of the affected contigs.

Once the graph has been cleaned of inconsistent edges, we find and isolate any directed cycles then compute the connected components of the graph. For each connected component, we find the terminal vertices of the component (vertices that have an edge in only one direction) and find all paths between each pair of terminal vertices. The path containing the largest amount of sequence is retained as the primary layout of the scaffold. The SGA scaffolder supports multiple libraries of different sizes.

The scaffolds are represented as an alternating list of contigs and gaps, $C_1, g_1, C_2, g_2, ..., C_n$. We attempt to fill in the gaps through a three-stage process. Let C_i and C_j be two adjacent contigs separated by a distance of g_i. As C_i and C_j are vertices in the string graph we previously constructed, we search the string graph for a walk connecting these vertices with the constraint that the total walk length can be no larger than $|C_i| + |C_j| + g_i + \theta_i$ where θ_i allows for the inexact distance estimate (by default 3 times the standard error of the distance estimate). If a single walk is found to meet this constraint, we replace C_i, g_i, C_j in the scaffold by the walk string. If no walk can be found connecting the vertices and g_i is negative (the contigs are predicted to overlap), we align the ends of C_i and C_j. If the predicted overlap is confirmed to exist, the sequences of C_i and C_j are merged. If the gap cannot be resolved, we simply fill the sequence between C_i and C_j with g_i ambiguity ("N") symbols.

As described in section 2.6, we can use the FM-index as a representation of the de Bruijn graph for all k. In the latest version of SGA, we can use this feature to optionally fill in scaffold gaps by finding walks through a de Bruijn graph. Let C_i, g_i, C_j be two contigs in a scaffold separated by a gap. Starting at $k = 91$, we use the last k-mer of C_i to seed a breadth-first search through the 91-mer de Bruijn graph. If a path through the graph ending at the first k-mer of C_j can be found, and the length of the path is within 100bp of the estimated size of the gap, the gap is replaced by the string corresponding to the path. To avoid searching very dense regions of the graph, the breadth-first search aborts if more than 2000 vertices have been visited, if the search branches more than 50 times or if more than 20 branches are being simultaneously following. If the gap cannot be successfully filled, k is decreased by 10 and the procedure restarts. This continues until the gap is filled or k is less than 51. The starting and stopping k are parameters to the program. There also exists an option to ignore k-mers that are seen less than t times in the reads ($t = 3$ by default). This method is typically able to fill 10-20% of the scaffold gaps, leading to slightly increased contig N50. This de Bruijn graph gap-filling procedure is a standalone component of SGA - it can be used on scaffolds from any assembler.

3.2.7 Implementation Details

3.2.7.1 FM-Index Implementation

SGA relies on pattern searches over the FM-index for both error correction and the construction of the string graph. While the FM-index provides optimal $O(|P|)$ queries for a pattern P, the implementation of the data structure has a significant impact on the performance of these queries. In this section I describe the implementation of the FM-index within SGA.

As described in section 2.4, the FM-index of a string X over an alphabet Σ consists of three data structures:

- $\mathbf{B}_X$ - the Burrows-Wheeler transform of X
- $\mathbf{Occ}_X(a, i)$ - the number of occurrences of the symbol a in $\mathbf{B}_X[1, i]$.
- $\mathbf{C}_X(a)$ - the number of symbols in X that are lexicographically lower than the symbol a

$\mathbf{C}_X$ only requires storing $|\Sigma|$ integers. If we explicitly stored $\mathbf{Occ}_X(a, i)$ for all $i \in \{1, |X|\}$ the amount of memory required would be $|X||\Sigma| \log_2(|X|)$. In our case with an alphabet of size 5, $|\Sigma| \log_2(|X|) \approx 40$ bytes. This huge memory cost would offset any benefit of using the FM-index. A common method to reduce the memory usage is to only store $\mathbf{Occ}_X(a, i)$ for i which is a multiple of a fixed value d. When a value $\mathbf{Occ}_X(a, j)$ is requested that is not explicitly stored, the closest stored value to j, $\mathbf{Occ}_X(a, k)$ is looked up and the requested value is calculated by explicitly counting the symbols in $\mathbf{B}_X$ between j and k. This allows the memory usage to be reduced to $40|X|/d$ bytes. The value of d offers a tradeoff between space and time. Larger values of d will use less memory but require longer stretches of $\mathbf{B}_X$ to be traversed during counting. In SGA, we use a two-tier encoding of the occurrence array similar to the encoding used in [Healy et al., 2003][1]. We store the absolute number of times each symbol have been seen in $\mathbf{B}_X[1, i]$ using an 8 byte integer for all i divisible by 8192. We call these absolute counts *large markers*. For all i divisible by d, we store a two-byte integer *small marker* which is the symbol count relative to the previous large marker.

[1] This implementation of the occurrence array was suggested to us by Travis Wheeler

This encoding requires an extra addition when calculating a value of $\mathbf{Occ}_X$, but lowers the memory usage to $40|X|/8192 + 10|X|/d$ bytes. The d parameter is chosen by the user at runtime and defaults to 128.

A naive encoding of $\mathbf{B}_X$ would require $|X| \log_2 |\Sigma|$ bits. However, as the Burrows-Wheeler transform sorts substrings of X, $\mathbf{B}_X$ contains long runs of repeated symbols. To account for this, we use run length encoding for $\mathbf{B}_X$. Each run is a byte encoding a <symbol, length> pair. We use 3 bits for the symbol and 5 bits for the length of the run. This encoding scheme is efficient for high-coverage data and requires ≈ 1.3 bits per base on average for high-depth data. However, when the run lengths are short due to lack of coverage or sequencing errors, this encoding scheme is inefficient. During the development of SGA I experimented with different methods of encoding $\mathbf{B}_X$, including Huffman and Golomb coding. Each of these encodings offered greater space efficiency than the <symbol, length> pair encoding, however they were all slower to decode during the critical $\mathbf{Occ}_X$ counting procedure. This issue remains to be further investigated as significant space savings could be made in SGA. Ideally, the FM-index would be implemented as a separate software library, allowing the time/space tradeoff to be selectable by the user.

3.2.7.2 Progam Design, Implementation and Libraries

SGA is implemented in C++. It uses zlib (`www.zlib.net`) to read compressed files and BamTools [Barnett et al., 2011] to read SAM/BAM files. It is multi-threaded using pthreads and the hoard parallel memory allocator [Berger et al., 2000]. The source code is licensed under GPLv3 and freely available online `www.github.com/jts/sga`.

SGA is designed to be modular, so that components of the assembler can be replaced as improved algorithms are available. The major components of SGA are listed below.

- `sga index` - constructs the FM-index for a set of sequence reads.
- `sga merge` - merges two indices together into a single index.

- `sga correct` - performs the error correction routines described in section 3.2.2 and 3.2.3.
- `sga filter` - removes duplicate reads and reads that contain low-frequency k-mers from the read set.
- `sga stats` - infers the error rate for a set of reads.
- `sga overlap` - constructs a string graph from the FM-index using the algorithm described in Simpson and Durbin [2010].
- `sga fm-merge` - detects and merges unipaths in the string graph.
- `sga assemble` - simplifies the string graph by removing sequence variation bubbles and outputs contigs.
- `sga scaffold` - reads in a scaffold graph and constructs linear scaffolds with gap size estimates.
- `sga scaffold2fasta` - attempts to fill in scaffold gaps and outputs the scaffolds in FASTA format.
- `sga gapfill` - standalone gapfiller based on de Bruijn graphs.

3.3 Results

In this section, I demonstrate the use of SGA on a variety of real data sets. I begin by benchmarking the indexing performance of the algorithms discussed in section 3.2.1. I then perform an assembly of a medium sized genome to compare the performance of SGA against widely used de Bruijn graph assemblers. I also use SGA to assemble a human genome, demonstrating the benefits of using a compressed data structure on memory usage. Finally, I describe the assembly of 104 *Schizosaccharomyces pombe* yeast strains.

Program	CPU time	Walltime	Max Memory
sga-sais	12581s	12567s	17.0 GB
sga-bcr	5393s	5367s	9.3 GB
BCR	9553s	18184s	1.1 GB
BCRext	4761s	7953s	0.05 GB

Table 3.1: Running time and memory usage for four different methods of constructing the BWT for 66.7 million 100bp reads.

3.3.1 Index construction results

I profiled the SGA implemention of Nong-Zhang-Chan (`sga-sais`), `BCR`, `BCRext` and the SGA implementation of in-memory `BCR` (`sga-bcr`) on 66.7 million 100bp reads from the *C. elegans* genome. In this test, I used version 0.9.20 of SGA and version 0.0.2 of the BCR authors' non-commerical reference implementation, named BEETL[1]. To conserve memory in `sga-sais`, the read set was broken into 4 subsets. `sga-sais` was used to calculate the BWT of each subset, then 2 BWT merging rounds were performed to compute the final result. For the other three algorithms the BWT was directly constructed from the full read set.

Each test was run on an Intel Xeon X5650 CPU (2.67GHz) with 38GB of available memory. The input, temporary and output files were stored on a Lustre parallel distributed file system. All programs were run with a single thread. The results are summarized in table 3.1. The wall-clock, cpu time and memory usage was all measured by our cluster computing environment, LSF (Load Sharing Facility).

The BCRext algorithm required negligable memory and was the fastest program when measured by CPU time. My in-memory implementation of BCR was the fastest program when measured by wall-clock time, however it required 9.3GB of memory. The Nong-Zhang-Chan algorithm implemented in SGA required the most CPU time and the most memory.

[1] `www.github.com/BEETL/BEETL`

3.3.2 *C. elegans* Assembly

To assess the performance of SGA I performed assemblies of the nematode *C. elegans* using SGA and three other assemblers. The Velvet assembler [Zerbino and Birney, 2008] was one of the first de Bruijn graph-based assemblers for short reads and has become a standard tool for assembling small to medium sized genomes. The ABySS assembler [Simpson et al., 2009] was developed to handle large genomes by distributing a de Bruijn graph across a cluster of computers. SOAPdenovo is also based on the de Bruijn graph and designed to assemble large genomes [Li et al., 2010b,c].

C. elegans provides a good real-world test case for assembly algorithms because it has a complete and accurate reference sequence [C. elegans Sequencing Consortium, 1998], it propagates as a hermaphrodite so the genome of an individual (or strain) is homozygous and essentially free of SNPs and structural variants, and the genome is a reasonable size for evaluation (100 Mbp). I downloaded *C. elegans* sequence reads (strain N2) from the NCBI SRA (accession SRX026594). The data set consists of 33.8M read pairs sequenced using the Illumina Genome Analyzer II. The mean DNA fragment size is 250 bp from which reads of length 100 bp were taken from both ends of the fragment. To reduce the impact of differences between the sequenced individual and the reference sequence, I called a new consensus sequence for the *C. elegans* reference genome (build WS222, www.wormbase.org) based on alignments of the reads to the reference using `samtools` as specified by the documentation[1].

As sequence assemblers are often sensitive to the input parameters, I performed multiple runs with each assembler. The de Bruijn graph assemblers were run for all odd k-mer sizes between 51 and 73 (inclusive). The k-mer size providing the largest scaffold N50 was selected for further analysis (67 for ABySS, 61 for Velvet, 59 for SOAPdenovo). Similarly, for SGA the k-mer size used for error correction and the minimum overlap parameter for assembly were selected to provide the largest scaffold N50 (k=41 for error correction, τ=75 for the minimum overlap). I also performed a SOAPdenovo assembly using their `GapCloser` program after scaffolding. `GapCloser` was able to fill in many gaps within scaffolds,

[1] The command run was `samtools mpileup -uf ref.fa aln.bam | bcftools view -cg -`

which increased the contig N50 and genome coverage. However, these increases came at the cost of substantially lowered accuracy. In the following analysis, I use the SOAPdenovo assembly without using GapCloser.

I broke the assembled scaffolds into their constituent contigs by splitting each scaffold whenever a run of "N" bases was found. I filtered the contig set by removing short contigs ($< 200bp$ in length). The remaining contigs were aligned to the consensus-corrected reference genome using bwa-sw [Li and Durbin, 2010] with default parameters. I considered a number of different assessment criteria, which are described below and summarized in table 3.2.

	SGA	Velvet	ABySS	SOAPdenovo
Scaffold N50 size	26.3 kbp	31.3 kbp	23.8 kbp	31.1 kbp
Aligned contig N50 size	16.8 kbp	13.6 kbp	18.4 kbp	16.0 kbp
Mean aligned contig size	4.9 kbp	5.3 kbp	6.0 kbp	5.6 kbp
Sum aligned contig size	96.8 Mbp	95.2 Mbp	98.3 Mbp	95.4 Mbp
Reference bases covered	96.2 Mbp	94.8 Mbp	95.9 Mbp	95.1 Mbp
Reference bases covered by contigs $\geq$ 1kb	93.0 Mbp	92.1 Mbp	93.9 Mbp	92.3 Mbp
Mismatch rate at all assembled bases	1 per 21,545 bp	1 per 8,786 bp	1 per 5,577 bp	1 per 26,585 bp
Mismatch rate at bases covered by all assemblies	1 per 82,573 bp	1 per 18,012 bp	1 per 8,209 bp	1 per 81,025 bp
Contigs with split/bad alignment (Sum size)	458 (4.4 Mbp)	787 (7.2 Mbp)	638 (9.1 Mbp)	483 (4.4 Mbp)
Total CPU time	41 hr	2 hr	5 hr	13 hr
Max Memory usage	4.5 GB	23.0 GB	14.1 GB	38.8 GB

Table 3.2: Assessment of various assembly programs on the C. elegans data set.

3.3.2.1 Substring coverage

For the first assessment, I sampled strings from the consensus sequence and tested whether they were exactly present in the contigs. I sampled 10,000 strings of length from 50 bp up to 5,000 bp. This assessment combines three measures; the contigs must be accurate (as exact matches are required), complete (as the string must be present in the contig) and contiguous (as strings broken between multiple

contigs will not be found). Figure 3.3 plots the proportion of strings found in the contigs as a function of the string length. All assemblers perform well for short strings (50 to 100 bp). For longer string lengths, SGA slightly outperforms the other three assemblers.

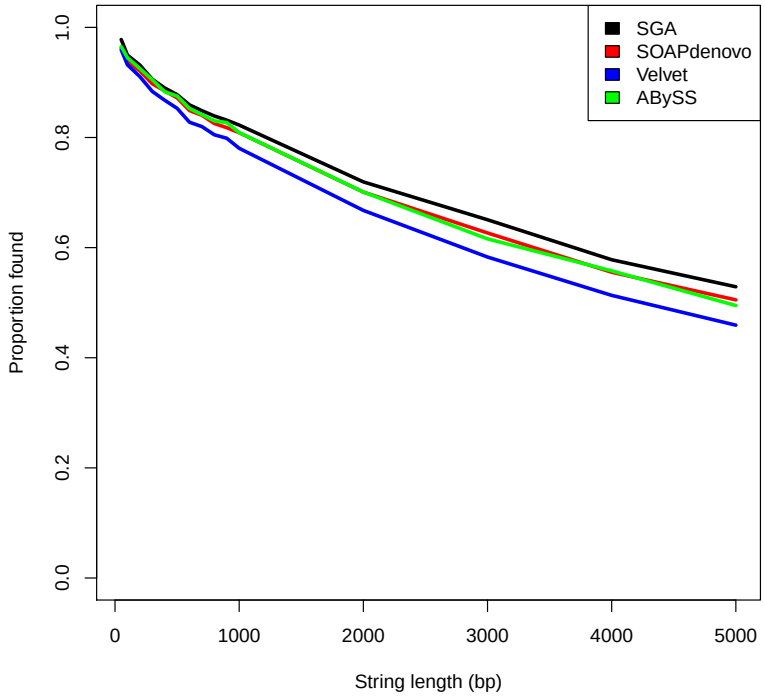


Figure 3.3: Reference string coverage analysis for the C. elegans N2 assembly. For string lengths from 50bp up to 5,000bp, 10,000 strings were sampled from the consensus-corrected C. elegans reference genome. The proportion of the strings found in the SGA, Velvet, ABySS and SOAPdenovo assemblies is plotted.

3.3.2.2 Assembly Contiguity

I assessed the contiguity of the assemblies by calculating the contig alignment length N50. By analyzing the contig alignment lengths, as opposed to the length of contigs themselves, I account for misassembled contigs that can inflate the assembly statistics. For SGA, contig alignments 16.8 kbp and greater covered

50% of the reference genome (50 Mbp). ABySS, SOAPdenovo and Velvet had contig alignment N50s of 18.4 kbp, 16.0 kbp and 13.6 kbp, respectively.

3.3.2.3 Assembly Completeness

The contigs assembled by SGA covered 95.9% of the reference genome. The ABySS assembly covered 95.6%, Velvet covered 94.5% and SOAPdenovo covered 94.8%. Figure 3.4 plots the reference genome coverage as a function of minimum contig alignment length. In this assessment ABySS generated the best assembly as it covered more of the reference genome with long contigs.

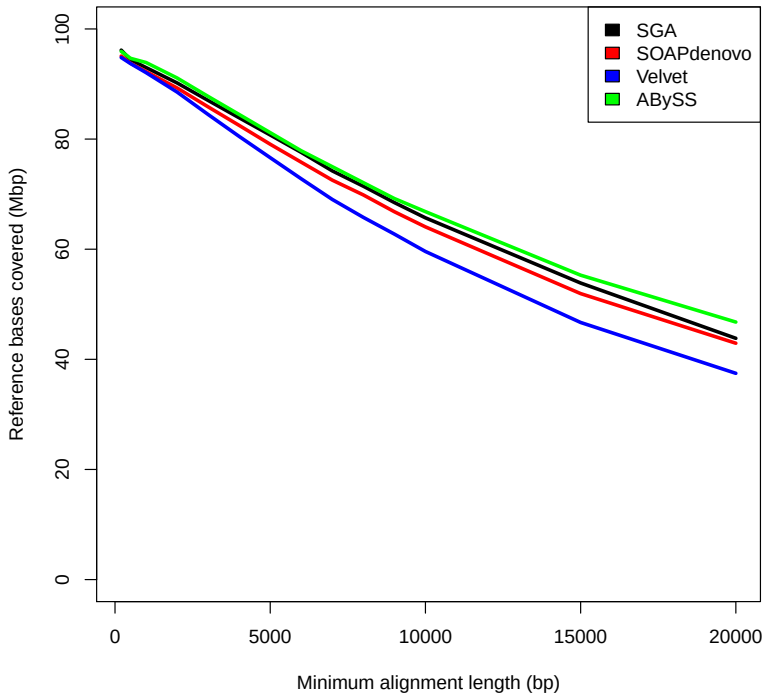


Figure 3.4: The number of bases of the C. elegans reference genome covered as a function of minimum contig alignment length.

3.3.2.4 Assembly Accuracy

I assessed both the structural accuracy and the per-base mismatch rate of the contigs. First, I categorized the contig alignments into three groups. The first group ("full-length") contains contigs that had a single alignment to the reference containing at least 95% of the contig length. The second group ("split") contained contigs that had two alignments to the same chromosome in close proximity (<10,000bp). These split contigs may either contain local assembly errors, or structural variation (for example a large insertion or deletion) with respect to the reference. All remaining alignments ("bad") were partially aligned (< 95% of the contig aligned to the reference), aligned to multiple chromosomes, aligned in greater than 2 pieces or did not align to the reference at all. For all assemblies a substantial proportion of the contigs were found to match the *E. coli* genome. As *C. elegans* eat *E. coli*, this is an expected contaminant and one might suspect other bacterial sequences to also be present. For this reason contigs that did not align to the *C. elegans* reference were not included in this analysis.

For the first measure of assembly accuracy, I counted the number and total size of contigs with split or bad alignments. The accuracy of the SGA and SOAPdenovo contigs was similar - 458 contigs for SGA (totaling 4.4 Mbp) and 483 contigs for SOAPdenovo (4.4 Mbp) had split or bad alignments. Velvet and ABySS had 787 contigs (7.2 Mbp) and 638 contigs (9.1 Mbp) with split or bad alignments, respectively.

For the second accuracy assessment, I calculated the rate at which aligned contig bases did not match the reference. In this assessment, I used the fully-aligned contigs only. I evaluated each assembly at all reference positions covered by its contigs, and also at the subset of positions that were covered by all assemblies. The latter case provides a fairer basis for comparison, removing the effect of differences of coverage of repetitive or complex sequence between the four assemblies. The results are summarized in table 3.2. Again, the accuracy of the SGA and SOAPdenovo assemblies was comparable, and both were more accurate than Velvet and ABySS. The mismatch rate of the SGA assembly at reference positions assembled by all four programs was approximately 1 mismatch per 83 kbp. SOAPdenovo, Velvet and ABySS had error rates at shared positions of 1

per 81 kbp, 1 per 18 kbp and 1 per 8 kbp, respectively.

3.3.2.5 Computational Requirements

Of the four assemblers, SGA used the least memory (4.5 GB vs 14.1 GB, 23.0 GB and 38.8 GB for ABySS, Velvet and SOAPdenovo, respectively). The de Bruijn graph assemblers were considerably more computationally efficient however as the SGA assembly required 8 times more CPU hours than ABySS, 20 times more than Velvet and 3 times more than SOAPdenovo. This speed difference is largely due to the time required to build the FM-index. However, we can reuse one FM-index for multiple runs of SGA, for instance to try different error correction or assembly parameters, whereas the de Bruijn table for ABySS, Velvet and SOAPdenovo must be re-calculated for each choice of k.

3.3.3 Human Genome Assembly

As a second demonstration, I assessed the ability of SGA to scale to very large data sets by assembling a human genome. I downloaded 2.5 billion reads (252 Gbp of sequence) for a member of the CEU HapMap population (identifier NA12878) sequenced by the Broad Institute[1]. The reads are 101bp in length from a paired-end insert library of 380 bp mean separation. As the total sequence depth is 84x, I chose to only assemble half the data to reflect typical coverage depths seen for human shotgun sequence data sets.

I constructed an FM-index for subsets of 20 million reads at a time (using the `sga-sais` variant of our indexing algorithm), then iteratively merged the sub-indices in pairs to obtain a single FM-index for the entire data set. I ran the error correction process using a cluster of computers. Each process used the full FM-index to correct 20 million reads. An FM-index was constructed for the corrected reads, duplicate and low-quality reads were removed, and non-branching chains of reads were merged together. A string graph was constructed from the merged sequences using a minimum overlap parameter $\tau = 77$. I re-aligned the reads to the resulting contig set using bwa [Li and Durbin, 2009] and constructed

[1] ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/technical/working/20101201_cg_NA12878/NA12878.hiseq.wgs.bwa.raw.bam

scaffolds.

In total, the assembly took 1,427 CPU hours across 140 wall clock hours, just under 6 days. The most compute intensive stages were error correcting the reads and building the FM-index of the corrected reads, which each required 355 CPU hours. However these stages were distributed across a cluster of computers by simply splitting the input data, substantially reducing the elapsed (wall clock) time. I ran 123 indexing/merging processes and 63 correction processes; the elapsed time for these stages was 32 hours and 1 hour, respectively. The post correction read filtering stage - where duplicate and low quality reads are discarded - was the memory high-water mark, requiring 54 GB of memory. Complete details of the number of processes, running time and memory usage for each stage of the assembly can be found in table 3.3.

Stage	Processes	Wall time	CPU time	Max Memory
Build index (raw)	123	23 hr	187 hr	45 GB
Correct reads	63	1 hr	355 hr	28 GB
Build index (corrected)	123	32 hr	355 hr	44 GB
Filter reads	1	33 hr	167 hr	54 GB
Merge reads	1	15 hr	105 hr	48 GB
Assemble reads	3	23 hr	41 hr	16 GB
Align to contigs	62	6 hr	210 hr	10 GB
Build scaffolds	4	7 hr	7 hr	13 GB
All stages	-	140 hr	1427 hr	54 GB

Table 3.3: Running time and memory summary for the SGA human genome assembly

I also assembled the data with SOAPdenovo [Li et al., 2010c]. I first error corrected the reads using the SOAPdenovo error correction tool then performed three assemblies, with k-mer sizes 55, 61 and 67. The 61-mer assembly had the largest scaffold and contig N50 and was used for the subsequent analysis. The 61-mer SOAPdenovo assembly (including error correction) required 479 CPU hours across 121 wall clock hours. The maximum amount of memory used was 118 GB. As with the *C. elegans* assembly described above, I did not use the SOAPdenovo GapCloser.

I evaluated the assemblies in terms of contiguity, completeness and accuracy. Note that unlike for the *C. elegans* assembly, in this case the sequenced sample

differs from the reference genome. As in the *C. elegans* analysis, I broke the assembled scaffolds into their constituent contigs, filtered out contigs less than 200bp in length then aligned the remaining contigs to the human reference genome (build GRC 37) using bwa-sw [Li and Durbin, 2010].

The SGA contig alignments cover 2.69 Gbp of the human reference autosomes and chromosome X (95.0% of the non-N portions of these chromosomes). The SOAPdenovo contigs cover 2.65 Gbp of the human reference (93.6%). The SGA contig alignment N50 is 9.4 kbp and the SOAPdenovo contig alignment N50 is 6.6 kbp. The corresponding raw contig N50s are 9.9 kbp and 7.2 kbp. Figure 3.5 plots the amount of the reference genome covered by each assembly as a function of the minimum contig alignment length. Across all contig alignment lengths, the SGA assembly covered more of the reference genome than SOAPdenovo. In contrast, SOAPdenovo gave larger scaffolds (N50 length of 34.8 kbp compared to 25.1 kbp for SGA), though the single short insert library for this data set limits the ability to build larger scaffolds.

The overall assembly accuracy for both SGA and SOAPdenovo was high; 94.5% of SGA contigs (totaling 2.64 Gbp) had full-length alignments to the reference genome, 1.1% (68 Mbp) had split alignments and 4.3% (50 Mbp) had low-quality alignments or did not align at all. 96.8% of the SOAPdenovo contigs had a full-length alignment to the reference (totaling 2.60 Gbp), 1.0% had split alignments (53 Mbp) and 2.2% (33 Mbp) had low-quality alignments or did not align to the reference at all. This is consistent with the SGA assembly being a little larger, covering a little more of the reference but also containing a little more additional material.

I also calculated the per-base mismatch rate of the contigs using the same methodology as the *C. elegans* assembly. In this case, I used the human reference genome combined with SNP calls produced by the Broad Institute in the same individual from the same data set by a mapping rather than assembly based approach [DePristo et al., 2011]. I only counted mismatches at positions that did not match the reference and did not match a Broad SNP call. I also calculated the mismatch rate at the subset of positions assembled by both SGA and SOAPdenovo. As both SNP calling and assembly can be confused by genomic repeats and segmental duplications, I also calculated the per-base accuracy at positions

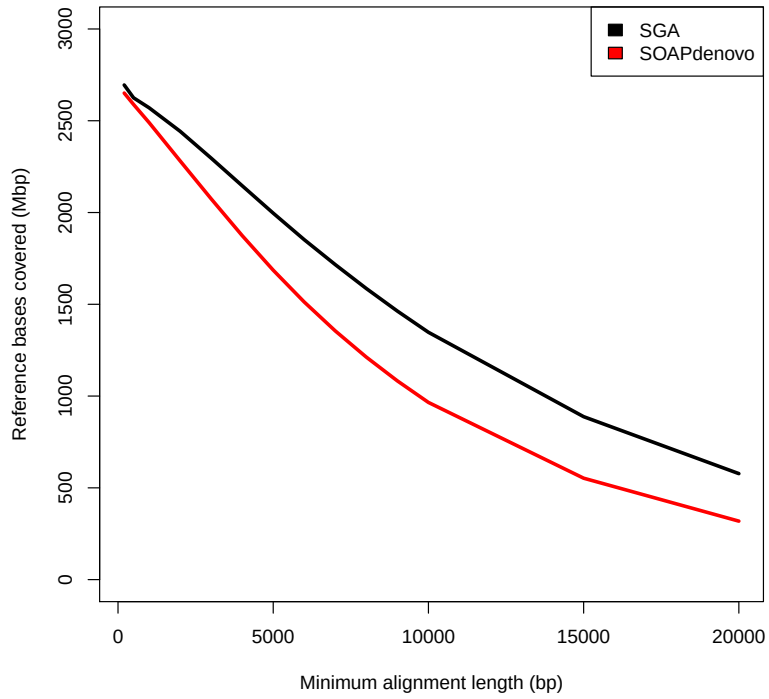


Figure 3.5: The amount of the human reference genome covered by a contig as a function of the minimum contig alignment length. For each length L on the x-axis, contig alignments less than L bp in length were filtered out and the amount of the reference genome covered by the remaining alignments was calculated.

of the reference that are not masked by RepeatMasker[1] and not annotated as segmental duplications (1.3 Gbp of the reference genome remains after this filter). Both assemblies were highly accurate. The mismatch rate for SGA over all covered positions of the reference was 1 per 3,574 bp. For SOAPdenovo, the mismatch rate was 1 per 4,285 bp. If I only consider reference positions covered by a contig from both assemblies, the mismatch rates are 1 in 4,325 bp for SGA and 1 per 5,041 bp for SOAPdenovo. When restricting the analysis to positions not masked by RepeatMasker and not annotated as segmental duplications, the mismatch rate is 1 per 52,464 bp for SGA and 1 per 51,125 for SOAPdenovo. At

[1] http://www.repeatmasker.org

positions assembled by both programs and not masked as repeats or segmental duplications, the mismatch rates are 1 per 59,884 bp and 1 per 60,511 bp, for SGA and SOAPdenovo, respectively.

I note that both the contig mismatches and the mapping-based SNP calls will contain false-positive variants due to mapping errors between the contig or read sequences and the reference. These false positives will have an opposing effect; if the contig sequence is misaligned to the reference, we may count a mismatch in the assembly that is not truly present. This will cause the error rate in the assembly to be overestimated. It is also possible that false positives from misalignments in the mapping-based call set may overlap errors in the assembly. This would lead to an underestimate of the assembly error rate. As I cannot assess the magnitude of these effects it is difficult to accurately estimate the true base-level error rate in the assemblies. However, if we conservatively consider all remaining mismatches to be assembly errors it would indicate the per-base accuracy of the SGA and SOAPdenovo assemblies are very similar and better than 1 error in 50 kbp in non-repetitive regions. The accuracy of SGA is supported by an independent assessment of our assemblers performed during the Assemblathon competition, which is described in the next section.

3.3.4 The Assemblathon

In 2010, a community organized project was launched with the goal of providing a simulated data set to benchmark and evaluate assembly software. This project was organized by UC Davis and UC Santa Cruz. They simulated a diploid genome derived from human chromosome 13. The organizers sampled simulated sequence reads from this genome from both short insert (200-300bp paired end separation) and long insert (3kb and 10kb) libraries. With the goal of modelling real sequence data, the organizers introduced base-calling errors, PCR duplications and bacterial contamination [Earl et al., 2011]. The sequence reads were openly released to the community and the developers of assembly software were invited to submit assemblies of the data. The organizers performed the analysis of the submitted assemblies, providing an unbiased comparison of assemblers on simulated data. I entered SGA into the competition. In the assessment, SGA had the largest scaf-

fold path NG50 (a measure of scaffold length, corrected for assembly errors), the lowest number of substitution errors, and the second lowest number of structural errors [Earl et al., 2011], highlighting the accuracy of my software. Overall, SGA placed 3rd out of 17 groups, behind ALLPATHS-LG [Gnerre et al., 2011] and SOAPdenovo [Li et al., 2010c].

3.3.5 *Schizosaccharomyces pombe* assemblies

As a final assessment of SGA, I assembled 104 strains of the fission yeast, *Schizosaccharomyces pombe*. These strains were sequenced as part of a project to determine the genetic diversity across the *S. pombe* population. There are two groups of strains. The first group (97 strains) had 65X sequence depth on average (range 29-91X). The second group (7 strains) were sequenced much deeper (mean 210X, range 107-436). I assembled each strain with SGA using a minimum overlap parameter (τ) of 65. Additionally for the $\geq$ 100X strains, I set a fixed threshold of 5 k-mer occurrences when running error correction. For the other strains, this parameter was automatically learned from the data.

With such a high number of samples, I am able to evaluate the impact of sequence coverage depth on contig N50, as well as run time and memory usage. The relationship between coverage and contig N50 is shown in figure 3.6. Contig N50 increases with coverage up to 100X, likely due to being able to use a longer overlap at higher coverage. Beyond 100X, adding additional coverage does not help and may actually be detrimental to assembly contiguity at very high depth (>200X). The relationship between coverage and CPU time is almost perfectly linear (figure 3.7).

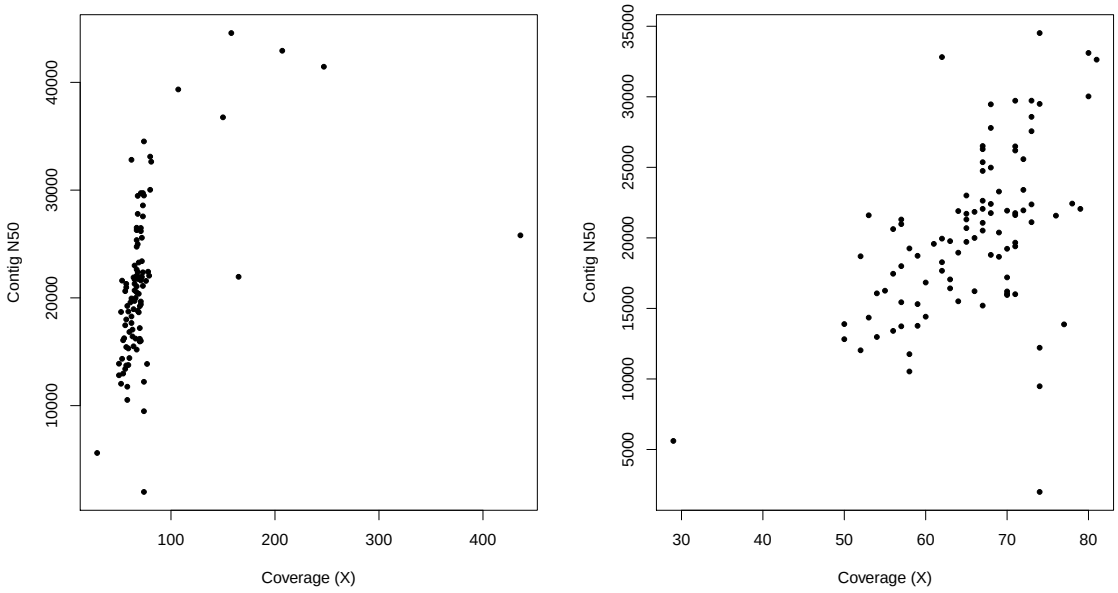


Figure 3.6: The relationship between sequence coverage and contig N50 for the S. pombe data set. The plot in the left panel displays the complete data set. The plot in the right panel only shows strains that have <100X coverage.

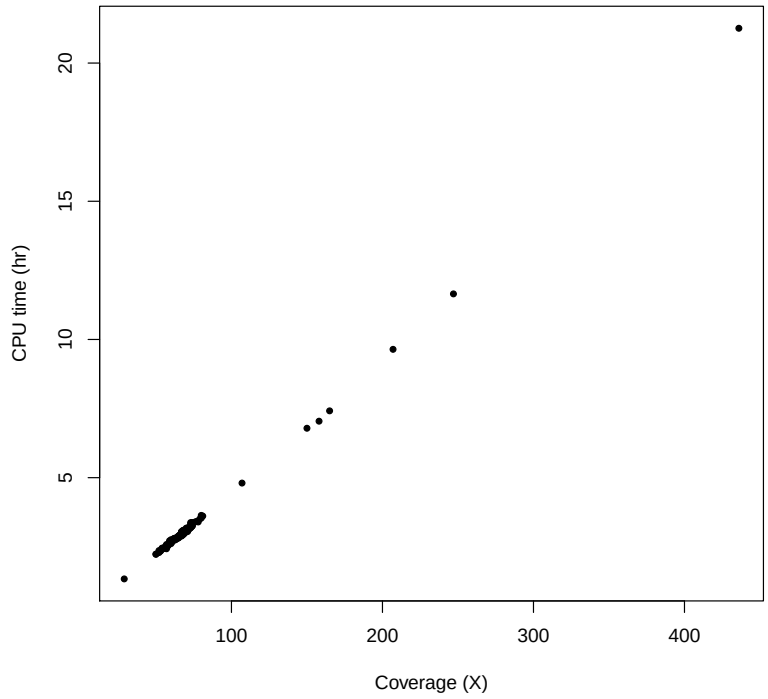


Figure 3.7: The relationship between sequence coverage and CPU time for the S. pombe data set.

Chapter 4

Algorithms for Variant Detection from an Assembly Graph

4.1 Introduction

In the preceding chapters, I described algorithms for assembling the complete genome of an individual from a set of sequence reads. Often we are only interested in the ways in which two (or more) genomes differ. For example, we may be interested in the differences between a sequenced individual and the reference genome for a species, or differences between a child and its parents. I will refer to the problem of finding genomic differences between closely related genomes as *variant detection*.

Reference-based variant detection algorithms map and align reads to a reference genome then find substrings of the reads that are consistently different with respect to the reference. As reads contain sequencing errors, the differences between the reads and reference are typically assessed using a probabilistic model to help distinguish between errors (either due to base-calling errors during sequencing or mis-aligning the reads to the reference) and true variants. While mapping-based algorithms have become the standard method for variant detection, it is considerably more difficult to accurately find indel variants than it is to find substitution variants [Li and Homer, 2010]. For this reason, assembly-based variant calling algorithms have been proposed [Catchen et al., 2011; Iqbal et al.,

2012]. In chapter 3 we saw how allelic differences present in a diploid genome form "bubbles" in the assembly graph. Typically assemblers will find and remove these structures to output a linear contig, with one of the two possible alleles chosen to represent the locus in the contig. Most assemblers will catalog these structures for later consideration as candidate variants[1]. However, for a diploid genome this will only find heterozygous sites. The Cortex program [2012] uses multi-colored de Bruijn graphs to directly compare the sequences of two or more genomes - each color in the graph represents a single individual. Walks following a single color in the graph provide partial assemblies of a single individual. By analyzing the pattern of colors through bubbles in the graph, variants can be found and attributed to particular individuals.

The method I discuss below is conceptually similar to that of Cortex with the major difference being that we use the FM-index as the underlying representation of the assembly graph, instead of explicitly using a hash-table based de Bruijn graph with a fixed k-mer size. As described in Chapter 2, the FM-index can represent both the de Bruijn graph (for all k up to the read length) and the string graph. We use this property to develop both de Bruijn graph and string graph-based variant detection algorithms. Additionally, we can use the fact that the FM-index stores the complete sequence of each read to extract all reads harboring a potential variant and use them as input into a Bayesian model to distinguish between true variants and sequencing errors.

The remainder of this chapter will describe the algorithms I have developed. In the following chapter I demonstrate the versatility of these algorithms by finding polymorphisms present in a human population, finding *de novo* mutations acquired by a child with respect to its parents and to discover mutations occurring in a tumour with respect to an individual's inherited genome.

4.1.1 Collaboration Note

The methods described in this chapter were developed in collaboration with Cornelis Albers. The variant detection, haplotype assembly, haplotype alignment

[1] ABySS and SGA write the sequences of the bubbles to a file, ALLPATHS-LG uses a marked-up FASTA file to describe the ambiguity in the assembly.

and read extraction algorithms are by the author. The probabilistic realignment model described in section 4.3 was developed by Cornelis Albers. Its description is included in this text to complete the description of the variant calling model.

4.2 Algorithms

We use the FM-index to represent the assembly graphs formed from the sets of sequencing reads. We will typically compare two sets of sequences against each other. We will call one set of sequences the *control* set and one set of sequences the *variant* set. We will call the underlying genomes G_c and G_v, respectively. The *variant* sequences will always be a set of sequence reads drawn from G_v. We will denote this read set as $\mathcal{R}_v$. The control sequences can either be a set of reads or the chromosomes of a reference genome. Below, we will describe the algorithms in general terms to cover both cases, with minor modifications that will be stated. We will refer to the set of control sequences in general as $\mathcal{C}$.

Our goal is to determine the loci in G_v that differ with respect to G_c. When a reference genome G_r is available, we will use it to provide a common coordinate system to describe the differences between G_v and G_c. The differences between one of the genomes and G_r will be described in terms of changes to G_r with tuples of the form:

<`reference-position, reference-sequence, variant-sequence`>[1]

These tuples encode the information required to locally change G_r into G_v at the variant site. Let $\mathcal{A}_{vr}$ be the set of tuples describing differences between G_v and G_r and $\mathcal{A}_{cr}$ be the corresponding set of tuples describing differences between G_c and G_r. The set of positions we are interested in finding is $\mathcal{B} = \mathcal{A}_{vr} - \mathcal{A}_{cr}$. When the control genome is the reference genome, this is just $\mathcal{A}_{vr}$.

We begin by building an FM-index for each of $\mathcal{R}_v$ and $\mathcal{C}$ using the methods presented in Chapter 3. We load the pair of FM-indices into memory, then our algorithm has four stages. First, we find a set of substrings that are present in $\mathcal{R}_v$ but not $\mathcal{C}$. These substrings may cover locations in G_v that are different with respect to G_c. We then extend these substrings into candidate haplotypes using

[1] This is information is typically encoded in a Variant Call Format (VCF) file

the joint assembly graph of $\mathcal{R}_v$ and $\mathcal{C}$, as represented by the pair of FM-indices. We then align the candidate haplotypes to G_r. Finally, we use a probabilistic model to assess whether the assembled haplotypes represent true differences between G_v and G_c or sequencing errors. The tuples for variants that are called by our probabilistic model are output to a VCF file. Each stage is discussed in detail below.

4.2.1 Motivating Example

Consider the simple case where G_v and G_c are random genomes that differ at a single position. For some k large enough to avoid spurious matches between unrelated sequence, let $S_v = K_1xK_2$ and $S_c = K_1yK_2$ be the $2k+1$ substrings of G_v and G_c surrounding this single difference. $S_v[1,k] = S_c[1,k] = K_1$ and $S_v[k+2, 2k+1] = S_c[k+2, 2k+1] = K_2$ are the k-mers that occur immediately before and after the single difference. These k-mers are shared between G_v and G_c. The k-mers covering x and y are unique to G_v and G_c. There are k such k-mers, which are the substrings $S_v[1+i, 1+i+k]$ for all $i \in \{1..k\}$ (respectively, S_c). Under our assumption that k is sufficiently large, then $S_v[1+i, 1+i+k]$ k-mers are unique to G_v. It is this set of k-mers that we wish to find as the set of candidate variant k-mers. This situation is depicted in figure 4.1.

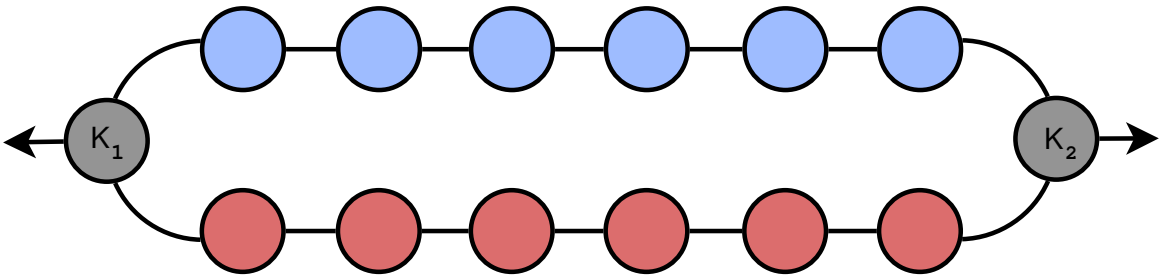


Figure 4.1: A bubble in a de Bruijn graph built from G_v and G_c. The grey k-mers (labelled K_1 and K_2) are shared between G_v and G_c and are the entry/exit points of the bubble. The red and blue vertices represented k-mers unique to G_v and G_c, respectively.

Once we have found the candidate variant k-mers, we assemble them into

haplotypes. In the context of the de Bruijn graph[1] shown in figure 4.1, we would start the haplotype generation process from one of the six red k-mers. The haplotype generation would perform a breadth-first search by starting from a red node and continue until both grey junction nodes have been found. The set of nodes found during this search would define a path through the red half of the bubble. We can calculate the assembly string corresponding to this path and output it as a candidate haplotype. We can perform an additional search between the two grey junction nodes using just the control sequences. This will search the blue half of the bubble and generate candidate haplotypes in the control sequences. It is worth noting that these haplotypes may also be present in the variant sequences.

After haplotypes have been generated, we align them to the reference genome then extract all the raw sequence reads from the FM-index that share a k-mer with a candidate haplotype. The candidate haplotypes, their alignments to the reference and the raw sequence reads are input into the Bayesian model which assesses the evidence for each haplotype and makes the final variant calls.

4.2.2 Discovering Candidate Variants

The first stage of the algorithm attempts to find k-mers of G_v that are not present in G_c. If we know the sequence of G_v and G_c this problem is easy. We could decompose the genomes into their k-mer sets $\mathcal{K}_v$ and $\mathcal{K}_c$ and compute the set $\mathcal{D} = \mathcal{K}_v \setminus \mathcal{K}_c$. Of course, we do not know the full sequence of G_v so we must approach the problem from a different direction. In Pevzner's original paper on de Bruijn graph assembly [2001], he observed that the set of k-mers present in a set of sequence reads drawn from G_v approximates the set of k-mers of G_v itself. In Chapter 3 we used this fact to correct substitution sequencing errors. Here, we use it again to solve the problem of finding k-mers unique to G_v. We could explicitly subtract the k-mers of $\mathcal{C}$ from the k-mers of $\mathcal{R}_v$ but this would require the intermediate storage of the full k-mer sets. Instead, we developed an efficient streaming algorithm.

For the moment we will ignore sequencing errors. The algorithm begins by

[1] A string graph based algorithm is given in section 4.2.4

iterating over all reads in $\mathcal{R}_v$ and all k-mers in each read. For each k-mer, Q, we use the FM-index to count the number of occurrences of Q in $\mathcal{R}_v$ and the number of occurrences in $\mathcal{C}$. If Q only appears in $\mathcal{R}_v$, we emit it as a candidate variant. As the same k-mer may appear in multiple reads, we want to avoid emitting duplicate k-mers. To do this, we use an efficient bit-vector marking procedure. At the program's start, we create a bit vector B with one bit per base in $\mathcal{R}_v$, initialized to zero. When we first visit a k-mer Q, we calculate its suffix array interval $[l_Q, u_Q]$ and set the bit $B[l_Q]$ to be one. Subsequent visits to Q will see that $B[l_Q]$ is one and skip it as it has already been visited. This avoids emitting duplicate candidate variants for the same k-mer and also accelerates the search by avoiding redundant k-mer occurrence queries.

An alternative approach to finding k-mers unique to $\mathcal{R}_v$ can avoid traversing over each read in $\mathcal{R}_v$. We can simulate a breadth-first traversal of the implicit suffix tree represented by the FM-index of $\mathcal{R}_v$, stopping once we have reached depth k. Such a traversal operates over suffix array intervals instead of individual k-mers, and hence visits each distinct k-mer only once without the need of the bit vector. When we visit a k-mer, we can check its count in the FM-index of $\mathcal{C}$ to determine whether it should be emitted as a candidate variant. While this algorithm is cleaner than iterating over every read in $\mathcal{R}_v$, it is faster in practice only for small k. The bit-vector based algorithm is very easy to parallelize using multiple threads. Our implementation uses atomic marking (implemented with compare-and-swap instructions) to allow concurrent updates of the bit vector without requiring locks. This highly parallel implementation makes the k-mer discovery portion of the algorithm very fast in practice.

In the presence of sequencing errors, the above algorithms require minor modifications. As discussed in the previous chapter, sequencing errors generate low-frequency k-mers. To help distinguish between unique k-mers arising from errors and unique k-mers arising from true variants, we set a threshold of d (typically 3-5) on the minimum number of occurrences of Q in $\mathcal{R}_v$ to emit the k-mer as a candidate variant. As an additional filter we require that both Q and $\overline{Q}$ are present in $\mathcal{R}_v$ - this requires that k-mer is seen on both sequencing strands, which helps discard systematic errors [Meacham et al., 2011]. Algorithm `generateCandidateVariant` encapsulates the procedure for finding candidate

variants.

Algorithm 12 generateCandidateVariants(k, d, $\mathcal{R}_v$, $\mathcal{C}$) - find candidate variant substrings

for all $R \in \mathcal{R}_v$ **do**
  $n \leftarrow |R| - k + 1$
  for $i = 1 \rightarrow n$ **do**
    $Q \leftarrow R[i, i + k]$
    if not `isMarked`(Q) **then**
      `mark`(Q)
      $v_f \leftarrow$ `countOccurrences`$(Q, \mathcal{R}_v)$
      $v_r \leftarrow$ `countOccurrences`$(\overline{Q}, \mathcal{R}_v)$
      $c \leftarrow$ `countOccurrences`$(Q, \mathcal{C})$ + `countOccurrences`$(\overline{Q}, \mathcal{C})$
      if $c = 0$ and $v_f > 0$ and $v_r > 0$ and $v_f + v_r \geq d$ **then**
        `emit`(Q)

Once the candidate variant k-mers have been found, we attempt to assemble them into haplotypes. We have two procedures for doing this, one which uses a de Bruijn graph and one which uses a string graph. We describe both below.

4.2.3 de Bruijn graph haplotype generation

Let Q be a variant k-mer found during the previous portion of the algorithm. If Q represents a true difference with respect to G_c it will lie on one branch of a bubble in the de Bruijn graph formed from the union of $\mathcal{R}_v$ and $\mathcal{C}$. For each vertex (k-mer) of this de Bruijn graph, we can indicate whether it is a k-mer from $\mathcal{R}_v$, $\mathcal{C}$ or both (see figure 4.1). By definition, Q is a k-mer present only in $\mathcal{R}_v$. The algorithm to assemble Q into a candidate haplotype proceeds by performing a breadth-first search starting from Q. The search proceeds until we find k-mers that are present in both $\mathcal{R}_v$ and $\mathcal{C}$. In the context of figure 4.1 we would start the search on one of the red k-mers, and perform the breadth first search outwards in both directions until one of the grey shared k-mers is reached. These *join* k-mers are the entry/exit points of the bubble. As sequencing errors generate new k-mers and paths in the graph, when searching the graph we ignore k-mers

that have not been seen at least m times (typically m is 1 to 3). Pseudocode for this algorithm is presented in `generateDeBruijnHaplotypes`.

The `generateDeBruijnHaplotypes` algorithm begins by initializing an empty de Bruijn graph (line 1) and two empty arrays (lines 2 and 3). The arrays will hold the join vertices, where the two halves of the bubble converge. We perform a breadth-first search for these vertices, starting at Q. Lines 5-7 initialize a direction-specific traversal queue with an element for searching from the prefix (left) side of Q and an element for searching from the suffix (right) side of Q. As de Bruijn graphs can be very complex in repetitive regions, we set a limit on how far we will search before aborting the process (lines 9 and 10). The algorithm then loops over all elements of the queue (lines 12-28). As each node is popped from the queue, its neighbors in the de Bruijn graph are found (lines 15-27) and added as vertices if they meet the minimum coverage parameter of m (lines 23,26). If the vertex is found in both $\mathcal{R}_v$ and $\mathcal{C}$, then the vertex is added to the LEFT or RIGHT join array, depending on the direction of traversal (lines 22-24). If the vertex is only present in $\mathcal{R}_v$, it is enqueued and the main loop starts over. Once the graph exploration phase of the algorithm is complete if we have found left and right join vertices we generate candidate haplotype strings by following the graph through each pair of join vertices (lines 28-33). This uses the function `buildHaplotypes` which takes a pair of vertices in the de Bruijn graph and generates all possible paths between the pair of vertices and returns the corresponding assembly string for each path. The strings generated by this procedure are the candidate haplotypes covering the input k-mer.

After candidate variant haplotypes have been generated, we use a similar procedure to generate candidate haplotypes using the control sequences. We perform a directed search of the de Bruijn graph of the control sequences between each pair of join vertices. The assembly string for each path found during this procedure is added to the set of candidate haplotypes.

Algorithm 13 generateDeBruijnHaplotypes(Q, k, m, $\mathcal{R}_v$, $\mathcal{C}$) - assemble candidate variant into a haplotype

```
1: init(graph, Q)
2: joins[LEFT] ← ∅
3: joins[RIGHT] ← ∅
4: queue ← ∅
5: append(queue,kmer=Q, direction=LEFT)
6: append(queue,kmer=Q, direction=RIGHT)
7: iterations ← 0
8: max_iterations ← 10000
9:
10: while queue not empty and iterations < max_iterations do
11:    n ← pop(queue)
12:    S ← n.kmer
13:    for all b ∈ {A, C, G, T} do
14:       if n.direction is LEFT then
15:          T ← bS[1, k − 1]
16:       else
17:          T ← S[2, k]b
18:       v ← countOccurrences(T, R_v) + countOccurrences(T̄, R_v)
19:       c ← countOccurrences(T, C) + countOccurrences(T̄, C)
20:       if v ≥ m and c > 0 then
21:          addDBGVertex(graph, T, BOTH)
22:          append(joins[n.direction], T)
23:       else if v ≥ m then
24:          addDBGVertex(graph, T, n.direction)
25:          append(queue,kmer=T, direction=n.direction)
26:    iterations ← iterations +1
27:
28: haplotypes ← ∅
29: if joins[LEFT] not empty and joins[RIGHT] not empty then
30:    for all l ∈ joins[LEFT] do
31:       for all r ∈ joins[RIGHT] do
32:          push(haplotypes, buildHaplotypes(graph, l, r)
33: return haplotypes
```

4.2.4 String graph haplotype generation

The second haplotype generation function uses the string graph. The string graph haplotype generation algorithm is a composition of algorithms described previously in chapters 2 and 3. Like in Chapter 3, we require all reads to be error corrected before inserting them in the graph. Likewise, we only allow exact overlaps between reads. Unlike our whole genome assembly algorithm we do not error correct the full read set. When G_v and G_c are closely related it is expected that they will have very few differences. In this case it would be inefficient to error correct every read, as most would not harbor variation. Instead, we correct each read as it is processed by the algorithm. The algorithm is described at high level in `generateStringGraphHaplotypes`.

We begin by initializing an empty graph, and arrays to hold join vertices. We extract all reads containing the input k-mer Q from the FM-index of $\mathcal{R}_v$. This set of reads is error corrected using the k-mer correction method described in Chapter 3. The corrected reads are inserted into the graph, and exact overlaps between the vertices are computed. Here, we simply use a hash of τ-mer sequences to compute candidate overlaps. The main loop of the algorithm finds "tip" reads in the graph - those that only have a neighbor on one side (a prefix neighbor or suffix neighbor). Reads sharing a substring with a tip vertex are extracted from the FM-Index (by `findNewOverlaps`) and corrected. The newly corrected reads are then added into the graph. As each read is inserted into the graph, we determine if it is a join vertex. If the k-mer at the start of read X occurs in both $\mathcal{R}_v$ and $\mathcal{C}$, we say that X is a left-join vertex. If the k-mer at the end of read X occurs in both $\mathcal{R}_v$ and $\mathcal{C}$, we say that X is a right-join vertex. After all new vertices have been added to the graph, we run Myers' transitive reduction algorithm [Myers, 2005] on the graph. We then attempt to find walks from the left-joins to the right-joins that cover the reads containing the candidate variant k-mer Q. If these reads are covered by walks, the walks are returned as the candidate haplotypes. As in `generateDeBruijnHaplotypes` we set a bound of max_iterations on the number of times to extend the graph before aborting.

Finally, we generate haplotypes for the control sequences using the same procedure as section 4.2.3. Here, we do not explicitly have the set of join vertices

in the implicit de Bruijn graph. Instead, we use the first and last k-mer of each variant candidate haplotype to seed the directed search through the de Bruijn graph.

Algorithm 14 generateStringGraphHaplotypes(Q, k, τ, $\mathcal{R}_v$, $\mathcal{C}$) - assemble candidate variant into a haplotype

```
1: init(graph)
2:
3: joins[LEFT] ← ∅
4: joins[RIGHT] ← ∅
5: iterations ← 0
6: max_iterations ← 1000
7:
8: I ← extractReads(Q, R_v)
9: I_c ← correctReads(I, R_v)
10: for all r ∈ I_C do
11:   addStringVertex(graph, r)
12: while iterations < max_iterations do
13:   T ← findGraphTips(graph)
14:   if T is ∅ then
15:     return ∅
16:   for all t ∈ T do
17:     O ← findNewOverlaps(graph, t, τ, R_v)
18:     O_c ← correctReads(O, R_v)
19:     for all o ∈ O_c do
20:       addStringVertex(graph, o)
21:       if isLeftJoin(o, k, C) then
22:         push(joins[LEFT], o)
23:       if isRightJoin(o, k, C) then
24:         push(joins[RIGHT], o)
25:   myersTransitiveReduction(graph)
26:   if joins[LEFT] ≠ ∅ and joins[RIGHT] ≠ ∅ then
27:     haplotypes ← findHapWalks(graph, joins[LEFT], joins[RIGHT])
28:     if haplotypes ≠ ∅ then
29:       return haplotypes
30:   iterations ← iterations + 1
```

4.2.5 Haplotype quality control

After we generate candidate haplotypes we perform a quality check. For a haplotype string H and a set of reads, let k_{max} be the largest k such that all k-mers in H are seen at least l times in the reads. In other words, all k_{max}-mers in H are seen at least l times in the FM-index but some $(k_{max}+1)$-mers of H are not found l times. We expect that haplotypes that are truly present in a genome and well-covered by sequence reads will have a large value k_{max}. Conversely, if a haplotype is not present in a genome, k_{max} will be very small as it will require random k-mer matches to find covering k-mers (we would expect k_{max} to be $\approx \log(|G|)$ for a random haplotype not present in a genome). We can use these observations to define a quality check on the haplotypes that we assembled above. For a haplotype H, let v be k_{max} for the haplotype in the variant read set $\mathcal{R}$. Let c be the corresponding value for k_{max} for the control sequences. We filter out haplotypes when $c \geq 31$ or when $v - c < 10$. The first check ($c \geq 31$) indicates that the haplotype is well-supported in the control sequence set. In this case it is unlikely that it represents a true difference between G_v and G_c. The second check requires the support for a haplotype to be significantly stronger in the variant sequences than the control sequences. The parameter of l (the number of times each k-mer must be seen) is determined by the control sequences. If we are calling variants between two sets of reads, we use $l = 2$ (every k-mer must be seen twice). If we are calling variants against a reference genome we use $l = 1$.

4.3 Probabilistic realignment

To distinguish between sequencing errors and true variants, we use a probabilistic model to determine how well each candidate haplotype is supported by the raw read sequences. Our FM-index based approach easily allows this, as we are able to efficiently extract the full sequence of each read from the index. Our realignment method begins by extracting reads from the FM-index that may match one of the assembled candidate haplotypes. These reads, along with the candidate haplotypes, are the input into our Bayesian model.

4.3.1 Extracting Haplotype Reads from the FM-Index

Extracting a single indexed read from the FM-index is straightforward. Let $\mathcal{R}_i$ be the read in the indexed sequence collection whose sequence we wish to extract. From the definition of the BWT in section 2.4, we know that the suffix array interval for the empty suffix of $\mathcal{R}_i$ is $I = [i, i]$. Correspondingly, the last base of $\mathcal{R}_i$ is given by $\mathbf{B}_{\mathcal{R}}[i]$. Let this base be denoted by b. We can use the function `updateBackward`(I, b) from section 2.4 to calculate the suffix array interval for the one-base suffix of $\mathcal{R}_i$, consisting of the string $b\$$. The corresponding character in the BWT gives the second-last base of $\mathcal{R}_i$. If we iterate this procedure until we reach the \$ symbol in the BWT string, we will have extracted the complete sequence of $\mathcal{R}_i$, as desired.

The procedure to extract haplotype reads is based on k-mer matches. Let H be a haplotype that we wish to find reads for. Let $K_1, K_2, K_3 ... K_n$ be the sequence of k-mers for a haplotype H. We use the FM-index (of $\mathcal{R}_v$ or $\mathcal{C}$) to find suffix array intervals for each of these k-mers. From these k-mer intervals, we backtrack in the FM-index until we reach the terminating \$ symbols. Once the dollar symbols are reached, we use the lexicographic index (section 2.5.1) to map from the lexicographic order of a read to its numeric index in $\mathcal{R}$. These numeric indices are then used in the procedure described in the previous paragraph to extract the full read sequence.

As some reads will share multiple k-mers with a haplotype, the procedure described above is inefficient. To account for multiple k-mers we cache visited intervals during backtracking. If a previously visited interval is visited during backtracking, we exclude that position from further consideration.

For each candidate haplotype we extract the reads from both $\mathcal{R}_v$ and $\mathcal{C}$ matching the haplotype. The set of haplotypes and their matching raw sequence reads are passed to the probabilistic model.

When performing multi-sample calling, like when calling variants present in a low-coverage population of individuals, we need to associate with each read the sample that it originated from. To do this, we create a single FM-index from all samples. We construct the read set $\mathcal{R}$ such that all the reads for sample i are before all reads for sample j. We can then build a simple interval index associating

a range of indices in $\mathcal{R}$ with which sample those reads came from. When extracting read i from the FM-index, we can then return the sample identifier along with the read sequence.

4.3.2 Probabilistic read-haplotype alignment

The purpose of realigning reads to candidate haplotype is to obtain the likelihoods $P(\mathbf{R}_i|\mathbf{H}_j, \theta)$. Here, $\mathbf{R}_i$ is the sequence of read i as generated by the sequencing machine and extracted from the FM-index in the previous section, $\mathbf{H}_j$ is the sequence of candidate haplotype j assembled in section 4.2.3 or 4.2.4, and θ is the vector of model parameters. As we do not currently use quality scores for the read bases the model parameters include an assumption that each read base is Q20. The parameters also include homopolymer sequencing error indel rates as described in [Albers et al., 2011]. These read-haplotype likelihoods are combined with a suitable prior probability distribution[1] over the haplotypes to infer which haplotypes are present in a sample or population of samples. The model that underlies the likelihood $P(\mathbf{R}_i|\mathbf{H}_j, \theta)$ is the Bayesian network described previously [Albers et al., 2011]. Here we use a fast approximate version of this model. The approximation consists of testing only two seed alignments rather than all possible alignments. The two seed alignments are computed using a 8-base hash of the read and haplotype sequence.

4.3.3 Annotating variants in the candidate haplotypes

The strategy for calling sequence variation in a reference-free fashion is to first determine which haplotypes are supported by the data, and only then to annotate the haplotypes with respect to a particular coordinate system or reference sequence. In principle the alignment of haplotypes to a reference sequence is a post-processing step. However, there are several advantages of having haplotype mapping locations available during the inference of the haplotypes. The confidence in a variant call depends on whether the haplotype(s) containing the variant is supported by the data, and whether the haplotype can be confidently

[1] The choice of prior probability distribution depends on whether we are calling variants by comparing two genomes or multiple individuals sequenced at low coverage

placed onto the reference. If one of these two factors is uncertain the variant call quality will be low. Furthermore, it is desirable to have available a number of statistics for each variant call that can be used for filtering. For instance, it is useful to know how many reads cover the variant without any mismatch. To be able to provide this information it is necessary to know all the possible mapping locations of a haplotype to a given reference sequence. To the end-user it may be also be useful to know that a novel haplotype is strongly supported by the data but cannot be confidently placed.

4.3.4 Aligning haplotypes to a reference genome

We align the candidate haplotypes to the reference genome, G_r. Our alignment method uses the FM-index of G_r to find l-mer seed matches between each haplotype and the reference genome G_r. These candidate alignments are refined by dynamic programming. During dynamic programming, we require a semi-global alignment between the haplotype and the reference (we require an end-to-end alignment of the haplotype but a local alignment to the reference). We do not require the alignment of the haplotype to the reference genome to be unique. For each candidate alignment, we calculate the number of edit *events* in the alignment. An edit event is a contiguous stretch of differences in the alignment between the haplotype and the reference (for example a 5bp deletion counts as one event, not 5). We keep all alignments that have fewer than 9 edit events. For a repetitive haplotype this may result in multiple locations with a reasonable alignment score.

Each alignment location may result in a different set of variants. We also compute a mapping quality for each mapping location using the haplotype-reference alignment scores. This mapping quality will be used in the calculation of the variant qualities as described below.

4.3.5 Comparative variant-calling

In comparative variant calling we have reads for both G_v and G_c and we wish to detect variants that are only found in G_v but not G_c. A primary application of comparative variant calling is finding somatically acquired mutations in a cancer

from a sequenced tumour-normal pair. We describe our comparative variant calling model in these terms. Since a tumour sample is not clonal and many contain entire chromosome duplications or loss, one can not assume a diploid model. We therefore assume that the number of haplotypes present in the tumour sample can be greater than two. For simplicity we made the same assumption for the normal sample.

To deal with a possibly large number of haplotypes, we apply a model selection approach to infer which haplotypes are supported by the reads. In this model selection approach, haplotypes are iteratively added until the improvement to the total score is below the minimum threshold required for adding a new haplotype. After the model selection algorithm has converged, the haplotype frequencies are estimated using the Expectation-Maximization algorithm [Dempster et al., 1977].

The scores for the haplotypes used in the model selection are defined as follows. The increase to the total score by adding a candidate haplotype j to the model is given by

$$\Delta S_j = \sum_i \left(\log P(\mathbf{R}_i | \mathbf{H}_j, \theta) - s_i \right), \tag{4.1}$$

where

$$s_i = \underset{k \in \text{selected haplotypes}}{\arg\max} \log P(\mathbf{R}_i | \mathbf{H}_k, \theta). \tag{4.2}$$

For the first iteration, when no haplotypes have been selected yet, s_i is set to a default minimum score. This minimum score is approximately $\log 10^{-6}$ (Q60), so that in practice a read-haplotype alignment with more than indel (penalty of Q40 outside homopolymer runs) or four mismatches (Q20 per mismatch) will not be above this minimum threshold. This minimum score prevents reads that do not have a reasonable alignment to any of the candidate haplotypes from favoring one haplotype over the other because of irrelevant differences in the read-haplotype likelihood.

To estimate the probability that a candidate variant is a somatic variant, a joint set of candidate haplotypes is created from the candidate haplotypes detected in the normal sample and the candidate haplotypes detected in the tumour sample. Conditional on the joint set of candidate haplotypes, inference in the normal and the tumour sample can be performed independently.

The quality scores for a somatic variant v are next computed as follows:

$$P(v \text{ is somatic}|\mathbf{R}_{\text{normal}}, \mathbf{R}_{\text{tumour}}) = P(v \text{ is present}|\mathbf{R}_{\text{tumour}})P(v \text{ is absent}|\mathbf{R}_{\text{normal}}), \tag{4.3}$$

where $P(v \text{ is present}|\mathbf{R}_{\text{tumour}})$ is the probability that a haplotype containing the variant v is present in the tumour, and $P(v \text{ is absent}|\mathbf{R}_{\text{normal}})$ is the probability that there is no haplotype containing the variant v in the normal sample. The quality score for a variant being present in a sample is calculated as follows:

$$\begin{aligned} &P(v \text{ is not present}|\mathbf{R}_{\text{sample}}) \approx \\ &\quad \prod_{j \in \text{selected}, \mathbf{H}_j \text{contains } v} \Big(1 - P(\mathbf{H}_j \text{ is present}|\mathbf{R}_{\text{sample}})P(\mathbf{H}_j \text{ maps to reference location of } v)\Big) \\ &\quad \approx \prod_j \Big(1 - \big(1 - \exp(-\Delta S_j)\big)P(\mathbf{H}_j \text{maps to reference location of } v)\Big) \end{aligned} \tag{4.4}$$

Thus, the variant quality takes into account both the uncertainty in the presence of the haplotype containing the variant, as well as the uncertainty that each of those haplotypes maps to the location of the variant.

4.3.6 Population calling

The algorithm for population calling is similar to the comparative variant-calling algorithm. The main difference is the calculation of the increase in the score from selecting a haplotype. Instead of Eq. 4.1 we use a multisample EM algorithm to estimate the increase in the likelihood achieved by adding a haplotype j. The log-likelihood for the model consisting of the candidate haplotypes selected in iterations $1, \ldots, k-1$ and candidate haplotype j in iteration k is defined as:

$$\exp L_j^k = \max_{\mathbf{f}_{k-1,j}} \prod_i \sum_{h_i^1} \sum_{h_i^2} P(h_i^1|\mathbf{f}_{k-1,j})P(h_i^2|\mathbf{f}_{k-1,j}) \prod_{l \in \text{reads}} \Big(\frac{1}{2}P(\mathbf{R}_i^l|\mathbf{H}_{h_i^1}, \theta) + \frac{1}{2}P(\mathbf{R}_i^l|\mathbf{H}_{h_i^2}, \theta)\Big) \tag{4.5}$$

Here h_i^1 and h_i^2 are indicator variables for the two haplotypes present in sample i; we explicitly assume a diploid model. $P(\mathbf{R}_i^l|\mathbf{H}_{h_i^1}, \theta)$ is the read-haplotype likelihood computed by the probabilistic realignment algorithm for read l from individual i. $\mathbf{f}_{k-1,j}$ is the vector of haplotype frequencies that is estimated using

the EM algorithm. The frequencies $\mathbf{f}_{k-1,j}$ are optimized subject to the constraint that only the haplotypes selected in iterations $1, \ldots, k-1$ and the candidate haplotype j can have non-negative values; other candidate haplotypes (not yet added to the model) are set to zero.

We then define the score as:

$$\Delta S_j = L_j^k - L^{k-1}, \tag{4.6}$$

with L^{k-1} the log-likelihood of Eq. 4.5 for the candidate haplotype added at iteration $k-1$. Finally, in iteration k we add the candidate haplotype with the largest ΔS_j to the model. Candidate haplotype are added to the model until there is no candidate haplotype with a score ΔS_j above the threshold.

4.4 Discussion

In this chapter I described a framework for performing assembly based variant calling with a probabilistic model. There are a number of improvements to this model that could be made in the future. In section 4.2.2 we assume that a variant k-mer does not appear in the control sequence set. In the case of high-depth sequence data, there may be sequencing errors that generate k-mers in the control sequences that match the variant k-mers by chance. These erroneous k-mers may mask the presence of variant k-mers and cause our model to miss variants. In practice this is not a significant problem because there is redundancy in the k-mer detection step, as up to k k-mers may contain the variant sequence - we will detect the variant k-mer if any of these is unique to the variant sequence set. When k is greater than half the read length, the same sequencing error would need to occur in multiple reads to mask all of these k-mers. Despite this redundancy in detection, we are likely to lose some variant calls due to errors, therefore this is a possible point of improvement.

In our probabilistic model, we do not use the per-base quality scores output by the sequencing instrument. An obvious point of improvement is to incorporate these into our model. Quality scores are typically encoded using a single ASCII character, which requires one byte per base. If we naively recorded the quality

scores for each base, this amount of memory would be far larger than the size of the FM-index to store the reads. In the future we intend to investigate other means of storing and accessing the quality values, including compressed representations (for example, Huffman coding) or downsampling the quality scores to a smaller range (for example using 2 bits per score by quantizing the scores to 4 levels).

Two recently published programs also take an assembly-based approach to variant calling. Cortex [Iqbal et al., 2012] builds a colored de Bruijn graph from the sequence reads from multiple individuals. It then searches for diverging paths in the graph, which are assembled into haplotypes. The haplotypes are mapped to the reference genome in a post-processing step. While my fundamental approach - finding divergent paths through an assembly graph built from multiple individuals - is similar to Cortex there are a number of important differences. The FM-index represents all de Bruijn graphs for k up to the read length. This allows flexibility in parameter choice as the graph does not need to be reconstructed for every k. Cortex represents the graph as a fixed hash table of k-mers and therefore needs to construct a new graph for every k that is used. The FM-index also allows string graph-based haplotype generation, as demonstrated in section 4.2.4[1]. Finally, the FM-index provides access to the full read sequences, allowing the haplotypes to be assessed in our probabilistic model after assembly (section 4.3).

Fermi [Li, 2012] uses modified versions of the algorithms in Chapter 2 and 3 to assemble reads into contigs using a string graph. After assembly the contigs are aligned to a reference genome and variants are parsed from the alignments. Fermi performs full assembly, in contrast to Cortex and the algorithms described in this chapter which only assemble the haplotypes that are expected to contain variation. The author of Fermi demonstrates impressive performance for human genome variation detection, with SNP calling sensitivity approaching that of mapping-based methods. However at this time Fermi is limited to single samples and does not support comparing multiple individuals.

[1] In the following chapter the difference in performance between the de Bruijn graph and string graph based approaches will be explored

Chapter 5

Assembly-Based Variant Calling Results

5.1 Introduction

In the previous chapter, I described methods for finding sequence variation by comparing sets of reads using a de Bruijn graph or string graph. In this chapter, I explore this approach to variant calling.

In section 4.2.2 I proposed that k-mers unique to a particular genome (or set of reads) can be used to discover candidate variant sequences. In section 5.2 I test this idea by simulating random mutations in the human reference genome. In sections 5.3 and 5.4 I test the full variant calling pipeline by simulating variants and sequence reads. The benefit of using simulated data is that the true set of variants is known, which allows direct calculation of the sensitivity and precision of the variant calling procedure. However, these simulations do not model some complications found in real data, like biased sequencing coverage, systematic sequencing errors or large structural variants. The remainder of the chapter uses real sequencing data from the Illumina platform. In section 5.5 I make variant calls for an individual genome compared to the human reference sequence. The variant calls for this individual are compared to previously published variants to assess the performance of my method. In 5.6 I explore the false positive rate of my variant caller.

In sections 5.7, 5.8 and 5.9 I apply my variant caller to three key variant calling problems. In section 5.7 I call mutations that occur in the child of two parents (*de novo* mutations) where all three individuals have been sequenced. Section 5.8 explores finding somatic mutations that occur during progression of cancer. Section 5.9 describes the use of assembly-based variant calling for a population of individuals, where each member of the population is sequenced at low coverage. The data used in this section is part of the 1000 Genomes Project. The results in this chapter demonstrate the performance of our algorithms and their software implementation in a wide variety of contexts. In the last section of this chapter the results are discussed in a broader context including future areas of work.

5.1.1 Implementation Note

The algorithms from Chapter 4 are implemented within my FM-Index assembler, SGA. Version 0.9.30 of SGA was used for this chapter. The source code is freely available online at `www.github.com/jts/sga`.

5.2 The power to detect variants using unique k-mers

In section 4.2.2 I described an algorithm to find candidate variants between two genomes, G_v and G_c, by finding k-mers unique to G_v. To assess the power of detecting candidate variants using this approach, I performed a simulation by introducing point mutations randomly into the human reference genome (build GRC 37, preprocessed to remove sequence gaps). If any k-mer containing the introduced point mutation is not found in the human reference genome (it is a unique k-mer), I call the mutation *detectable*. If all k-mers containing the point mutation are unique, I say that the mutation forms a *clean bubble*. I performed this simulation for all k from 16 to 71. In each simulation, $10,000,000$ random mutations were introduced.

The results of this simulation are plotted in figure 5.1. When $k = 21$, 93.8% of variants are detectable but only 63.6% of variants form clean bubbles. When $k = 51$, 99.6% of variants are detectable and 93.7% form clean bubbles. These

results highlight the power of using unique k-mers for finding potential variants - even for relatively small k most changes generate unique k-mers which we can use to start the haplotype assembly process described in the previous chapter.

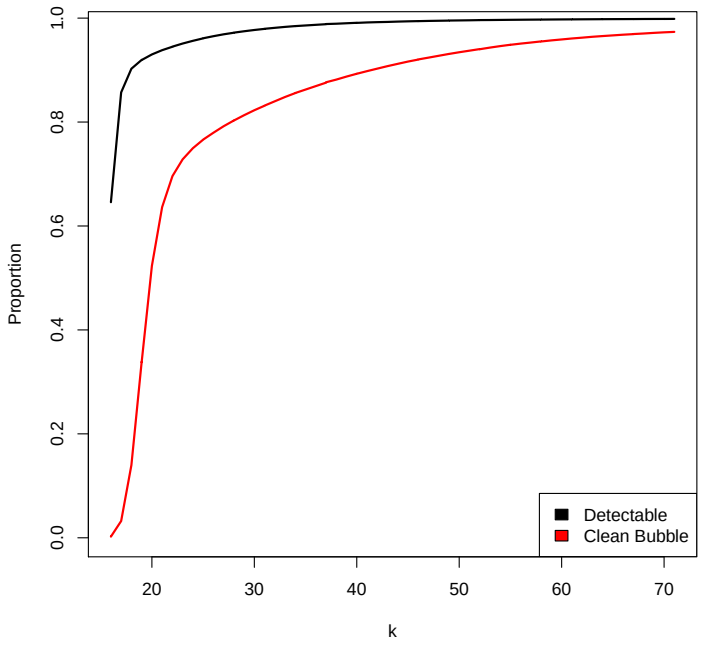


Figure 5.1: The k-mer detectability of point mutations introduced into the human reference genome. The black line indicates the proportion of introduced variants that are detectable at a given k. The red line indicates the proportion of variants that form clean bubbles.

5.3 Simulated single-genome variants calls

As an initial test of our complete variant calling algorithm, I generated two new chromosomes derived from human chromosome 20. First, the human chromosome 20 sequence was pre-processed to remove "N"s from the reference without changing the coordinate system by choosing a random base for each "N" symbol. I then generated two sets of mutations - one set of homozygous changes and one

set of heterozygous changes. Each set contained random substitution mutations at a frequency of 1 in 2,000bp and random indel mutations at frequency 1 in 20,000bp. The indel size was generated by starting at 1bp and extending the event with probability 0.3. In total, 34,701 homozygous and 34,670 heterozygous events were created. The heterozygous events were randomly partitioned into two subsets. To derive the new chromosomes from chromosome 20, all homozygous events were applied to chromosome 20 and one of the two heterozygous subsets using the tool `FastaAlternateReferenceMaker` from the Genome Analysis Toolkit [DePristo et al., 2011]. I will refer to the derived chromosomes as G_1 and G_2.

I sampled 20X read coverage from each of G_1 and G_2 (100bp reads, uniform 1% error rate) using DWGSIM[1]. The 20X read sets were mixed together into one 40X read set, which simulates random shotgun coverage of a diploid genome. I built an FM-index from the 40X reads using the `sga-bcr` algorithm. Variant calls were made by comparing these reads against the chromosome 20 reference sequence. This is the *reference-based* calling mode of our program - the reference genome serves as the set of control sequences. To assess the performance of the de Bruijn graph haplotype generator (4.2.3) and the string graph haplotype generator (4.2.4), calls were made using both modes. I will refer to these modes as the de Bruijn graph caller and the string graph caller, respectively. To assess the effect of the k-mer parameter, I ran multiple trials with each caller, using k from 33 to 75 in increments of 3. A minimum of 5 variant k-mer occurrences were required to trigger haplotype assembly.

Figure 5.2 plots the sensitivity (true positives / (true positives + false negatives)) and precision ((true positives / (true positives + false positives)) for each caller as a function of k. Here k refers to the variant detection k-mer for the string graph caller and both the variant detection and haplotype assembly k-mer for the de Bruijn graph caller.

[1] https://github.com/nh13/DWGSIM

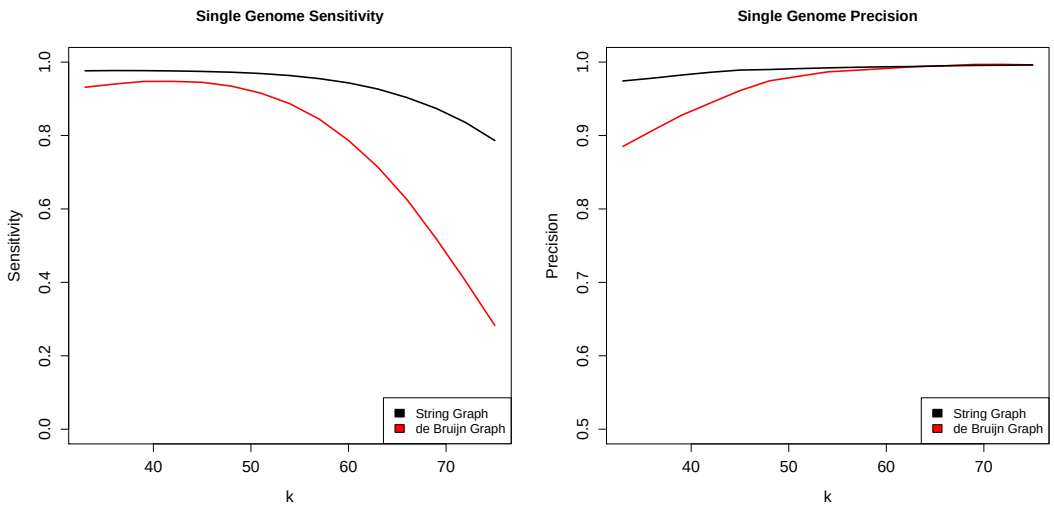


Figure 5.2: Sensitivity (left panel) and precision (right panel) of reference-based calls on simulated data. Note the different range of the y-axis in each panel.

The sensitivity of the de Bruijn graph method decreased sharply with increasing k. This is due to the need to sample a k-mer for every position overlapping the point of variation. When k is large there is insufficient read coverage to ensure that each position has been sampled. This has a weaker effect on the string graph method, as k-mers are only used to detect the variant and a perfect tiling of k-mers is not necessary. Additionally, by performing error correction the string graph caller is able to recover some k-mers that are lost to the de Bruijn graph caller due to sequencing errors.

The peak sensitivity for the de Bruijn method was 0.9476 at $k = 42$. At this k, 96.16% of the homozygous variants were found and 93.34% of the heterozygous variants were found. The increased sensitivity for homozygous variants is expected as they have twice the sequence coverage as heterozygous differences. The sensitivity for indels was slightly higher than that of SNPs (0.9553 vs 0.9469).

The peak sensitivity for the string graph method was 0.9770 at $k = 36$. At $k = 36$, the string graph method recovered 98.50% of the homozygous variants and 96.88% of the heterozygous variants. Like for the de Bruijn graph method, the sensitivity for indels was slightly higher than that of SNPs (0.9821 and 0.9764,

respectively).

At the k-mer chosen to maximize sensitivity the precision of the two methods was 0.9445 (de Bruijn graph) and 0.9781 (string graph). Of the $3,865$ false positive calls for the de Bruijn graph method, 360 (9.3%) are within annotated segmental duplications of the human genome[1]. The string graph method generated $1,520$ false positives at $k = 36$, $1,289$ (85%) of which are in segmental duplications. The lower precision for the de Bruijn graph method is due to the low k-mer used to maximize sensitivity. When a low k-mer is chosen, it is much more likely that a complete bubble forms around sequencing errors. As an illustration of this principle consider the case when k is less than half the read length. When an error occurs in the middle of the read, the erroneous k-mers may be flanked by correct k-mers at the ends of the read. This sequence of k-mers will generate a complete path in the de Bruijn graph between the correct k-mers. When k is high, this situation is much less likely to occur as the sequencing error would need to occur in multiple reads for a complete bubble to form. This effect will be offset to an extent by the requirement that each variant k-mer is seen in at least 5 reads.

In practice the string graph caller with k in the range $50 - 60$ gives better precision (0.9912 to 0.9937) and good sensitivity (0.9690 to 0.9435). For this reason, the default k-mer is set to 54.

5.3.1 Computation Requirements

The de Bruijn graph caller is significantly faster than the string graph caller (10.4 CPU hours versus 24.5 CPU hours, respectively). As both algorithms use the same compressed FM-index, both modes have the same peak memory usage of 1.5GB.

5.4 Simulated genome comparison

Our variant calling model is designed to directly detect variation between two related genomes by directly comparing their sequence reads. I designed a second

[1] as annotated by the UCSC genome browser

simulation to test this method. I started from the pair of chromosomes G_1 and G_2 used in the previous simulation. I generated a new set of substitution variants at frequency 1 in $10,000$bp and indels at 1 in $100,000$bp. These variants were split into two sets and one set was applied to G_1 and one set was applied to G_2 to generate two new genomes G_3 and G_4[1]. I sampled 20X coverage from each of G_3 and G_4 and mixed the reads into one 40X set. Let $\mathcal{R}_A$ be the reads generated from G_1 and G_2 in the previous section and $\mathcal{R}_B$ be the new reads generated from G_3 and G_4 in this section. I made comparative calls using $\mathcal{R}_B$ as the variant sequences and $\mathcal{R}_A$ being the control sequences. Chromosome 20 was used as the reference genome. To trigger assembly, a unique k-mer in $\mathcal{R}_B$ must occur at least 5 times in the $\mathcal{R}_B$ reads and not be present in $\mathcal{R}_A$. Again I used both the de Bruijn graph caller and string graph caller over a range of k.

The results are presented in figure 5.3. The overall trend - that sensitivity decreases as a function k - is similar to the single-genome assessment. At peak sensitivity, the string graph method made slightly more calls (sensitivity 0.9290 at $k = 45$) than the de Bruijn graph method (sensitivity 0.9205 at $k = 36$) and was more accurate (precision 0.9954 vs 0.9752). As in the previous simulation the sensitivity to detect indels was slightly higher (string graph 0.9489 vs 0.9271, de Bruijn graph 0.9425 vs 0.9183).

[1]Note this implies all variants are heterozygous in this simulation

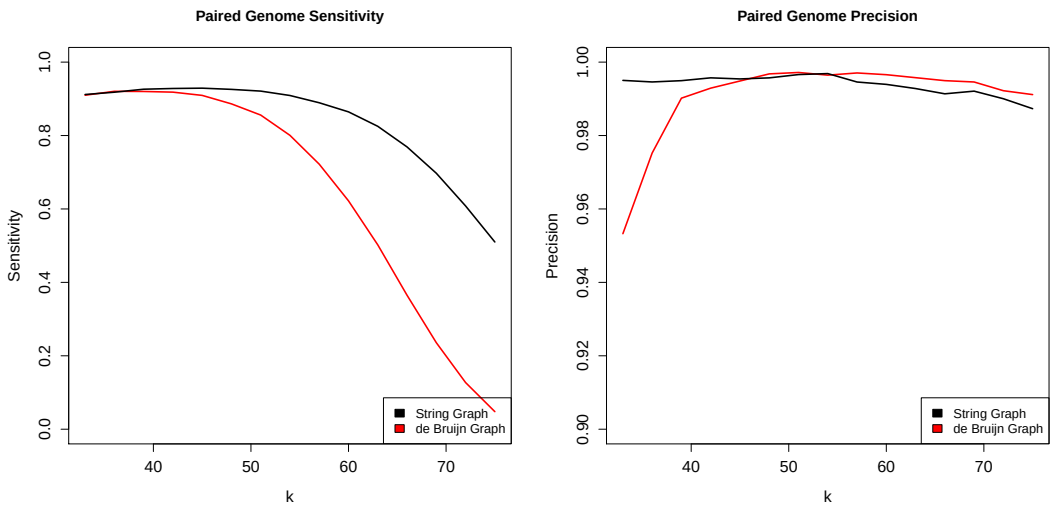


Figure 5.3: Sensitivity (left panel) and precision (right panel) of the simulated genome comparison. Note the different range of the y-axis in each panel.

5.4.1 Computation Requirements

The de Bruijn graph caller required 10.3 CPU hours to make calls at $k = 36$. The string graph caller required 11.4 CPU hours at $k = 45$. Despite having the same number of reads as the reference-based simulation in the previous section both programs were faster in a comparative calling framework. This is particularly true for the string graph caller, which required less than half the time. These results highlight that the number of variants is a crucial determinant of the runtime of the program. The memory usage for both modes was 2.4GB.

5.5 Reference-based Substitution Calls

The results presented above validates that our assembly-based variant calling method can recover the vast majority of simulated SNPs and indels, while retaining high accuracy. Real sequencing data is more challenging however as sequence bias, systematic errors and large structural variants complicate variant calling. To explore the application of our approach to real data, I made reference-based

calls for an individual genome. I used 100bp Illumina sequence data from the NA12878 individual of the CEU population, which has been extensively studied before [Conrad et al., 2011; DePristo et al., 2011; Simpson and Durbin, 2012]. The input data set is available online at `ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/technical/working/20120117_ceu_trio_b37_decoy/`. I used human chromosome 20 as a test case. Reads for this chromosome were extracted from the whole genome BAM file. An FM-index of this subset of reads was created using the `sga-bcr` algorithm. When aligning haplotypes to the reference, I did not limit the alignment to chromosome 20 but rather used the entire reference genome as I found that this helped to reduce the number of false positive variants due to the input reads being mapped to the wrong chromosome. I made two call sets, one using the full set of reads (over 80X coverage) and one using half of the reads. As before, calls were made using both the de Bruijn graph caller and string graph caller.

As the true differences between NA12878 and the reference genome are unknown, I cannot directly evaluate the sensitivity and precision of my variant calls. Instead, I assessed the completeness of my call set by calculating the proportion of mapping-based calls that were found by the assembly callers. The mapping-based calls are from the publication of the GATK variant caller [DePristo et al., 2011]. As a measure of the accuracy of my calls, I compared the calls to variants present in dbSNP v1.32. This build of dbSNP contains variants found by the pilot project of the 1000 Genomes Project which includes NA12878 as a sample. Both of these assessments are restricted to SNP calls. The results are summarized in figure 5.4.

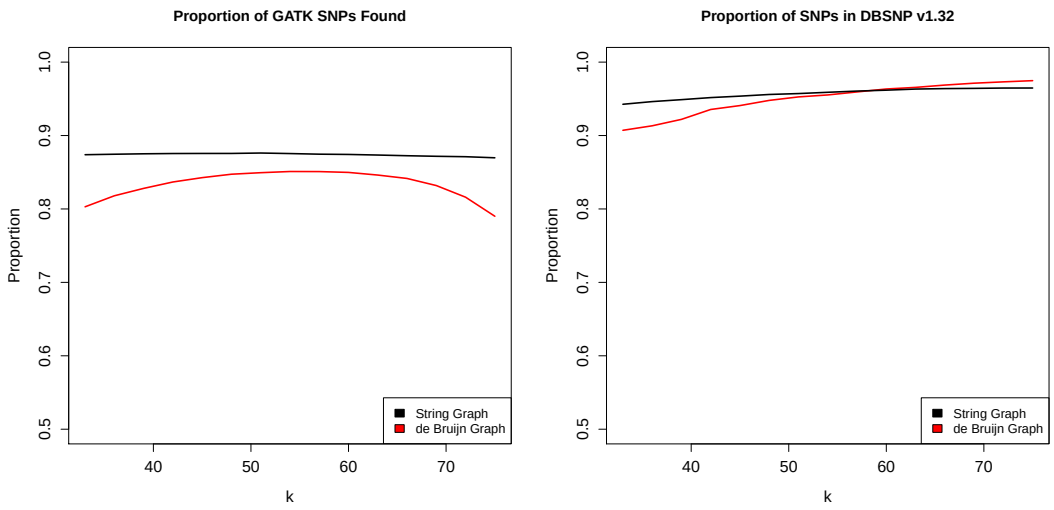


Figure 5.4: The left panel plots the proportion of mapping-based SNP calls using GATK that were found by the de Bruijn graph and string graph callers as a function of k. In the right panel, the proportion of SNP calls that are found in dbSNP v1.32 is plotted.

The peak proportion of mapping SNP calls recovered by the string graph method was 0.8764 at $k = 51$. For the de Bruijn graph method, the peak was 0.8512 at $k = 54$. The performance of the string graph caller was more consistent across the range of k. The proportion of variants found by the de Bruijn graph caller dropped at low and high k-mer values, highlighting the importance of carefully choosing this parameter. Both methods were accurate when assessed by the number of variant calls that are already present in dbSNP v1.32 - 96.88% for the string graph calls ($k = 51$) and 96.74% for the de Bruijn graph calls ($k = 54$) were in dbSNP.

When downsampling the coverage to 42X, the differences between the algorithms become more apparent (figure 5.5). The peak proportion of mapping calls dropped from 0.8764 to 0.8498 for the string graph method ($k = 39$) and 0.8512 to 0.8248 for the de Bruijn graph method ($k = 45$). The profile of the de Bruijn graph caller was similar to the results for simulated data in section 5.3 - there was a steep drop in sensitivity for large k due to lack of coverage. The accuracy was largely unaffected by the decrease in coverage. The proportion of SNPs found

in dbSNP v1.32 for the string graph method at $k = 39$ was 97.25%. For the de Bruijn graph method the dbSNP proportion was 96.87% at $k = 45$.

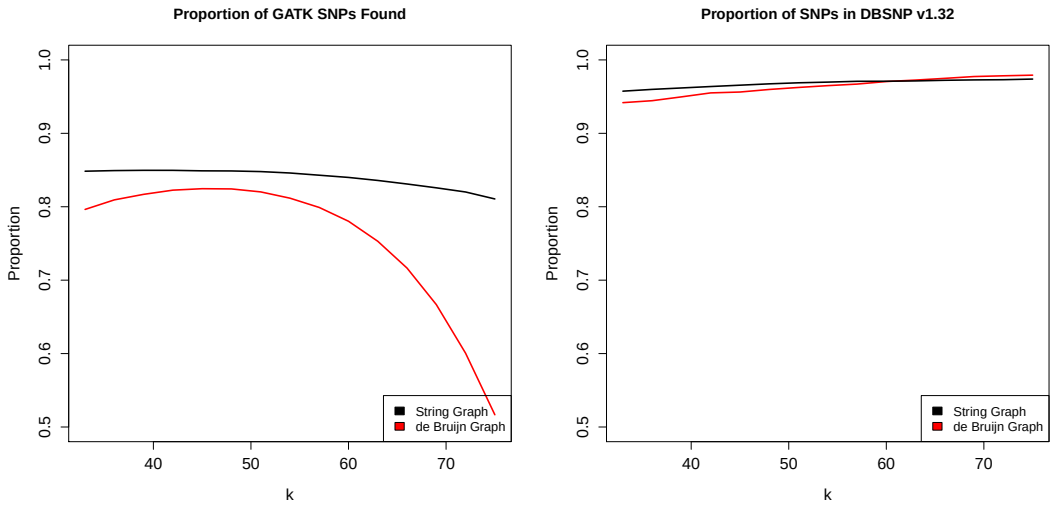


Figure 5.5: The proportion of mapping-based SNPs found (left) and the proportion of our SNP calls contained in dbSNP v1.32 (right) for the downsampled data set.

To further characterize the performance of my assembly based variant caller, I investigated the variant calls that were found by the mapping-based caller (GATK) but not called by the assembly-based callers. In total, GATK called 75,838 substitutions on chromosome 20. Of these, 42,686 (56.3%) are in known segmental duplications or repetitive elements masked by RepeatMasker[1]. The string graph caller ($k = 51$) did not find 9,391 GATK calls in the high depth (>80X) data set. Of these 8,093 (86.2%) lie within annotated repeats. Similarly the de Bruijn graph caller ($k = 54$) missed 11,300 variants, 9,714 (86.0%) of which are in known repeats. This suggests that our ability to make assembly calls is higher in non-repetitive regions of the genome. This is expected as repetitive regions will lead to a more complicated assembly graph making it less likely that clean haplotypes can be assembled. It is also possible that the false positive rate of the mapping caller is higher in these difficult regions of the genome.

[1] Annotations were downloaded from the UCSC genome browser

5.6 Estimating the background error rate for comparative variant calling

I used the NA12878 sequence data to estimate the false positive rate of our comparative variant caller. I split the NA12878 chromosome 20 reads into two subsets of approximately 40X each by randomly assigning each read pair to one of two files. I will refer to these two halves as $\mathcal{H}_1$ and $\mathcal{H}_2$. I made comparative calls by using $\mathcal{H}_1$ as the variant sequences and $\mathcal{H}_2$ as the control sequences. As all the reads were drawn from the same individual no variants should be called - all variants found by this procedure are false positives either due to sequencing errors, assembly errors or incorrectly aligning the assembled haplotypes to the reference genome. As before, I ran the caller in both the string graph mode and de Bruijn graph mode over a range of k. I required 5 occurrences of a k-mer to trigger variant assembly. I classified the errors into three categories - substitutions, indels in homopolymer sequences (a string of ≥ 7 or more occurrences of the same base) or indels outside of homopolymers. The results are summarized in table 5.1. The number of false positive calls drops sharply with increasing k. In all cases, the majority of false positive calls are due to mis-calling the length of a long homopolymer run. This is likely due to the increased sequencing error rate associated with these regions [Albers et al., 2011; Li, 2012].

Table 5.1: False positive variant calls found by splitting the NA12878 chromosome 20 data into two halves. The variants are classified into substitutions (Subs), indels outside of homopolymer runs (non-HP indels) and indels within homopolymer runs (HP indels).

	de Bruijn Graph Calls			String Graph Calls		
k	Subs	Non-HP indels	HP indels	Subs	Non-HP indels	HP indels
33	41	6	118	15	2	48
36	24	6	112	13	4	50
39	20	7	97	15	3	48
42	18	6	83	16	2	45
45	8	4	71	16	2	46
48	5	3	55	16	1	47
51	11	2	39	12	1	44
54	3	2	31	14	1	39
57	1	2	15	10	1	30
60	2	2	6	10	1	22
63	2	0	4	7	1	18
66	0	1	2	6	1	17
69	0	1	2	7	2	9
72	0	0	0	2	2	5
75	0	0	0	2	3	3

5.7 Calling *de novo* mutations in a trio

I will now describe the application of our comparative variant caller to real sequencing problems. The first problem I will address is the discovery of *de novo* mutations. These are mutations that occur in the germline of an individual's parents and are subsequently passed along to the child. *De novo* mutations have been implicated in a number of human diseases including schizophrenia [Girard et al., 2011] and autism [Sanders et al., 2012]. To find *de novo* mutations the genome of a child is sequenced along with the genome of both of its parents (this is commonly referred to as a sequencing a "trio"). Conrad et al. [2011] developed an algorithm to call *de novo* mutations using reads mapped to a reference genome. Their framework considers the three individuals jointly in a Bayesian

framework[1].

I used our assembly-based approach to call *de novo* mutations in a trio from the CEU population, which was also studied by Conrad et al. [2011] using lymphoblastoid cell line DNA. It is known there are many somatic cell line mutations in this sample. The individual NA12878 used in section 5.5 is the child in this trio. Conrad et al. used early Illumina sequencing data from the pilot of the 1000 Genomes Project. In this section, I use more recent data consisting of 101bp reads[2]. In this data set, NA12878 was sequenced to 81X depth. The parents were sequenced to 71X (identifier NA12891) and 70X (NA12892). I used the read set of the child, $\mathcal{R}_c$, as the variant sequences and the union of the read sets from the two parents, $\mathcal{R}_p$, as the control set. For computational convenience, I made calls chromosome-by-chromosome by taking reads mapped to the reference genome and separating them into subsets based on the chromosome the reads mapped to. While this introduces a weak bias towards the reference genome, I believe this drawback is offset by the reduction in run time and memory usage. I used $k = 54$ when making calls and required 5 occurrences of a variant k-mer to trigger assembly. Candidate haplotypes were mapped to the full reference genome.

In Conrad et al.'s original paper, they selected a large number of calls for experimental validation. The selected calls were validated by PCR amplification followed by Illumina sequencing or target enrichment followed by SOLiD sequencing. As the first measure of the performance of my software on calling *de novo* mutations, I compared our calls to the successfully validated mapping calls. The results are presented in table 5.2. In total, the de Bruijn graph caller found 908 of the 936 (97.0%) of the validated subset of calls. The string graph caller found 898 of 936 (95.6%). While these results are encouraging, it is worth noting that the calls selected for validation were filtered to avoid difficult regions of the genome. Mapping-based calls in simple repeats, known copy number variants, segmental duplications or in dbSNP v1.29 were excluded from this validation set. Additionally, sites without read coverage in all three individuals or near a short

[1] Details can be found in [Conrad et al., 2011]

[2] ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/technical/working/20120117_ceu_trio_b37_decoy/

polymorphic indel were removed. In total, these filters excluded 468 Mbp of the genome.

Table 5.2: Chromosome by chromosome breakdown of the number of de novo substitution calls in the validation set and for the de Bruijn (DBG) and string graph (SG) callers.

Chr.	Validated	DBG Calls	Found by DBG (%)	SG Calls	Found by SG (%)
1	51	184	51 (100.0%)	199	50 (98.0%)
2	63	235	63 (100.0%)	267	61 (96.8%)
3	71	201	71 (100.0%)	219	70 (98.6%)
4	100	252	97 (97.0%)	276	99 (99.0%)
5	92	187	92 (100.0%)	213	92 (100.0%)
6	64	176	60 (93.8%)	190	58 (90.6%)
7	64	115	60 (93.8%)	118	59 (92.2%)
8	75	155	73 (97.3%)	165	72 (96.0%)
9	58	128	56 (96.6%)	146	57 (98.3%)
10	47	113	45 (95.7%)	131	45 (95.7%)
11	27	128	26 (96.3%)	144	26 (96.3%)
12	17	89	16 (94.1%)	95	15 (88.2%)
13	31	119	30 (96.8%)	128	30 (96.8%)
14	31	121	29 (93.5%)	129	28 (90.3%)
15	28	94	28 (100.0%)	104	28 (100.0%)
16	23	63	21 (91.3%)	73	18 (78.3%)
17	20	49	20 (100.0%)	64	20 (100.0%)
18	33	80	30 (90.9%)	89	30 (90.9%)
19	8	97	7 (87.5%)	114	7 (87.5%)
20	22	62	22 (100.0%)	69	22 (100.0%)
21	8	24	8 (100.0%)	38	8 (100.0%)
22	3	27	3 (100.0%)	31	3 (100.0%)
total	936	2699	908 (97.0%)	3002	898 (95.9%)

The assembly-based callers found thousands of substitutions that are not present in the validation set. To get a more complete measure of the performance of the assembly caller, I compared the assembly calls to the raw output of

Conrad's caller, DeNovoGears. Here I did not use the same DeNovoGears callset as in [Conrad et al., 2011] but rather I used an updated call set using the most recent version of the program (v0.3) on the recent 101bp sequencing data [1]. I parsed the raw DeNovoGear output to remove SNP calls that had a posterior probability of being a *de novo* mutation less than 0.75. After this filter, 4488 calls remained. 2006 of the 2699 de Bruijn graph SNP calls (74.3%) are found in this set of DeNovoGears calls. For the string graph caller, 2003 of 3002 (66.7%) are found in the DeNovoGear set. This suggests that the majority of the assembly substitution calls are true *de novo* mutations, not false positives.

The de Bruijn graph caller made 2,157 indel calls. After filtering out indels that occur in homopolymers of length 7 or greater, 245 indels remain. The String Graph caller made 2,795 indel calls, 321 of which are not in homopolymer runs. In both call sets the non-homopolymer events are biased towards deletions. In the de Bruijn graph call set the ratio of deletions to insertions is 4.3:1. In the string graph call set the ratio is 3.2:1.

5.8 Cancer mutations

As a second test of our comparative assembly algorithm, I called mutations in a human breast cancer. A typical cancer sequencing experiment sequences a tumor along with the individual's matched normal genome. Variants found only in the tumor are putative *somatic* mutations. In this test, I used a breast cancer sample sequenced at the Sanger Institute as part of the Cancer Genome Project. The tumor read set consists of 1.65 billion 100bp reads (55X). The matched normal genome has 1.32 billion reads (44X). This data set was recently used as part of a large project to catalog mutations [Nik-Zainal et al., 2012a] and mutational history [Nik-Zainal et al., 2012b] in 21 breast cancers. Finding cancer mutations is a more difficult use case than other applications as tumors typically exhibit subclonal structure and some mutations are found only in a subset of tumor cells. In addition the tumor is typically not an entirely pure sample and is contaminated with normal tissue. For this reason, some mutations will be covered by few reads.

[1] These calls are provided by Art Wuster of the Hurles lab

To account for this I used a lower number of required k-mer occurrences to trigger assembly, 3. As in the trio variant calling, I used $k = 54$.

In the framework of my comparative variant caller the reads from the tumor, $\mathcal{R}_T$, are the variant reads, and the reads from the normal, $\mathcal{R}_N$ are the control set. As in the trio experiment I made calls chromosome-by-chromosome. As part of the Cancer Genome Project's standard pipeline they call somatic mutations from the mapped reads using in-house software. In table 5.3 I compare CGP's mapping calls to the assembly calls. Of the $10,381$ calls found by CGP's mapping based caller, the de Bruijn graph caller found 8,035 (77.4%). The string graph caller found $8,593$ (82.8%). There is a noteworthy excess of substitutions on chromosome 6. Closer inspection revealed a dense cluster of C>T transitions on this chromosome. These events occurred primarily in a TpC context (TC>TT substitution). As these C>T events are in close proximity, they often assemble into a single haplotype. Using the string graph calls as an example, the most divergent chromosome 6 haplotype assembled had 35 C>T mutations. Nik-Zainal et al. studied this hypermutation phenomenon in detail in [Nik-Zainal et al., 2012a].

Table 5.3: Chromosome-by-chromosome breakdown of the substitutions called in the breast cancer tumor.

Chr.	Mapping Calls	DBG Calls	Found by DBG (%)	SG Calls	Found by SG (%)
1	783	807	612 (78.2%)	973	647 (82.6%)
2	849	827	660 (77.7%)	946	699 (82.3%)
3	632	590	489 (77.4%)	678	537 (85.0%)
4	645	613	502 (77.8%)	678	536 (83.1%)
5	480	438	366 (76.2%)	482	393 (81.9%)
6	1286	1262	1050 (81.6%)	1313	1082 (84.1%)
7	837	921	661 (79.0%)	1096	717 (85.7%)
8	459	457	352 (76.7%)	519	377 (82.1%)
9	296	309	222 (75.0%)	357	241 (81.4%)
10	519	499	399 (76.9%)	592	422 (81.3%)
11	429	431	329 (76.7%)	491	349 (81.4%)
12	391	419	311 (79.5%)	490	330 (84.4%)
13	247	224	186 (75.3%)	249	204 (82.6%)
14	186	178	143 (76.9%)	215	157 (84.4%)
15	192	185	146 (76.0%)	220	157 (81.8%)
16	260	282	194 (74.6%)	379	211 (81.2%)
17	181	205	113 (62.4%)	241	136 (75.1%)
18	379	396	316 (83.4%)	463	328 (86.5%)
19	128	144	81 (63.3%)	213	93 (72.7%)
20	228	318	176 (77.2%)	387	178 (78.1%)
21	198	197	151 (76.3%)	240	163 (82.3%)
22	95	107	66 (69.5%)	169	76 (80.0%)
X	681	661	510 (74.9%)	701	560 (82.2%)
total	10381	10470	8035 (77.4%)	12092	8593 (82.8%)

The Cancer Genome Project validated 309 of the substitution calls made by their mapping-based caller. Of these 309, 255 were found by the de Bruijn graph caller (82.5%) and 270 (87.4%) were found by the string graph caller.

The de Bruijn graph caller made $4,499$ indel calls, 974 of which are not in a homopolymer run. For the string graph caller $4,510$ indel calls were made, $1,104$ outside of homopolymers. The Cancer Genome Project called indels on

this sample using Pindel [Ye et al., 2009], 333 of which were validated. Of the 333 validated indels, 297 (89.2%) were found by the de Bruijn graph caller and 293 (88.0%) were found by the string graph caller[1]. A number of the assembly indel calls are near a CGP validated indel but did not have the exact same sequence. If I relax the matching criteria to only require the assembly call to be within 20bp of the CGP event, the number of matching events increases to 320 for the de Bruijn graph caller (96.1%) and 319 for the string graph caller (95.8%)[2]. The reasons that the breakpoint sequences differ in these cases remains to be investigated.

Our probabilistic realignment method estimates the allele frequency of variants. In the context of cancer sequencing, this is an estimation of the proportion of chromosomes present in the entire tumor sample (including non-cancerous contaminating tissue) that harbors a particular mutation. Figure 5.6 plots the distribution of allele frequencies for the substitutions called by our string graph method. If mutations occurred on one of the two chromosomes at random and the mutated chromosome was present in all cells of the tumor, we would expect the allele frequencies to be distributed around 0.5. However as cancers continuously accumulate mutations as they evolve, all cells will not contain every mutation. Additionally, contamination by normal tissue will shift the allele frequency distribution towards lower frequency. These effects are shown in figure 5.6 as the median allele frequency is 0.225.

[1] Homopolymer indels were included in this analysis
[2] Calculated with BEDTools' intersectBED program

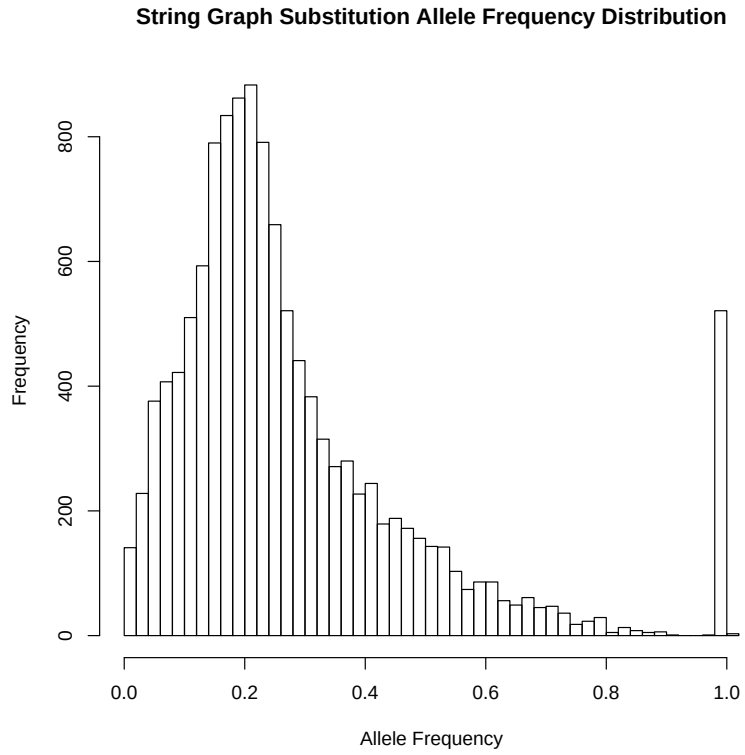


Figure 5.6: The allele frequency distribution for substitution calls made by the string graph caller

To assess the accuracy of our allele frequency estimates, I compared our estimates to those made by CGP's mapping-based pipeline. The allele frequency estimates for variants common to both call sets is plotted in figure 5.7. Our allele frequency estimates are well-correlated to those of the mapping based caller ($r = 0.957$).

The string graph substitution call set contains 447 substitutions with estimated allele frequency 1.0. Few of these calls are also contained in the CGP call set (figure 5.7). Of the 447 high allele frequency substitution calls, 376 are found in dbSNP v1.32. This suggests these calls are probably homozygous SNPs that are incorrectly called as somatic mutations. The likely cause for these false positives is poor sequence coverage of the sites in the matched normal sample.

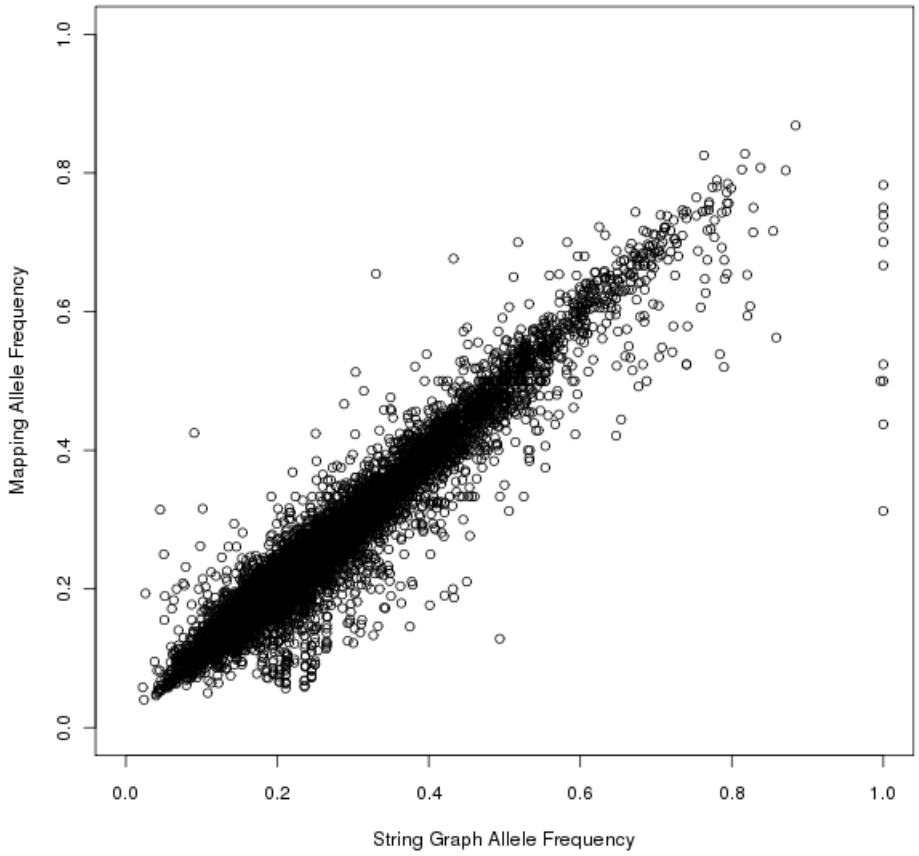


Figure 5.7: The allele frequency calculated by the string graph caller (x-axis) and CGP (y-axis) for calls made a common sites

Finally, I assessed the functional consequences of the string graph calls. Using the Variant Effect Predictor provided by Ensembl [Flicek et al., 2012; McLaren et al., 2010], I predicted the effect of all substitution mutations and all non-homopolymer indels. Ensembl release version 66 was used. My mutation call set had 3 frameshift mutations, including a single base deletion in the important tumor suppressor *TP53*. This variant was also found by the CGP and experimentally validated. Two substitution mutations generated new stop codons. There are 47 non-synonymous coding mutations and 32 synonymous changes.

5.8.1 Analysis Notes

The mapping-based variants, their estimated allele frequencies and validation status were provided by Serena Nik-Zainal of the Cancer Genome Project. The CGP variant calls were made by Caveman, an in-house caller.

5.9 Low-Coverage Population Calls

Finally, I used the assembly-based variant caller on low-coverage human population sequencing. The data used is from Phase 2 of the 1000 Genomes Project [1000 Genomes Project Consortium, 2010]. I used all reads mapping to chromosome 20 for the African continental group (LWK, YRI, ASW, ACB populations). Only individuals that had 75bp reads or greater were included. This subset of the data contains 191 individuals. I used the de Bruijn graph caller for this data set with a k-mer size of 61. Five occurrences of a k-mer were required to trigger assembly and five occurrences of a k-mer were required to use it in the de Bruijn graph (m parameter in `generateDeBruijnHaplotypes`).

The de Bruijn graph caller found 218,852 single nucleotide polymorphisms, 35,846 indels and 2,246 multi-nucleotide polymorphisms[1]. To assess the accuracy of my call set, I calculated the transition/transversion ratio of the SNP variants and the proportion of variants that were previously found. For completely random mutations in random sequence the transition/transversion ratio (Ti/Tv) would be 1:2. In actual sequence however transitions are more likely to occur [Wakeley, 1996]. The transition/transversion ratio of the chromosome 20 calls for African samples in phase 1 of the 1000 Genomes Project is 2.37. The transition/transversion ratio of my call set is 2.20:1. To assess the novelty of my calls, I compared the SNP calls to dbSNP v1.32, which contains calls for the pilot data of the 1000 Genomes Project. 89.68% of my SNP calls are known variants.

In addition to SNPs and MNPs, I called 35,846 indels. To assess the accuracy of my indel calls, I calculated the ratio of in-frame indels (those that do not change the reading frame of protein translation) versus the number of frameshift mutation. As it is expected that frameshift mutations are significantly damaging

[1] Block substitutions of length > 1

to protein function, very few frameshift mutations are expected. My call set contains 14 in-frame and 14 frameshift indels.

Our population caller estimates genotype likelihoods for each sequenced individual and uses these likelihoods to estimate allele frequencies in the population. The allele frequency distribution for the de Bruijn graph SNP and indel calls is presented in figure 5.8. As the assembly based caller requires significant read coverage of each variant sequence to successfully assemble it into a haplotype, we have reduced power to detect low-frequency variants (allele frequency $< 5\%$).

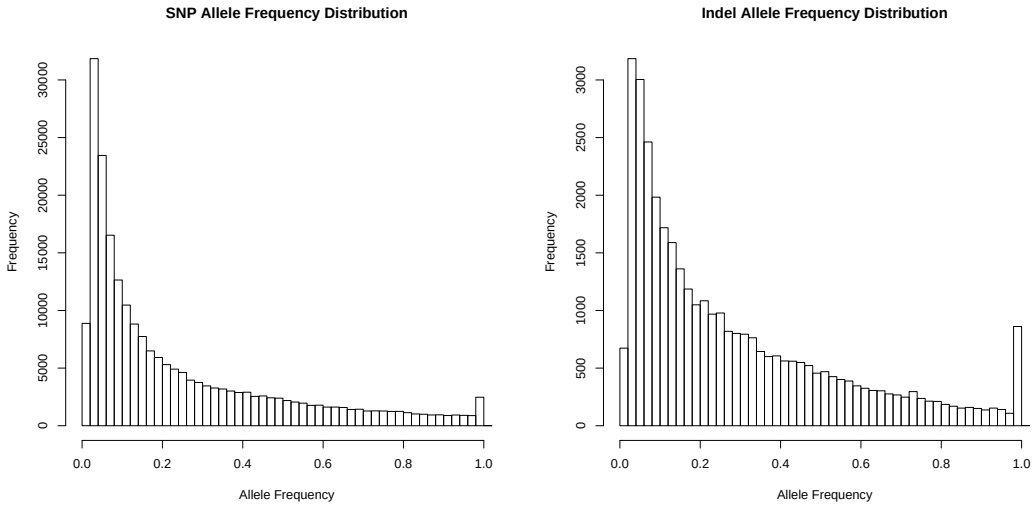


Figure 5.8: The allele frequency distribution for SNP and Indel calls on the AFR continental group of the 1000 Genomes Project

I also compared my indel calls to the mapping-based indel calls from phase 1 of the 1000 Genomes Project[1]. The mapping-based calls were made from 1,094 individuals. I made a subset of the phase 1 calls consisting of calls on chromosome 20 that are not contained in the "excluded" calls file[2]. The mapping-based indel

[1] The calls were downloaded from ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase1/analysis_results/consensus_call_sets/indels/ALL.wgs.VQSR_V2_GLs_polarized.20101123.indels.low_coverage.sites.vcf.gz

[2] ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase1/analysis_results/supporting/excluded_indel_sites/ALL.wgs.excluded_sites_20120312.20101123.indels.sites.vcf.gz

call set contains few calls of length greater than 20bp. Despite using far fewer samples, the assembly call set contains many more large indels, demonstrating the benefit of assembly approaches for finding complex variation (figure 5.9).

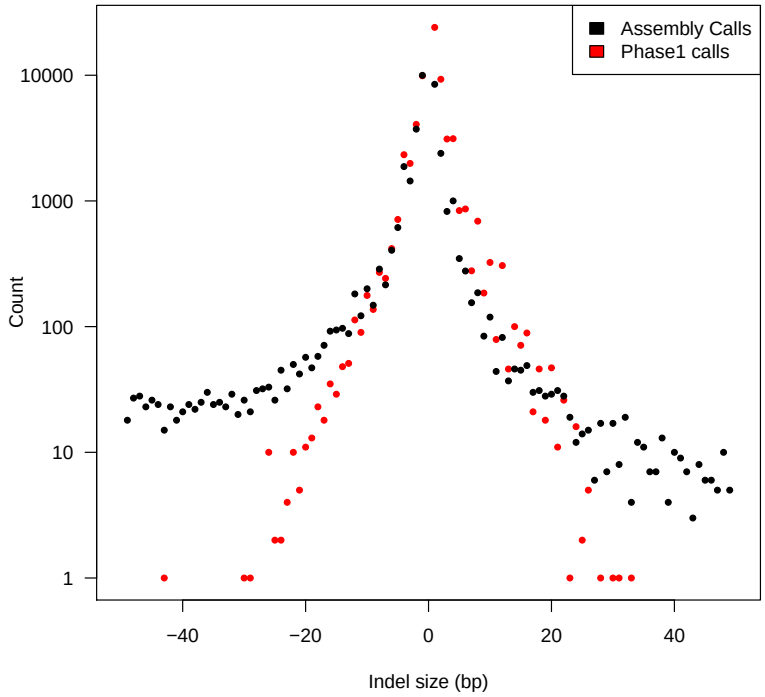


Figure 5.9: The distribution of insertion (positive) and deletion (negative) lengths for the 1000 Genomes data set. The data set consists of 35,846 *assembly indel calls (black points) and* 64,319 *mapping calls from Phase 1 of the 1000 Genomes Project (red points). Events larger than 50bp were excluded from this plot.*

5.9.0.1 Computation Requirements

Constructing the FM-index for the population required 162 CPU hours (59 hours elapsed time). The peak memory usage during index construction was 26GB. Variant calling required 359 CPU hours. Variant calling was run using 16 computation threads, which allowed the task to complete in 29 wall-cloock hours.

The peak memory usage during variant calling was 39GB.

5.10 Discussion

In this chapter, I explored the properties of our assembly-based approach to variant calling. On simulated data, the assembly-based caller recovered the majority of variants while retaining high accuracy. For real data, some power is clearly lost when compared to mapping-based approaches. It is an open question of how many of the "missed" mapping-based variants are true SNPs or indels and how many are false positives. Assembly-based calling requires higher coverage than mapping so it is expected that some true variants will be missed due to insufficient sequence depth. Likewise, we do not yet use read pairs in our haplotype generation functions. This may lead to a loss of power in difficult to assemble regions, which is reflected by the fact that most of the GATK SNPs that we missed in NA12878 are found in annotated repeats. Despite these limitations, the assembly-based approach is promising. The assembly-based caller found most of the validated *de novo* mutations in the trio and validated indels in the cancer sample. The indel size distribution on the 1000 Genomes data suggests the assembly caller has better representation of large events when compared to mapping-based approaches.

Assembly-based variant calling is a new technique. Cortex [Iqbal et al., 2012] and Fermi [Li, 2012] were published this year - the algorithms described in this work were developed in parallel. I did not directly compare to Cortex and Fermi due to the practicalities of running these programs on the range of data sets presented here. A comparison and assessment including Cortex, SGA and state of the art mapping and local reassembly methods is underway for phase 2 of the 1000 Genomes Project. This upcoming assessment should help demonstrate the pros and cons of assembly based approaches.

Chapter 6

Conclusions

In this work, I have developed assembly and variant calling algorithms based on the compressed FM-index data structure. Using the algorithms developed, I performed the first overlap-based assembly of a human genome from Illumina sequence reads [Simpson and Durbin, 2012]. Subsequent to the publication of my assembly method in [Simpson and Durbin, 2010], other groups have followed a similar approach. Dinh and Rajasekaran [2011] developed an efficient data structure for representing an exact-match overlap graph. Gonnella and Kurtz extended my idea of directly outputting only the irreducible edges of a string graph to develop a fast string graph construction algorithm [Gonnella and Kurtz, 2012]. Heng Li reformulated the algorithms in chapter 2 and 3 based on a new representation of the FM-index which stores the read sequences and their reverse complement in the same data structure [Li, 2012].

I have extended upon the *de novo* assembly algorithms to perform comparative variant calling between two genomes. The initial results presented in chapter 5 suggest this is a promising approach for finding relatively complex differences between the pair of genomes. I believe that methods which work directly with sequencing reads, rather than relying on alignments to a reference genome, will become increasing important as sequencing technology improves.

High-throughput short read sequencing profoundly changed genomics. New algorithms needed to be developed to cope with the volumes of data. As sequencing costs continue to fall and more genomes are sequenced there will be constant pressure to lower the computational cost of sequence analysis. One approach that

has recently become prominent is to use probabilistic data structures, such as the bloom filter. This approach has been shown to lower the memory requirements of representing a de Bruijn graph [Chikhi and Rizk, 2012; Pell et al., 2012]. I believe these approaches are complimentary to the FM-index algorithms developed in this work. For example, one could use a bloom filter when performing k-mer based error correction and the FM-index when assembling the corrected reads into contigs. These approaches can easily be implemented within our software framework, which is designed as a modular pipeline.

As the third generation of sequencing technology is developed, which is projected to be based on directly reading the sequence of DNA as it passes through a biological nanopore, the algorithmic landscape will change again. Already read lengths of up to 48kbp have been publicly discussed using nanopore approaches [Schneider and Dekker, 2012]. The algorithmic challenge of indexing the data and constructing an assembly graph will remain, and I believe the FM-index and string graph algorithms presented in this work are well-suited for this task. With >10kb reads, genomic repeats become far less of a barrier to reconstructing the complete sequence of a large genome. I believe the core algorithmic challenge in assembly will not be to simply reconstruct the full sequence of a genome but to reconstruct the full haplotype-resolved *phased* genome of diploid organisms. The logical starting point of all sequence analysis should be the complete genetic content of a cell, and I believe this goal is not far off.

References

1000 Genomes Project Consortium. A map of human genome variation from population-scale sequencing. *Nature*, 467(7319):1061–1073, 2010. ISSN 0028-0836. doi:10.1038/nature09534. URL http://dx.doi.org/10.1038/nature09534. 5, 15, 115

Abouelhoda, Mohamed I., Kurtz, Stefan, and Ohlebusch, Enno. The enhanced suffix array and its applications to genome analysis algorithms in bioinformatics. In Guigó, Roderic and Gusfield, Dan, editors, *Algorithms in Bioinformatics*, volume 2452 of *Lecture Notes in Computer Science*, chapter 35, pages 449–463. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-540-44211-0. doi:10.1007/3-540-45784-4_35. URL http://dx.doi.org/10.1007/3-540-45784-4_35. 16

Abouelhoda, Mohamed I., Kurtz, Stefan, and Ohlebusch, Enno. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004. ISSN 15708667. doi:10.1016/S1570-8667(03)00065-0. URL http://dx.doi.org/10.1016/S1570-8667(03)00065-0. 22

Adams, Mark D., Sutton, Granger G., Smith, Hamilton O., Myers, Eugene W., and Venter, J. Craig. The independence of our genome assemblies. *Proceedings of the National Academy of Sciences*, 100(6):3025–3026, 2003. ISSN 1091-6490. doi:10.1073/pnas.0637478100. URL http://dx.doi.org/10.1073/pnas.0637478100. 4

Albers, Cornelis A., Lunter, Gerton, MacArthur, Daniel G., McVean, Gilean, Ouwehand, Willem H., and Durbin, Richard. Dindel: Accurate indel calls from

short-read data. *Genome Research*, 21(6):961–973, 2011. ISSN 1549-5469. doi: 10.1101/gr.112326.110. URL http://dx.doi.org/10.1101/gr.112326.110. 88, 105

Arabidopsis Genome Initiative. Analysis of the genome sequence of the flowering plant arabidopsis thaliana. *Nature*, 408(6814):796–815, 2000. ISSN 0028-0836. doi:10.1038/35048692. URL http://dx.doi.org/10.1038/35048692. 3

Barnett, Derek W., Garrison, Erik K., Quinlan, Aaron R., Strömberg, Michael P., and Marth, Gabor T. BamTools: a c++ API and toolkit for analyzing and managing BAM files. *Bioinformatics*, 27(12):1691–1692, 2011. ISSN 1460-2059. doi:10.1093/bioinformatics/btr174. URL http://dx.doi.org/10.1093/bioinformatics/btr174. 58

Bauer, Markus J., Cox, Anthony J., and Rosone, Giovanna. Lightweight BWT construction for very large string collections combinatorial pattern matching. volume 6661 of *Lecture Notes in Computer Science*, chapter 20, pages 219–231. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-21457-8. doi:10.1007/978-3-642-21458-5_20. URL http://dx.doi.org/10.1007/978-3-642-21458-5_20. 32, 46

Bentley, David R., Balasubramanian, Shankar, Swerdlow, Harold P., Smith, Geoffrey P., Milton, John, Brown, Clive G., Hall, Kevin P., Evers, Dirk J., Barnes, Colin L., Bignell, Helen R., et al. Accurate whole human genome sequencing using reversible terminator chemistry. *Nature*, 456(7218):53–59, 2008. ISSN 1476-4687. doi:10.1038/nature07517. URL http://dx.doi.org/10.1038/nature07517. 5, 47

Bentley, Jon L. and Sedgewick, Robert. Fast algorithms for sorting and searching strings. In *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 360–369. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997. ISBN 0-89871-390-0. URL http://portal.acm.org/citation.cfm?id=314321. 32

Berger, Emery D., McKinley, Kathryn S., Blumofe, Robert D., and Wilson, Paul R. Hoard: a scalable memory allocator for multithreaded ap-

plications. *SIGPLAN Not.*, 35(11):117–128, 2000. ISSN 0362-1340. doi: 10.1145/356989.357000. URL http://dx.doi.org/10.1145/356989.357000. 58

Blattner, F. R., Plunkett, G., Bloch, C. A., Perna, N. T., Burland, V., Riley, M., Collado-Vides, J., Glasner, J. D., Rode, C. K., Mayhew, G. F., et al. The complete genome sequence of escherichia coli k-12. *Science (New York, N.Y.)*, 277(5331):1453–1462, 1997. ISSN 0036-8075. doi:10.1126/science.277.5331.1453. URL http://dx.doi.org/10.1126/science.277.5331.1453. 3

Burrows, M. and Wheeler, D. J. A block-sorting lossless data compression algorithm. Technical Report 124, 1994. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.6774. 16, 28

Butler, Jonathan, MacCallum, Iain, Kleber, Michael, Shlyakhter, Ilya A., Belmonte, Matthew K., Lander, Eric S., Nusbaum, Chad, and Jaffe, David B. ALLPATHS: de novo assembly of whole-genome shotgun microreads. *Genome research*, 18(5):810–820, 2008. ISSN 1088-9051. doi:10.1101/gr.7337908. URL http://dx.doi.org/10.1101/gr.7337908. 13

C. elegans Sequencing Consortium. Genome sequence of the nematode c. elegans: a platform for investigating biology. *Science (New York, N.Y.)*, 282(5396):2012–2018, 1998. ISSN 0036-8075. doi:10.1126/science.282.5396.2012. URL http://dx.doi.org/10.1126/science.282.5396.2012. 3, 61

Catchen, Julian M., Amores, Angel, Hohenlohe, Paul, Cresko, William, and Postlethwait, John H. Stacks: Building and genotyping loci de novo from Short-Read sequences. *G3: Genes, Genomes, Genetics*, 1(3):171–182, 2011. ISSN 2160-1836. doi:10.1534/g3.111.000240. URL http://dx.doi.org/10.1534/g3.111.000240. 74

Chaisson, Mark J. and Pevzner, Pavel A. [duplicate] short read fragment assembly of bacterial genomes. *Genome Research*, 18(2):324–330, 2008. ISSN 1549-5469. doi:10.1101/gr.7088808. URL http://dx.doi.org/10.1101/gr.7088808. 8, 54

Chikhi, Rayan and Rizk, Guillaume. Space-Efficient and exact de bruijn graph representation based on a bloom filter algorithms in bioinformatics. volume 7534 of *Lecture Notes in Computer Science*, chapter 19, pages 236–248. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-33121-3. doi:10.1007/978-3-642-33122-0_19. URL http://dx.doi.org/10.1007/978-3-642-33122-0_19. 8, 120

Chimpanzee Sequencing and Analysis Consortium. Initial sequence of the chimpanzee genome and comparison with the human genome. *Nature*, 437(7055):69–87, 2005. ISSN 1476-4687. doi:10.1038/nature04072. URL http://dx.doi.org/10.1038/nature04072. 4

Conrad, Donald F., Keebler, Jonathan E., DePristo, Mark A., Lindsay, Sarah J., Zhang, Yujun, Casals, Ferran, Idaghdour, Youssef, Hartl, Chris L., Torroja, Carlos, Garimella, Kiran V., et al. Variation in genome-wide mutation rates within and between human families. *Nature genetics*, 43(7):712–714, 2011. ISSN 1546-1718. doi:10.1038/ng.862. URL http://dx.doi.org/10.1038/ng.862. 102, 106, 107, 109

Conway, Thomas, Wazny, Jeremy, Bromage, Andrew, Zobel, Justin, and Beresford-Smith, Bryan. Gossamer a resource-efficient de novo assembler. *Bioinformatics*, 28(14):1937–1938, 2012. ISSN 1460-2059. doi:10.1093/bioinformatics/bts297. URL http://dx.doi.org/10.1093/bioinformatics/bts297. 14

Conway, Thomas C. and Bromage, Andrew J. Succinct data structures for assembling large genomes. *Bioinformatics (Oxford, England)*, 27(4):479–486, 2011. ISSN 1367-4811. doi:10.1093/bioinformatics/btq697. URL http://dx.doi.org/10.1093/bioinformatics/btq697. 8, 14

de Bruijn, N. G. A combinatorial problem. *Koninklijke Nederlandse Akademie v. Wetenschappen*, 49:758–764, 1946. 8

Dempster, A. P., Laird, N. M., and Rubin, D. B. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society.*

Series B (Methodological), 39(1):1–38, 1977. ISSN 00359246. doi:10.2307/2984875. URL http://dx.doi.org/10.2307/2984875. 90

DePristo, Mark A., Banks, Eric, Poplin, Ryan, Garimella, Kiran V., Maguire, Jared R., Hartl, Christopher, Philippakis, Anthony A., del Angel, Guillermo, Rivas, Manuel A., Hanna, Matt, et al. A framework for variation discovery and genotyping using next-generation DNA sequencing data. *Nat Genet*, 43(5):491–498, 2011. ISSN 1546-1718. doi:10.1038/ng.806. URL http://dx.doi.org/10.1038/ng.806. 68, 97, 102

Dinh, Hieu and Rajasekaran, Sanguthevar. A memory-efficient data structure representing exact-match overlap graphs with application for next-generation DNA assembly. *Bioinformatics*, 27(14):1901–1907, 2011. ISSN 1460-2059. doi:10.1093/bioinformatics/btr321. URL http://dx.doi.org/10.1093/bioinformatics/btr321. 119

Drmanac, Radoje, Sparks, Andrew B., Callow, Matthew J., Halpern, Aaron L., Burns, Norman L., Kermani, Bahram G., Carnevali, Paolo, Nazarenko, Igor, Nilsen, Geoffrey B., Yeung, George, et al. Human genome sequencing using unchained base reads on self-assembling DNA nanoarrays. *Science (New York, N.Y.)*, 327(5961):78–81, 2010. ISSN 1095-9203. doi:10.1126/science.1181498. URL http://dx.doi.org/10.1126/science.1181498. 5

Earl, Dent, Bradnam, Keith, St. John, John, Darling, Aaron, Lin, Dawei, Fass, Joseph, Yu, Hung On Ken, Buffalo, Vince, Zerbino, Daniel R., Diekhans, Mark, et al. Assemblathon 1: A competitive assessment of de novo short read assembly methods. *Genome Research*, 21(12):2224–2241, 2011. ISSN 1549-5469. doi:10.1101/gr.126599.111. URL http://dx.doi.org/10.1101/gr.126599.111. 10, 14, 70, 71

Eid, John, Fehr, Adrian, Gray, Jeremy, Luong, Khai, Lyle, John, Otto, Geoff, Peluso, Paul, Rank, David, Baybayan, Primo, Bettman, Brad, et al. Real-Time DNA sequencing from single polymerase molecules. *Science*, 323(5910):133–138, 2009. ISSN 1095-9203. doi:10.1126/science.1162986. URL http://dx.doi.org/10.1126/science.1162986. 5

Euler, Leonard. Solutio problematis ad geometriam situs pertinentis. *Commentarii academiae scientiarum Petropolitanae*, 8:128–140, 1741. URL http://www.math.dartmouth.edu/~{}euler/pages/E053.html. 23

Ferragina, P. and Manzini, G. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, volume 0, pages 390–398. IEEE Comput. Soc, Los Alamitos, CA, USA, 2000. ISBN 0-7695-0850-2. ISSN 0272-5428. doi:10.1109/SFCS.2000.892127. URL http://dx.doi.org/10.1109/SFCS.2000.892127. 16, 28

Ferragina, Paolo, Gagie, Travis, and Manzini, Giovanni. Lightweight data indexing and compression in external memory. In *Proceedings of the Latin American Theoretical Informatics Symposium*. 2010. URL http://arxiv.org/abs/0909.4341. 46

Fiers, W., Contreras, R., Duerinck, F., Haegeman, G., Iserentant, D., Merregaert, J., Min Jou, W., Molemans, F., Raeymaekers, A., Van den Berghe, A., et al. Complete nucleotide sequence of bacteriophage MS2 RNA: primary and secondary structure of the replicase gene. *Nature*, 260(5551):500–507, 1976. ISSN 0028-0836. URL http://view.ncbi.nlm.nih.gov/pubmed/1264203. 2

Fleischmann, R. D., Adams, M. D., White, O., Clayton, R. A., Kirkness, E. F., Kerlavage, A. R., Bult, C. J., Tomb, J. F., Dougherty, B. A., Merrick, J. M., et al. Whole-genome random sequencing and assembly of haemophilus influenzae rd. *Science*, 269(5223):496–512, 1995. ISSN 1095-9203. doi:10.1126/science.7542800. URL http://dx.doi.org/10.1126/science.7542800. 3

Fleury, M. Deux problemes de geometrie de situation. *Journal de mathematiques elementaires*, pages 257–261, 1883. 24

Flicek, Paul, Amode, M. Ridwan, Barrell, Daniel, Beal, Kathryn, Brent, Simon, Carvalho-Silva, Denise, Clapham, Peter, Coates, Guy, Fairley, Susan, Fitzgerald, Stephen, et al. Ensembl 2012. *Nucleic Acids Research*, 40(D1):D84–D90, 2012. ISSN 1362-4962. doi:10.1093/nar/gkr991. URL http://dx.doi.org/10.1093/nar/gkr991. 114

Genome 10K Community of Scientists. Genome 10K: A proposal to obtain Whole-Genome sequence for 10000 vertebrate species. *Journal of Heredity*, 100(6):659–674, 2009. ISSN 1465-7333. doi:10.1093/jhered/esp086. URL http://dx.doi.org/10.1093/jhered/esp086. 5

Girard, Simon L., Gauthier, Julie, Noreau, Anne, Xiong, Lan, Zhou, Sirui, Jouan, Loubna, Dionne-Laporte, Alexandre, Spiegelman, Dan, Henrion, Edouard, Diallo, Ousmane, et al. Increased exonic de novo mutation rate in individuals with schizophrenia. *Nat Genet*, 43(9):860–863, 2011. ISSN 1061-4036. doi:10.1038/ng.886. URL http://dx.doi.org/10.1038/ng.886. 106

Gnerre, Sante, MacCallum, Iain, Przybylski, Dariusz, Ribeiro, Filipe J., Burton, Joshua N., Walker, Bruce J., Sharpe, Ted, Hall, Giles, Shea, Terrance P., Sykes, Sean, et al. [duplicate] high-quality draft assemblies of mammalian genomes from massively parallel sequence data. *Proceedings of the National Academy of Sciences*, 108(4):1513–1518, 2011. ISSN 1091-6490. doi:10.1073/pnas.1017351108. URL http://dx.doi.org/10.1073/pnas.1017351108. 13, 14, 71

Goffeau, A., Barrell, B. G., Bussey, H., Davis, R. W., Dujon, B., Feldmann, H., Galibert, F., Hoheisel, J. D., Jacq, C., Johnston, M., et al. Life with 6000 genes. *Science*, 274(5287):546–567, 1996. ISSN 1095-9203. doi:10.1126/science.274.5287.546. URL http://dx.doi.org/10.1126/science.274.5287.546. 3

Gonnella, Giorgio and Kurtz, Stefan. Readjoiner: a fast and memory efficient string graph-based sequence assembler. *BMC Bioinformatics*, 13(1):82+, 2012. ISSN 1471-2105. doi:10.1186/1471-2105-13-82. URL http://dx.doi.org/10.1186/1471-2105-13-82. 119

Green, Phil. Whole-genome disassembly. *Proceedings of the National Academy of Sciences*, 99(7):4143–4144, 2002. ISSN 1091-6490. doi:10.1073/pnas.082095999. URL http://dx.doi.org/10.1073/pnas.082095999. 4

Grossi, Roberto and Vitter, Jeffrey S. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In

Proceedings of the thirty-second annual ACM symposium on Theory of computing, STOC '00, pages 397–406. ACM, New York, NY, USA, 2000. ISBN 1-58113-184-4. doi:10.1145/335305.335351. URL http://dx.doi.org/10.1145/335305.335351. 17

Gusfield, Dan. *Algorithms on strings, trees, and sequences : computer science and computational biology*. Cambridge Univ. Press, 1997. ISBN 0521585198. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/0521585198. 16, 22

Healy, John, Thomas, Elizabeth E., Schwartz, Jacob T., and Wigler, Michael. Annotating large genomes with exact word matches. *Genome research*, 13(10):2306–2315, 2003. ISSN 1088-9051. doi:10.1101/gr.1350803. URL http://dx.doi.org/10.1101/gr.1350803. 16, 57

Idury, R. M. and Waterman, M. S. A new algorithm for DNA sequence assembly. *Journal of computational biology*, 2(2):291–306, 1995. ISSN 1066-5277. URL http://view.ncbi.nlm.nih.gov/pubmed/7497130. 8

International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001. ISSN 0028-0836. doi:10.1038/35057062. URL http://dx.doi.org/10.1038/35057062. 4

Iqbal, Zamin, Caccamo, Mario, Turner, Isaac, Flicek, Paul, and McVean, Gil. De novo assembly and genotyping of variants using colored de bruijn graphs. *Nature genetics*, 44(2):226–232, 2012. ISSN 1546-1718. doi:10.1038/ng.1028. URL http://dx.doi.org/10.1038/ng.1028. 14, 15, 74, 75, 93, 118

Jou, W. M., Haegeman, G., Ysebaert, M., and Fiers, W. Nucleotide sequence of the gene coding for the bacteriophage MS2 coat protein. *Nature*, 237:82–88, 1972. doi:10.1038/237082a0. URL http://dx.doi.org/10.1038/237082a0. 2

Kececioglu, John D. and Myers, Eugene W. Combinatorial algorithms for DNA sequence assembly. In *Algorithmica*, volume 13, pages 7–51. 1993. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.44.5469. 7

Kelley, David R., Schatz, Michael C., and Salzberg, Steven L. Quake: quality-aware detection and correction of sequencing errors. *Genome biology*, 11(11):R116+, 2010. ISSN 1465-6914. doi:10.1186/gb-2010-11-11-r116. URL http://dx.doi.org/10.1186/gb-2010-11-11-r116. 47, 48

Kent, W. James. BLATthe BLAST-like alignment tool. *Genome Research*, 12(4):656–664, 2002. ISSN 1549-5469. doi:10.1101/gr.229202.\%20Article\%20published\%20online\%20before\%20March\%202002. URL http://dx.doi.org/10.1101/gr.229202.%20Article%20published%20online%20before%20March%202002. 16

Ko, P. and Aluru, S. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2-4):143–156, 2005. ISSN 15708667. doi:10.1016/j.jda.2004.08.002. URL http://dx.doi.org/10.1016/j.jda.2004.08.002. 32

Kurtz, Stefan, Narechania, Apurva, Stein, Joshua C., and Ware, Doreen. A new method to compute k-mer frequencies and its application to annotate large repetitive plant genomes. *BMC genomics*, 9(1):517+, 2008. ISSN 1471-2164. doi:10.1186/1471-2164-9-517. URL http://dx.doi.org/10.1186/1471-2164-9-517. 16

Lam, T. W., Li, Ruiqiang, Tam, Alan, Wong, Simon, Wu, Edward, and Yiu, S. M. High throughput short read alignment via bi-directional BWT. In *2009 IEEE International Conference on Bioinformatics and Biomedicine*, volume 0, pages 31–36. IEEE, Los Alamitos, CA, USA, 2009. ISBN 978-0-7695-3885-3. doi:10.1109/BIBM.2009.42. URL http://dx.doi.org/10.1109/BIBM.2009.42. 36

Lam, T. W., Sung, W. K., Tam, S. L., Wong, C. K., and Yiu, S. M. Compressed indexing and local alignment of DNA. *Bioinformatics*, 24(6):791–797, 2008. ISSN 1460-2059. doi:10.1093/bioinformatics/btn032. URL http://dx.doi.org/10.1093/bioinformatics/btn032. 16

Langmead, Ben, Trapnell, Cole, Pop, Mihai, and Salzberg, Steven. Ultrafast and memory-efficient alignment of short DNA sequences to the human

genome. *Genome Biology*, 10(3):R25–10, 2009. ISSN 1465-6906. doi:10.1186/gb-2009-10-3-r25. URL http://dx.doi.org/10.1186/gb-2009-10-3-r25. 16

Li, Fugen and Stormo, Gary D. Selection of optimal DNA oligos for gene expression arrays. *Bioinformatics*, 17(11):1067–1076, 2001. ISSN 1460-2059. doi:10.1093/bioinformatics/17.11.1067. URL http://dx.doi.org/10.1093/bioinformatics/17.11.1067. 16

Li, Heng. A statistical framework for SNP calling, mutation discovery, association mapping and population genetical parameter estimation from sequencing data. *Bioinformatics (Oxford, England)*, 27(21):2987–2993, 2011. ISSN 1367-4811. doi:10.1093/bioinformatics/btr509. URL http://dx.doi.org/10.1093/bioinformatics/btr509. 15

Li, Heng. Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. *Bioinformatics*, 28(14):1838–1844, 2012. ISSN 1460-2059. doi:10.1093/bioinformatics/bts280. URL http://dx.doi.org/10.1093/bioinformatics/bts280. 14, 15, 93, 105, 118, 119

Li, Heng and Durbin, Richard. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics (Oxford, England)*, 25(14):1754–1760, 2009. ISSN 1367-4811. doi:10.1093/bioinformatics/btp324. URL http://dx.doi.org/10.1093/bioinformatics/btp324. 16, 55, 66

Li, Heng and Durbin, Richard. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics (Oxford, England)*, 26(5):589–595, 2010. ISSN 1367-4811. doi:10.1093/bioinformatics/btp698. URL http://dx.doi.org/10.1093/bioinformatics/btp698. 62, 68

Li, Heng and Homer, Nils. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 11(5):473–483, 2010. ISSN 1477-4054. doi:10.1093/bib/bbq015. URL http://dx.doi.org/10.1093/bib/bbq015. 15, 74

Li, Ruiqiang, Fan, Wei, Tian, Geng, Zhu, Hongmei, He, Lin, Cai, Jing, Huang, Quanfei, Cai, Qingle, Li, Bo, Bai, Yinqi, et al. The sequence and de novo

assembly of the giant panda genome. *Nature*, 463(7279):311–317, 2010a. ISSN 1476-4687. doi:10.1038/nature08696. URL http://dx.doi.org/10.1038/nature08696. 4, 13

Li, Ruiqiang, Li, Yingrui, Zheng, Hancheng, Luo, Ruibang, Zhu, Hongmei, Li, Qibin, Qian, Wubin, Ren, Yuanyuan, Tian, Geng, Li, Jinxiang, et al. Building the sequence map of the human pan-genome. *Nat Biotech*, 28(1):57–63, 2010b. ISSN 1546-1696. doi:10.1038/nbt.1596. URL http://dx.doi.org/10.1038/nbt.1596. 61

Li, Ruiqiang, Yu, Chang, Li, Yingrui, Lam, Tak-Wah, Yiu, Siu-Ming, Kristiansen, Karsten, and Wang, Jun. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009. ISSN 1460-2059. doi:10.1093/bioinformatics/btp336. URL http://dx.doi.org/10.1093/bioinformatics/btp336. 16

Li, Ruiqiang, Zhu, Hongmei, Ruan, Jue, Qian, Wubin, Fang, Xiaodong, Shi, Zhongbin, Li, Yingrui, Li, Shengting, Shan, Gao, Kristiansen, Karsten, et al. [duplicate] de novo assembly of human genomes with massively parallel short read sequencing. *Genome Research*, 20(2):265–272, 2010c. ISSN 1549-5469. doi:10.1101/gr.097261.109. URL http://dx.doi.org/10.1101/gr.097261.109. 8, 13, 47, 48, 54, 61, 67, 71

Maccallum, Iain, Przybylski, Dariusz, Gnerre, Sante, Burton, Joshua, Shlyakhter, Ilya, Gnirke, Andreas, Malek, Joel, McKernan, Kevin, Ranade, Swati, Shea, Terrance P., et al. ALLPATHS 2: small genomes assembled accurately and with high continuity from short paired reads. *Genome biology*, 10(10):R103+, 2009. ISSN 1465-6914. doi:10.1186/gb-2009-10-10-r103. URL http://dx.doi.org/10.1186/gb-2009-10-10-r103. 13

Malde, Ketil, Coward, Eivind, and Jonassen, Inge. Fast sequence clustering using a suffix array algorithm. *Bioinformatics*, 19(10):1221–1226, 2003. ISSN 1460-2059. doi:10.1093/bioinformatics/btg138. URL http://dx.doi.org/10.1093/bioinformatics/btg138. 16

Manber, Udi and Myers, Gene. Suffix arrays: a new method for on-line string searches. In *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 319–327. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1990. ISBN 0-89871-251-3. URL http://portal.acm.org/citation.cfm?id=320176.320218. 16, 28

Maxam, A. M. and Gilbert, W. A new method for sequencing DNA. *Proceedings of the National Academy of Sciences of the United States of America*, 74(2):560–564, 1977. ISSN 0027-8424. URL http://www.ncbi.nlm.nih.gov/pmc/articles/PMC392330/. 2

McLaren, William, Pritchard, Bethan, Rios, Daniel, Chen, Yuan, Flicek, Paul, and Cunningham, Fiona. Deriving the consequences of genomic variants with the ensembl API and SNP effect predictor. *Bioinformatics (Oxford, England)*, 26(16):2069–2070, 2010. ISSN 1367-4811. doi:10.1093/bioinformatics/btq330. URL http://dx.doi.org/10.1093/bioinformatics/btq330. 114

Meacham, Frazer, Boffelli, Dario, Dhahbi, Joseph, Martin, David, Singer, Meromit, and Pachter, Lior. Identification and correction of systematic error in high-throughput sequence data. *BMC Bioinformatics*, 12(1):451+, 2011. ISSN 1471-2105. doi:10.1186/1471-2105-12-451. URL http://dx.doi.org/10.1186/1471-2105-12-451. 79

Mouse Genome Sequencing Consortium. Initial sequencing and comparative analysis of the mouse genome. *Nature*, 420(6915):520–562, 2002. ISSN 0028-0836. doi:10.1038/nature01262. URL http://dx.doi.org/10.1038/nature01262. 4

Myers, Eugene W. The fragment assembly string graph. *Bioinformatics*, 21(suppl 2):ii79–ii85, 2005. ISSN 1460-2059. doi:10.1093/bioinformatics/bti1114. URL http://dx.doi.org/10.1093/bioinformatics/bti1114. 7, 22, 24, 28, 44, 55, 83

Myers, Eugene W., Sutton, Granger G., Smith, Hamilton O., Adams, Mark D., and Venter, J. Craig. On the sequencing and assembly of the human genome. *Proceedings of the National Academy of Sciences*, 99(7):4145–4146, 2002. ISSN

1091-6490. doi:10.1073/pnas.092136699. URL http://dx.doi.org/10.1073/pnas.092136699. 4

Nagarajan, Niranjan and Pop, Mihai. Parametric complexity of sequence assembly: theory and applications to next generation sequencing. *Journal of computational biology : a journal of computational molecular cell biology*, 16(7):897–908, 2009. ISSN 1557-8666. doi:10.1089/cmb.2009.0005. URL http://dx.doi.org/10.1089/cmb.2009.0005. 24

Nik-Zainal, Serena, Alexandrov, Ludmil B., Wedge, David C., Van Loo, Peter, Greenman, Christopher D., Raine, Keiran, Jones, David, Hinton, Jonathan, Marshall, John, Stebbings, Lucy A., et al. Mutational processes molding the genomes of 21 breast cancers. *Cell*, 149(5):979–993, 2012a. ISSN 00928674. doi:10.1016/j.cell.2012.04.024. URL http://dx.doi.org/10.1016/j.cell.2012.04.024. 109, 110

Nik-Zainal, Serena, Van Loo, Peter, Wedge, David C., Alexandrov, Ludmil B., Greenman, Christopher D., Lau, King Wai W., Raine, Keiran, Jones, David, Marshall, John, Ramakrishna, Manasa, et al. The life history of 21 breast cancers. *Cell*, 149(5):994–1007, 2012b. ISSN 1097-4172. doi:10.1016/j.cell.2012.04.023. URL http://dx.doi.org/10.1016/j.cell.2012.04.023. 109

Ning, Z., Cox, A. J., and Mullikin, J. C. SSAHA: a fast search method for large DNA databases. *Genome research*, 11(10):1725–1729, 2001. ISSN 1088-9051. doi:10.1101/gr.194201. URL http://dx.doi.org/10.1101/gr.194201. 16

Nong, Ge, Zhang, Sen, and Chan, Wai H. Linear suffix array construction by almost pure Induced-Sorting. *Data Compression Conference*, 0:193–202, 2009. ISSN 1068-0314. doi:10.1109/DCC.2009.42. URL http://dx.doi.org/10.1109/DCC.2009.42. 32, 46

Pell, Jason, Hintze, Arend, Canino-Koning, Rosangela, Howe, Adina, Tiedje, James M., and Brown, C. Titus. Scaling metagenome sequence assembly with probabilistic de bruijn graphs. *Proceedings of the National Academy of Sciences*, 109(33):13272–13277, 2012. ISSN 1091-6490. doi:10.1073/pnas.1121464109. URL http://dx.doi.org/10.1073/pnas.1121464109. 8, 120

Pevzner, P. A. 1-Tuple DNA sequencing: computer analysis. *Journal of biomolecular structure & dynamics*, 7(1):63–73, 1989. ISSN 0739-1102. URL http://view.ncbi.nlm.nih.gov/pubmed/2684223. 8

Pevzner, Pavel A., Tang, Haixu, and Waterman, Michael S. An eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001. ISSN 1091-6490. doi:10.1073/pnas.171285098. URL http://dx.doi.org/10.1073/pnas.171285098. 8, 23, 47, 48, 78

Pop, Mihai, Kosack, Daniel S., and Salzberg, Steven L. Hierarchical scaffolding with bambus. *Genome research*, 14(1):149–159, 2004. ISSN 1088-9051. doi:10.1101/gr.1536204. URL http://dx.doi.org/10.1101/gr.1536204. 55

Prufer, Kay, Munch, Kasper, Hellmann, Ines, Akagi, Keiko, Miller, Jason R., Walenz, Brian, Koren, Sergey, Sutton, Granger, Kodira, Chinnappa, Winer, Roger, et al. The bonobo genome compared with the chimpanzee and human genomes. *Nature*, 486(7404):527–531, 2012. ISSN 0028-0836. doi:10.1038/nature11128. URL http://dx.doi.org/10.1038/nature11128. 4

Puglisi, Simon J., Smyth, W. F., and Turpin, Andrew H. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2):4+, 2007. ISSN 0360-0300. doi:10.1145/1242471.1242472. URL http://dx.doi.org/10.1145/1242471.1242472. 31

Rasmussen, Kim R., Stoye, Jens, and Myers, Eugene W. Efficient q-gram filters for finding all epsilon-matches over a given length. *Journal of computational biology : a journal of computational molecular cell biology*, 13(2):296–308, 2006. ISSN 1066-5277. doi:10.1089/cmb.2006.13.296. URL http://dx.doi.org/10.1089/cmb.2006.13.296. 7, 22

Sanders, Stephan J., Murtha, Michael T., Gupta, Abha R., Murdoch, John D., Raubeson, Melanie J., Willsey, A. Jeremy, Ercan-Sencicek, A. Gulhan, DiLullo, Nicholas M., Parikshak, Neelroop N., Stein, Jason L., et al. De novo mutations revealed by whole-exome sequencing are strongly associated with autism. *Nature*, 485(7397):237–241, 2012. ISSN 1476-4687. doi:10.1038/nature10945. URL http://dx.doi.org/10.1038/nature10945. 106

Sanger, F., Brownlee, G. G., and Barrell, B. G. A two-dimensional fractionation procedure for radioactive nucleotides. *Journal of Molecular Biology*, 14(1):303+, 1965. ISSN 00222836. doi:10.1016/S0022-2836(65)80253-4. URL http://dx.doi.org/10.1016/S0022-2836(65)80253-4. 2

Sanger, F., Nicklen, S., and Coulson, A. R. DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences*, 74(12):5463–5467, 1977. ISSN 1091-6490. doi:10.1073/pnas.74.12.5463. URL http://dx.doi.org/10.1073/pnas.74.12.5463. 3

Scally, Aylwyn, Dutheil, Julien Y., Hillier, LaDeana W., Jordan, Gregory E., Goodhead, Ian, Herrero, Javier, Hobolth, Asger, Lappalainen, Tuuli, Mailund, Thomas, Marques-Bonet, Tomas, et al. Insights into hominid evolution from the gorilla genome sequence. *Nature*, 483(7388):169–175, 2012. ISSN 1476-4687. doi:10.1038/nature10842. URL http://dx.doi.org/10.1038/nature10842. 4, 13

Schmid, C. W. and Deininger, P. L. Sequence organization of the human genome. *Cell*, 6(3):345–358, 1975. ISSN 0092-8674. URL http://view.ncbi.nlm.nih.gov/pubmed/1052772. 3

Schneider, Gregory F. and Dekker, Cees. DNA sequencing with nanopores. *Nat Biotech*, 30(4):326–328, 2012. ISSN 1087-0156. doi:10.1038/nbt.2181. URL http://dx.doi.org/10.1038/nbt.2181. 120

Simpson, Jared T. and Durbin, Richard. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, 26(12):i367–i373, 2010. ISSN 1460-2059. doi:10.1093/bioinformatics/btq217. URL http://dx.doi.org/10.1093/bioinformatics/btq217. ii, 19, 59, 119

Simpson, Jared T. and Durbin, Richard. [duplicate] efficient de novo assembly of large genomes using compressed data structures. *Genome research*, 22(3):549–556, 2012. ISSN 1549-5469. doi:10.1101/gr.126953.111. URL http://dx.doi.org/10.1101/gr.126953.111. ii, 44, 102, 119

Simpson, Jared T., Wong, Kim, Jackman, Shaun D., Schein, Jacqueline E., Jones, Steven J. M., and Birol, İnanç. ABySS: A parallel assembler for short read

sequence data. *Genome Research*, 19(6):1117–1123, 2009. ISSN 1549-5469. doi: 10.1101/gr.089532.108. URL http://dx.doi.org/10.1101/gr.089532.108. 8, 13, 54, 55, 61

Sirén, Jouni. Compressed suffix arrays for massive data. In *String Processing and Information Retrieval*, pages 63–74. 2009. doi:10.1007/978-3-642-03784-9_7. URL http://dx.doi.org/10.1007/978-3-642-03784-9_7. 53

Smith, L. M., Sanders, J. Z., Kaiser, R. J., Hughes, P., Dodd, C., Connell, C. R., Heiner, C., Kent, S. B. H., and Hood, L. E. Fluorescence detection in automated DNA sequence analysis. *Nature*, 321(6071):674–679, 1986. ISSN 0028-0836. doi:10.1038/321674a0. URL http://dx.doi.org/10.1038/321674a0. 3

Staden, R. A strategy of DNA sequencing employing computer programs. *Nucleic acids research*, 6(7):2601–2610, 1979. ISSN 0305-1048. URL http://www.ncbi.nlm.nih.gov/pmc/articles/PMC327874/. 6

The International Cancer Genome Consortium. International network of cancer genome projects. *Nature*, 464(7291):993–998, 2010. ISSN 1476-4687. doi: 10.1038/nature08987. URL http://dx.doi.org/10.1038/nature08987. 5

Tomato Genome Consortium. The tomato genome sequence provides insights into fleshy fruit evolution. *Nature*, 485(7400):635–641, 2012. ISSN 1476-4687. doi:10.1038/nature11119. URL http://dx.doi.org/10.1038/nature11119. 13

Välimäki, Niko, Ladra, Susana, and Mäkinen, Veli. Approximate All-Pairs Suffix/Prefix overlaps. In Amir, Amihood and Parida, Laxmi, editors, *Combinatorial Pattern Matching*, volume 6129 of *Lecture Notes in Computer Science*, pages 76–87. Springer Berlin / Heidelberg, 2010. doi:10.1007/978-3-642-13509-5_8. URL http://dx.doi.org/10.1007/978-3-642-13509-5_8. 52

Valouev, Anton, Ichikawa, Jeffrey, Tonthat, Thaisan, Stuart, Jeremy, Ranade, Swati, Peckham, Heather, Zeng, Kathy, Malek, Joel A., Costa, Gina, McKernan, Kevin, et al. A high-resolution, nucleosome position map of c. elegans

reveals a lack of universal sequence-dictated positioning. *Genome research*, 18(7):1051–1063, 2008. ISSN 1088-9051. doi:10.1101/gr.076463.108. URL http://dx.doi.org/10.1101/gr.076463.108. 5

Venter, J. Craig, Adams, Mark D., Myers, Eugene W., Li, Peter W., Mural, Richard J., Sutton, Granger G., Smith, Hamilton O., Yandell, Mark, Evans, Cheryl A., Holt, Robert A., et al. The sequence of the human genome. *Science*, 291(5507):1304–1351, 2001. ISSN 1095-9203. doi:10.1126/science.1058040. URL http://dx.doi.org/10.1126/science.1058040. 4, 7, 22

Wakeley, J. The excess of transitions among nucleotide substitutions: new methods of estimating transition bias underscore its significance. *Trends in ecology & evolution*, 11(4):158–162, 1996. ISSN 0169-5347. URL http://view.ncbi.nlm.nih.gov/pubmed/21237791. 115

Warren, René L., Sutton, Granger G., Jones, Steven J. M., and Holt, Robert A. Assembling millions of short DNA sequences using SSAKE. *Bioinformatics*, 23(4):500–501, 2007. ISSN 1460-2059. doi:10.1093/bioinformatics/btl629. URL http://dx.doi.org/10.1093/bioinformatics/btl629. 13

Waterston, Robert H., Lander, Eric S., and Sulston, John E. On the sequencing of the human genome. *Proceedings of the National Academy of Sciences*, 99(6):3712–3716, 2002. ISSN 1091-6490. doi:10.1073/pnas.042692499. URL http://dx.doi.org/10.1073/pnas.042692499. 4

Waterston, Robert H., Lander, Eric S., and Sulston, John E. More on the sequencing of the human genome. *Proceedings of the National Academy of Sciences*, 100(6):3022–3024, 2003. ISSN 1091-6490. doi:10.1073/pnas.0634129100. URL http://dx.doi.org/10.1073/pnas.0634129100. 4

Wheeler, David A., Srinivasan, Maithreyan, Egholm, Michael, Shen, Yufeng, Chen, Lei, McGuire, Amy, He, Wen, Chen, Yi-Ju, Makhijani, Vinod, Roth, G. Thomas, et al. The complete genome of an individual by massively parallel DNA sequencing. *Nature*, 452(7189):872–876, 2008. ISSN 0028-0836. doi:10.1038/nature06884. URL http://dx.doi.org/10.1038/nature06884. 5

Ye, Kai, Schulz, Marcel H., Long, Quan, Apweiler, Rolf, and Ning, Zemin. Pindel: a pattern growth approach to detect break points of large deletions and medium sized insertions from paired-end short reads. *Bioinformatics*, 25(21):2865–2871, 2009. ISSN 1460-2059. doi:10.1093/bioinformatics/btp394. URL http://dx.doi.org/10.1093/bioinformatics/btp394. 112

Zerbino, Daniel R. and Birney, Ewan. Velvet: Algorithms for de novo short read assembly using de bruijn graphs. *Genome Research*, 18(5):821–829, 2008. ISSN 1549-5469. doi:10.1101/gr.074492.107. URL http://dx.doi.org/10.1101/gr.074492.107. 8, 13, 54, 61