APPROACHES TO SCALING AND IMPROVING METAGENOME SEQUENCE ASSEMBLY

By

Jason Pell

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Computer Science

2013

ABSTRACT

APPROACHES TO SCALING AND IMPROVING METAGENOME SEQUENCE ASSEMBLY

By

Jason Pell

Since the completion of the Human Genome Project in the early 2000s, new high-throughput sequencing technologies have been developed that produce more DNA sequence reads at a much lower cost. Because of this, large quantities of data have been generated that are difficult to analyze computationally, not only because of the sheer number of reads but due to errors. One area where this is a particularly difficult problem is metagenomics, where an ensemble of microbes in an environmental sample is sequenced. In this scenario, blends of species with varying abundance levels must be processed together in a Bioinformatics pipeline. One common goal with a sequencing dataset is to assemble the genome from the set of reads, but since comparing reads with one another scales quadratically, new algorithms had to be developed to handle the large quantity of short reads generated from the latest sequencers. These assembly algorithms frequently use de Bruijn graphs where reads are broken down into k-mers, or small DNA words of a fixed size k. Despite these algorithmic advances, DNA sequence assembly still scales poorly due to errors and computer memory inefficiency.

In this dissertation, we develop approaches to tackle the current shortcomings in metagenome sequence assembly. First, we devise the novel use of a Bloom filter, a probabilistic data structure with false positives, for storing a de Bruijn graph in memory. We study the properties of the de Bruijn graph with false positives in detail and observe that the components in the graph abruptly connect together at a specific false positive rate. Then, we analyze the memory efficiency of a partitioning algorithm at various false positive rates and find that this approach can lead to a 40x decrease in memory usage.

Extending the idea of a probabilistic de Bruijn graph, we then develop a two-pass error correction algorithm that effectively discards erroneous reads and corrects the remaining majority to be more accurate. In the first pass, we use the digital normalization algorithm to collect novelty and discard reads that have already been at a sufficient coverage. In the second, a read-to-graph alignment strategy is used to correct reads. Some heuristics are employed to improve the performance. We evaluate the algorithm with an *E. coli* dataset as well as a mock human gut metagenome dataset and find that the error correction strategy works as intended.

To my wife, Heather.

ACKNOWLEDGMENTS

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Overview

In Bioinformatics, the genome assembly problem is among the most challenging and pressing to date. Informally, the goal of assembly is to determine, from shredded pieces of DNA fragments, the original DNA sequence. Though assembly algorithms have been in development for decades, an accurate and efficient assembler for all but the most simple genomes has not been achieved, for reasons we will discuss briefly.

One principle reason for the difficulty in assembling genomes is the proliferation of so-called next-generation sequencing technologies. Though these newer platforms have dramatically reduced the cost of sequencing, they typically produce smaller, more erroneous reads, which are more difficult to find overlaps between. For example, Illumina's HiSeq 2000 System is capable of producing 6 billion reads of approximately 100 base pairs in length. In contrast, the traditional, capillary-based methods (i.e. Sanger sequencing) have generally produced longer reads with fewer errors but at a greater cost and lower throughput. As newer technologies are developed, the phrase next-generation sequencing becomes more of a misnomer as companies are constantly developing new approaches that produce a combination of higher throughput, fewer errors, and longer reads.

1.2 Assembly Algorithms

As sequencing technologies have evolved, so have the assembly algorithms. The primary assembly method for assembling traditional Sanger data is called Overlap-Layout-Consensus (OLC). These assemblers typically apply three primary stages using the long reads that capillary-based sequencers produce. In the Overlap stage, an assembly/overlap graph is constructed by checking for overlaps between reads. In the next step, the reads are laid out along a putative genome. Finally, the reads are aligned to one another and a consensus sequence is determined. Examples of such assemblers include Celera[36], pcap[20], and Arachne[2].

Due to the large number of reads generated by next-generation sequencing platforms, the overlap step of OLC assemblers became intractable due to being quadratic with respect to the number of reads. To better handle these types of datasets, k-mer graph, or DNA de Bruijn graph, methods took over as the predominant assembly method. Rather than checking for overlaps among entire reads, each read is broken up into words of fixed size k. Each k-mer is stored in a data structure, generally a hash table, to enable fast lookup. An assembler can represent k-mers as vertices in an assembly graph, where k-mers can share an edge if they share a k-1 word overlap. The algorithms for traversing the graph to generate contigs are assembler-dependent. One of the first DBG assemblers, EULER, finds an Eulerian path through the graph[40]. Other assemblers, such as Velvet[54] and SOAPdenovo[28], use Dijkstra's algorithm to detect bubbles between two k-mers. Assemblers apply their own heuristics to handle repetitive elements, errors, and heterozygosity[35].

Though the de Bruijn graph approach is the most predominant in next-generation sequence assembly, other methods exist. One such approach has been with string graphs[37]. A string graph is similar to the overlap graph used in OLC approaches, but it does not

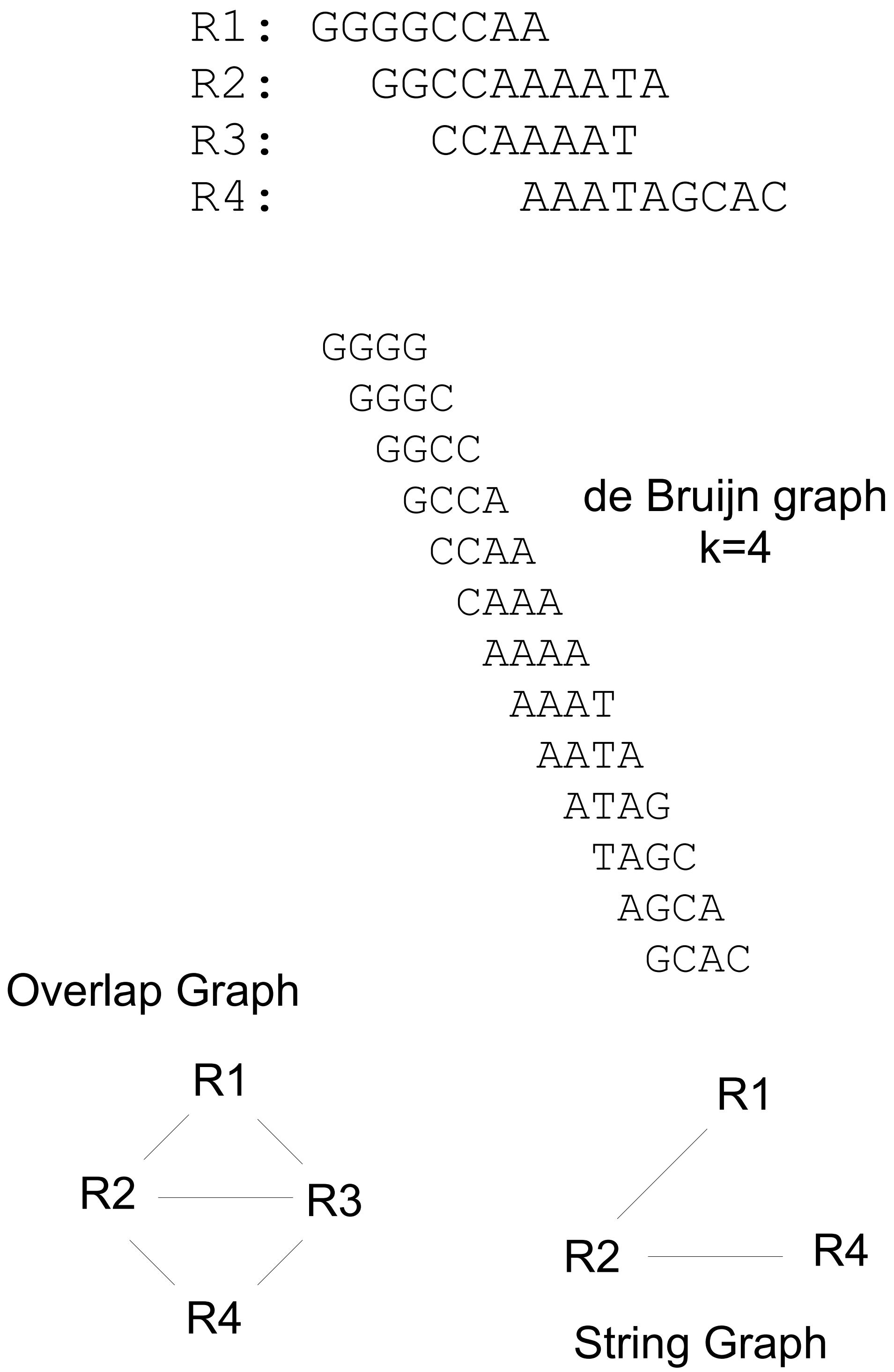


Figure 1.1: Three main types of assembly graphs.

necessarily need to know all of the overlaps among reads. Instead, it requires all of the read overlaps that cannot be transitively reduced. For example, if read A overlaps with read B, read B overlaps with read C, and read A overlaps with read C (and reads B and C overlap the same side of the read as A), then the overlaps from A to B and B to C do not need to be considered since the overlap from A to C transitively implies the former overlaps. Another feature of the string graph is that it does not contain the entire set of reads from a sequencing dataset. If a read is contained within another read (i.e. it is a substring), then it does not need to be part of the initial string graph. However, knowing all of the reads may be important for determining coverage levels that are useful for resolving repetitive regions and copy number variations (CNVs). Initially, the string graph approach was not utilized because the running time was still quadratic with respect to the number of reads. However, using the FM-index data structure, the running time was brought down to O(N)[45]. In the next chapter, we go into more detail about how the FM-index can be used to store assembly graphs and compare it to other data structures and approaches.

Figure 1.1 provides a visual comparison between the three main types of assembly graphs. One immediate observation is that the overlap graph and string graph scale with the number of reads while the k-mer graph scales with the number of distinct k-mers in the sample. This implies a trade-off where longer reads make string and overlap graphs more memory efficient while de Bruijn graph-based assemblers are better for shorter and more abundant reads. Another observation is that the string graph does not utilize read 3 since it is contained within read 2. This offers a potentially significant memory reduction since many reads may not be needed. Furthermore, sequencing projects utilizing both short and long reads benefit from this approach because many more short reads will be contained within the longer reads.

1.3 Metagenomics

The assembly problem is made even more challenging when presented with metagenomic data. Metagenomics, or environmental sequencing, is the study of sequence data obtained from environmental samples, which can contain different species of microbes with varying abundance levels. This is particularly useful when it is difficult to isolate a microbial species from a sample or one is interested in the entire population as a whole. For example, one may be interested in the microbial population in the human gut or in a water sample from the ocean. Since many species exist in a sample, more questions can be answered from looking at the entire population than isolating individual species, especially considering the complex dynamics that occur in microbial ecology.

In contrast to metagenome sequencing, genomic sequencing is considered an easier problem because, other than issues such as contamination and PCR biases, only one genome is isolated and sequenced. A further hindrance to metagenomes is the amount of coverage needed. As of 2010, the total amount of environmental sequencing constitutes less than 1% of what would be found in a gram of soil[17]. In addition, due to the uneven abundances of microbial species, deep sequencing is needed to discover all of the diversity in such a sample. For example, if two species exist in a sample with one being 100 times more common than the second, it is clear that one would need 1000X coverage of the sample in order to obtain at least 10X coverage for each individual species. The difference between the least and most abundant species in a metagenomics sample is likely to be much greater in highly diverse populations than others.

1.4 Why Assemble?

A variety of tools are available that do not require an assembly to obtain information about a metagenome sequence dataset. Furthermore, depending on the analysis, it may not make sense to obtain the shotgun sequence of the environmental sample. If the goal is to determine the diversity within a population, it is useful to target and amplify the ribosomal 16S sequence and sequence with a platform that provides longer reads such as 454. This allows one to construct a phylogenetic tree that can be compared with similar samples of different treatments or locations. If one is interested in the functional composition of a microbial population, then a more complete sampling of the metagenome is needed. In this case, a variety of analyses are possible without assembling. Gene fragments can be obtained by looking at the six open reading frames and searching through a protein database. Many tools such as MG-RAST[34] and FragGeneScan[42] are available that can assist in metagenome sequence analyses that do not require an assembly.

However, despite the options for metagenome sequence analysis without a reference assembly, there are many limitations. First, because a sequence obtained from next-generation sequencers has a specific maximum read length, only gene fragments can be obtained rather than entire genes. This limits how effective metagenome analysis tools can be at analyzing a sample since it may only be possible to find a match to a protein domain rather than a protein. This lowers the likelihood that a function can be assigned to the sequence. By assembling, one has the ability to pull out longer contigs and possibly large portions of entire genomes.

1.5 Problem Statement

Though current sequencing technologies has enabled deep sampling of highly diverse microbial ecosystems, algorithmic methods for analyzing these datasets are still in their infancy. Assembling less diverse metagenomics datasets is possible, especially with the computing resources available at large sequencing centers and supercomputing facilities. However, high-throughput sequencing technologies is now inexpensive enough that labs with modest budgets are able to sequence environmental samples of interest. Thus, a clear gap exists between the ability to sequence complex environments and the ability to analyze these large datasets. Though the running time of sequence assembly is a concern, the primary bottleneck that needs to be addressed is memory usage.

Another concern with metagenome sequence assembly is accuracy. Since an estimated 99% of microbes cannot be cultured in a laboratory, it is unlikely that reference genomes can be obtained for the majority of microbes in an environmental sample. Since many assembly validation techniques require the use of a reference genome, new techniques must be developed to produce accurate assemblies. Prior to assembly, one of the most important procedures in an assembly pipeline is read error correction. However, many techniques rely on being able to obtain the coverage of the sequenced genome. Metagenome assembly pipelines need methods for improving the accuracy of assemblies without the ability to compare to a reference.

1.6 Significance of Research

In this dissertation, we will establish a set of approaches that enable the assembly of extremely diverse metagenomic samples. Currently, this is not possible in high-performance computing facilities, let alone on commodity machines. By making metagenomic assembly easier to do on the cloud and in supercomputing facilities, we can enable researchers in any lab to obtain a metagenomic assembly. By attempting to tackle both the high-memory and compute requirements of assembly, this research will reduce the bottlenecks that currently plague assembly pipelines.

Solving the metagenome sequence assembly problem has far reaching consequences in basic and applied sciences. As mentioned previously, few microbes can be cultured in the laboratory, so sequencing environmental populations is one of the few ways to better understand individual microbial species and populations. By deep and broad sampling throughout the Earth, it is likely that novel genes will be discovered that can be used to produce better biofuels and lead to new insights into drug development. By improving soil assemblies, it is possible to better understand how soil and its microbial ecosystem sequesters carbon, fixates nitrogen, and how it provides a fertile location for crops to grow. Though studies on the human gut microbiome are in progress, better assemblies will lead to more insight into the interactions between microbial populations and human cells within the human ecosystem. Given that so much diversity has not been sampled, new approaches that make the analysis of metagenomics datasets will lead to more discoveries.

1.7 Outline of Dissertation

To aid in the assembly of metagenomes, we use a Bloom filter (or Counting Bloom Filter, where appropriate) to store sequence data in order to accurately and efficiently pre-process next-generation sequencing datasets, which will improve the overall quality of assembled metagenomes. In the first approach, we will show how metagenome partitioning can greatly reduce the amount of memory needed to assemble datasets. We evaluate this approach on an *E. coli* dataset as well as a small soil metagenome. For the second part, we utilize a read error correction algorithm that uses k-mer coverage information to discern between erroneous and true k-mers. We evaluate the error correction procedure on another *E. coli* dataset and a mock human gut microbiome dataset. By combining these approaches together, it is possible to assemble much larger datasets than with current methods.

Chapter 2

Review of Relevant Literature

DNA sequence assembly is a rapidly changing field. Early algorithms were designed to handle longer reads of higher quality, but next-generation sequencing platforms have forced Bioinformaticians to rethink previous approaches. In this chapter, we present several existing works on sequence assembly, particularly for metagenomes. In addition, we introduce pre-assembly filtering and error correction methods that are relevant to the framework of this dissertation. However, because a focus of this research is on scaling assembly, we begin our discussion with a review of data structures that are useful in assembly pipelines.

2.1 Data Structures in Assembly

To scale assembly the choice of data structures is critical as an assembly tool must be able to handle processing and storing billions of reads quickly. In this section, we will discuss many of the data structures used for k-mer storage in pre-assembly tools as well as those used for storing assembly graphs.

2.1.1 Hash Tables

Hash-style data structures are likely the most popular choice for storing assembly graphs, particularly in de Bruijn graph-based assemblers. Hash tables are one of the most important data structures in Computer Science, and many variants exist. A hash table maps a key to a value. For k-mer based assemblers, generally the k-mer is the hash and the value is the number of occurrences for that k-mer in the dataset.

For a representative example of how a hash table is used in an assembler, we will discuss how ABySS[47] implements a distributed de Bruijn graph. ABySS uses Google's SparseHash implementation of a hash map. This is advantageous because within the range of k used in assembly (generally greater than 20), the number of possible k-mers is much larger than the actual number in a dataset. Thus, using SparseHash keeps the memory overhead relatively low. For each k-mer, 8 bits are used to represent the up to 8 possible edges that a k-mer can have. If the corresponding neighbor exists, then the bit is flipped. This is advantageous to querying the k-mer itself because if the bin where the neighbor k-mer is located is on another node, it will be considerably slower than simply checking the status of a bit.

To distribute the assembly graph across multiple nodes, ABySS uses a simple scheme. Each k-mer (with its reverse complement) is converted to a number, say i. If there are n hash tables, then the k-mer is located at node $i\%n$. Though this is simple to implement, there are some drawbacks depending on the type of high-performance computing environment where the assembler is being run. Due to the nature of de Bruijn graphs, the neighboring k-mers for a particular k-mer are probably on different nodes. The 8 bit scheme discussed earlier partially remedies this, but if a neighboring k-mer does indeed exist, then the node on which that k-mer exists must be queried. Thus, components in the assembly graph are likely to be located on several different nodes if many nodes are used by the assembler. In the Genome Assembler section, we will discuss in greater detail how ABySS implements a k-mer graph.

Hash tables offer a number of advantages over other data structures concerning scaling behavior. As we discussed previously with ABySS, it is possible to easily distribute hash tables on multiple nodes. Another advantage is that while query and insertion times are not guaranteed to be constant time given a sub-par hash function, for practical purposes hash tables are generally very fast. However, handling collisions can add to the memory footprint.

2.1.2 Trie

Though a regular trie data structure is rarely used in sequence assemblers, special cases such as suffix trees, suffix arrays, and so on have been used in assemblers in the past[8, 18]. Trie-style data structures offer a number of interesting properties and are useful for string processing applications in Bioinformatics. A trie, or prefix tree, is a tree where the root is an empty string and words build from it. The first character in each word of the data structure is a child node of the root. Thus, if two words share the same prefix, they share a set of nodes that contains that prefix. Several variants of tries exist that offer a number of advantages over a basic trie.

2.1.2.1 Suffix Tree

An important building block in examining trie-style data structures in assemblers is the suffix tree. A suffix tree is a trie where each possible suffix of a tree is added to a trie. In 1973, the first linear-time construction of a suffix tree enabled faster string searching at the expense of a string indexing step[52]. However, one drawback that keeps classic suffix trees from being useful in Bioinformatics is that they are more memory intensive than the leading approaches.

2.1.2.2 Suffix Array

Though suffix arrays[31] have not been extensively used for de Bruijn graph-based assembly (though exceptions such as the Edena[18] assembler exist), they have been useful for a variety of pre-assembly filtering and sequence analysis purposes. Suffix arrays use up to three to five times less memory than suffix trees[52]. A related data structure, the FM-index[16], has recently been used as a way to store string assembly graphs, which we will elaborate on briefly.

To construct a suffix array, first one must generate all of the suffixes for a string (or a set of strings). These suffixes are then sorted, and the starting position of the suffix is stored as the array element where the array index is the sorted order for that suffix. The result is a single array with the number of elements equal to the size of the string. However, this alone does not enable fast lookup since a binary search would be required to determine whether a pattern exists in the original string. Adding an additional data structure such as an LCP (longest common prefix) array improves the speed of searching.

Suffix arrays enable many rapid string operations that are amenable for Bioinformatics algorithms. One application for this data structure is for k-mer counting. Jellyfish[32] and tallymer[49] are two examples of k-mer counting software packages that utilize some kind of suffix array. Though suffix arrays improve the speed of searching for a pattern within a dataset, they can significantly add to the memory footprint. Tallymer tackles this problem by giving the user an option to split the suffix array into several different partitions. If the array is divided into two partitions, then only half is put into memory at a time. If the entire suffix array cannot fit into the amount of memory given, this can significantly improve the running time of the data structure construction even if multiple passes through the dataset are required. Jellyfish handles the problem differently; it uses a specialized, memory-efficient hash table and takes advantage of compare-and-swap in modern CPUs to enable parallelization. Combining these approaches allow jellyfish to be much more memory efficient than tallymer and other k-mer counters.

2.1.2.3 FM-Index

Short read mappers such as Bowtie[26] and BWA[27] used the Burrows-Wheeler transform (BWT)[7] to reduce the memory requirements for storing the reference genome on which the short reads are mapped. However, the use of the BWT was not used in assembly algorithms until a method for storing a string graph with an FM-index was discovered[16]. An FM-index, which we will describe in greater detail below, uses the BWT as a component of the FM-index. This approach led to the creation of the String Graph Assembler (SGA)[46].

FM-indices offer a number of advantages compared to other data structures. However, while FM-indices enable fast lookup after graph construction, constructing the data structure from a set of reads is relatively time consuming. Another issue is that the data structure scales with the number of reads inserted rather than k-mers. With effective compression, FM-indices can compare favorably to other implementations, but for highly diverse metagenomics datasets this is not the case.

The first step of an FM-index is to take the Burrows-Wheeler Transform of the set of reads. The transform takes every cyclic permutation of a string and then sorts them. Since each read is a separate string, a unique character can be used between each read to enable fast lookup by read ID. One important feature of the Burrows-Wheeler Transform is that it can compress the reads due to the way the strings are sorted. In addition to this transform, the FM-index requires two additional data structures: a counts table and an occurrences table. The counts table contains a simple lookup between a character (which could only be the four letters used to represent nucleotides or an expanded alphabet) and the number of times it appears in the dataset along with any characters lexicographically smaller. The occurrences table provides a lookup for the number of times a character occurs in a prefix of the transformed text. Using these three data structures together, it is possible to enable fast queries of the text.

When reads are added to an FM-index, it is possible to determine the reads that overlap another using the algorithm described in [45]. However, this algorithm returns all reads that overlap rather than an irreducible set, which they also include. One drawback to the paper (which is explicitly discussed) is that it does not handle mismatches, which is necessary for real sequencing data. In another paper, the same authors outline the algorithm used in SGA that enables the assembler to handle inexact overlaps[46].

2.2 Assemblers

Currently, most mainstream assemblers are tailored specifically for genome assembly. Some assemblers, such as SOAPdenovo[41] include an option on the command line to denote a metagenome assembly. In other cases, 3rd party add-ons are built around the original assembler to handle non-genomics datasets (e.g. MetaVelvet[38]). Finally, many other assemblers are built specifically for metagenome assembly. We will discuss many of the approaches and current metagenome sequencing projects that have occurred to date.

2.2.1 Genome Assembly

EULER[40] was the first assembler that used a de Bruijn graph and was published in 2001, a few years before the first 2nd generation sequencers were released. Thus, EULER made use of the long reads produced by Sanger, and it was built primarily to resolve repeats rather than improve memory efficiency. It also made use of k-mer abundance based error correction.

Once sequencing platforms from Solexa (now Illumina) and 454 were released, many new assemblers emerged. Velvet[54] was released in 2008 and is still maintained and used in many assembly pipelines today. In addition to being a de Bruijn graph assembler, it includes procedures for removing tips (i.e. erroneous, hanging ends in the read) and bubbles (i.e. cycles in the graph caused by errors or heterozygosity). ABySS[47] was released shortly after and included many similar error correction methods. What distinguished ABySS was its use of a distributed hash table to enable high-memory assembly of large genomes. However, because k-mers that are part of the same component frequently end up on different nodes, the assembler can run into performance issues due to the speed of the network.

A recent assembler/variant caller, Cortex[21], introduced the concept of colored de Bruijn graphs. Rather than loading reads from different samples into the same bins in a hash table, Cortex assigns a different color to each sample so that variants such as SNPs, indels, and structural variation can be called from the assembly phase. Cortex was primarily designed for bacterial variant calling and for the human genome project to enable the ability to call variants in 1000s of different human genomes.

2.2.2 Metagenome Assemblers

Assemblers created specifically for metagenomes have not been available until recently. One reason for this is that sequencing platforms have only now been able to produce the deep sampling required to probe highly diverse samples. Another reason is that they are inherently more challenging than genome assemblers: while many genome assemblers can take a more or less uniform sampling for granted, metagenomes contain varying abundance levels. Furthermore, with the sequencing of a single species, while heterozygosity is a challenging problem when determining whether a SNP is real or a sequencing error, the concept of a single species is more fuzzy for a population of microbes. Thus, metagenome assemblers are still in their infancy and there is much room for progress in the field.

2.2.2.1 MetaVelvet

One such assembler is called MetaVelvet[38]. MetaVelvet uses much of Velvet's code base (and thus is a de Bruijn graph based assembler) but is tailored to handle the uneven coverage level found in most metagenomes. One key feature behind MetaVelvet is that it does a type of partitioning to separate different microbial species based on graph connectivity and coverage level. Whereas Velvet truncates sequence when the coverage level in the graph differs largely from the expected coverage, MetaVelvet uses a mixture model to approximate different coverage levels and separates them in order to prevent chimeric contigs. This is advantageous compared to neglecting to incorporate coverage levels because species that share sequences could end up being assembled together, leading to a misassembly. While MetaVelvet was shown to produce better results (measured by N50 length) compared to other methods, it is very memory intensive, which makes it impractical for large datasets.

2.2.2.2 Meta-IDBA

Another assembler created specifically for metagenomes is called Meta-IDBA[39]. Meta-IDBA, too, uses a de Bruijn graph for assembly. The authors make a distinction between cr-branches, which are branches in the graph that result from similar genes, and sp-branches, which are branches that result from variation, such as SNPs and indels. These types of branches do not occur as often when assembling a single genome, so Meta-IDBA is designed to detect and handle these cases. However, the method to resolving these issues are similar to genome assemblers, except when encountering a branch (i.e. a bubble or tip as described in Velvet), both paths are generally kept while a single genome assembler would generally discard one branch as an error or treat the bubble as being caused by heterozygosity.

2.2.2.3 Ray Meta

In late 2012, a metagenome assembler was published called Ray Meta[5], which specializes in distributed assembly of metagenomes. It is based off a genome assembler called Ray[4], which also focuses on the scalability of genome assembly through a better distributed computing model. Ray Meta is a de Bruijn graph based assembler, but rather than partitioning the graph based on coverage levels, component finding, or graph simplification, it applies a "heuristic-guided graph traversal" that result in good assemblies.

To demonstrate the effectiveness of the approach, the authors simulated two metagenomes containing 100 and 1,000 genomes where each genome had a different coverage level. They also assembled and analyzed a dataset of 124 gut microbiomes. These results were compared with the MetaVelvet assembler with the Ray Meta assembler showing superior results.

Ray Meta shows that distributing a de Bruijn graph generated from metagenomic data is effective in scaling. However, it is not clear how the approach will scale to highly diverse datasets like soil since a simulated dataset of 1,000 genomes required 1,024 cores with 1.5GB of memory per core. A soil sample could potentially contain 1,000 times more genomes and the graph could be more complex. Nonetheless, the approach is well-demonstrated and will likely be improved upon to handle such scale.

2.3 Recent Works in Metagenomics

Though some types of metagenomics, particularly those that used 16S ribosomal sequencing, have been around for a longer period of time, it was not until recently that shotgun sequencing of environmental samples has become popular. A seminal work in the field involved shotgun sequencing an Acid Mine Drainage sample[50]. Due to the high abundance and low heterozygosity of only a handful of the species in the sample, it was possible to almost completely construct two genomes (*Leptospirillum* group II and *Ferroplasma* type II) and partially reconstruct three additional genomes.

Another important work in metagenomics was led by Craig Venter where they sequenced a water sample from the Sargasso Sea[51]. They obtained approximately a gigabase of data, and attempted an assembly using the Celera assembler[36]. However, they experienced a now-commonly encountered problem where reads from highly abundant species would be flagged as repetitive regions in the assembler, which would result in a poor assembly. To resolve this, they had to incorporate some manual steps to resolve the issue. Ultimately, 1,800 different species were found where 148 were newly-discovered. They also discovered more than a million genes, suggesting that a great deal of novelty is yet to be discovered using metagenomics. As this paper covered a small number of samples in the Sargasso Sea, different communities are likely to live in other waters and depth levels. Furthermore, because Sanger sequencing was used, there is likely even more to be discovered in the water sample taken.

Both of the previously mentioned works used Sanger sequencing to obtain their data. More recently, a number of works have been published using Illumina technology. The MetaHIT Consortium published the initial results of their international human gut microbiome project[41]. They sequenced the microbes in 124 different faecal samples with the goal of obtaining a gene catalog. Overall, approximately 576Gb of sequence data was obtained. Because of the short reads in Illumina platforms, they used the SOAPdenovo assembler to obtain contigs before cataloging the genes found. Initially, the dataset for each sample was assembled separately, and then any of the reads leftover were pooled together and assembled. Prior to any assembly, the reads were filtered to remove any low abundance reads that would be unlikely to be part of any assembly. Resulting from the assemblies, they found over three million genes in total.

Other metagenomics efforts that have required assembly include the cow rumen[19] and permafrost soil[30]. To assemble the cow rumen metagenome dataset, a filtering program developed at the Joint Genome Institute was used to remove well-known sequencing artifacts. Furthermore, low abundance reads were removed since their coverage level would not meet the threshold required by most assemblers. The permafrost soil assembly was filtered and assembled in a similar manner as it was also a project by the Joint Genome Institute. In 2012, Iverson et al. reported that they were able to close the assembly of a highly predominant microbial species found in the Puget Sound[22]. The genome was from marine group II *Euryarchaeota*, which has not yet been cultured. They used several custom filtering steps prior to assembly, primarily for dealing with SOLiD reads. These procedures do simple read trimming using quality scores, read error correction, and PCR amplification bias removal.

Frequently, previously published metagenome datasets are reanalyzed to reproduce the results of a paper or look for new insights. For example, Wu et al. developed a method to look for previously unknown branches in the evolutionary tree[53]. Using their approach, they identified homologous genes that had not been characterized from the Sargasso Sea dataset. They concluded that these may be from viruses, ancient paralogs, or even genes belonging to organisms distantly unrelated to any that have been discovered and characterized. However, because these genes nor the organism or virus to which they belong have not been studied in greater detail, it is impossible to determine with certainty how the homologs should be classified.

2.4 Sequence Filtering

Due to the large size of datasets, it is becoming more common to filter out a large number of reads to make it feasible to assemble or do some type of high-memory analysis of the data. However, several different methods exist that can pose problems for downstream analyses. We will introduce a couple of these methods and discuss their advantages and shortcomings.

2.4.1 Abundance Filtering

Most commonly for metagenomic datasets, reads are filtered by using k-mer coverage as a way of removing erroneous k-mers and reads[41, 19]. In some cases, reads that fall below a certain level of k-mer coverage as one walks from the 5' end of the read to the 3' read is trimmed at a certain threshold. If all of the k-mers in a read fall below the threshold, then the entire read is discarded. This method has been effectively used to lower the memory requirements for assemblies, specifically for metagenomic datasets. However, the biggest shortcoming of this approach is that low-abundance novelty will be discarded since coverage levels are uneven.

2.4.2 Digital Normalization

A recently developed method, digital normalization, looks at the median k-mer coverage for a read[6]. If the read falls below a specific threshold, then the read is kept and discarded otherwise. While the algorithm is simple and easy to implement, it has the effect of evening out the level of coverage in a dataset. This is particularly attractive for metagenomics datasets because most assemblers expect an even level of coverage for parameter estimation.

The digital normalization algorithm relies on the fact that the coverage level of the k-mers within a read are similar. By using a Count-Min Sketch[11], a data structure that enables memory-efficient counting with a false positive rate and possible overestimation, it is possible to process large amounts of sequence data with little memory footprint. By treating a sequence read dataset as a stream, the digital normalization algorithm can be considered an online algorithm as it only requires one pass through a dataset. Though the digital normalization paper used genome and mRNAseq datasets, the results clearly showed that the "correct" k-mers were retained while a large portion of erroneous k-mers are thrown out.

2.5 Graph Alignment

One recent advance in gene targeted assembly combines A* graph search with aligning profile hidden Markov models (profile HMM) with de Bruijn graphs in a software package called Xander(Jordan Fish, personal communication). In this approach, the profile HMMs are treated as the query sequence and a set of sequence reads are considered the database. The sequence reads are initially loaded into a Bloom filter with a low false positive rate. To assemble a gene, one searches the corresponding profile HMM against the de Bruijn graph found in the Bloom filter. More accurately, the search is done by combining the elements of both k-mers and the states of a hidden Markov model. Thus, a vertex has a k-mer, read position, and state (e.g. insertion, match, deletion). The edges connecting these vertices are determined by the probabilities of the HMM. Since the database of sequence reads are in nucleotide-space, the profile HMM must be transformed to nucleotide-space in order to find seeds. The graph is then walked by 3 bases at a time to ensure that valid codons are being searched. Once seeds are found, an A* graph search strategy is applied by using the optimistic heuristic that the remaining nodes in the path to the goal (i.e. the end of the alignment) will be perfect matches. Because this is an optimistic heuristic, it is admissible and thus the graph search strategy will find a shortest path/optimal alignment.

The biggest advantage to the approach in Xander is that it gives biologists a method for finding genes of interest in metagenomic samples. For many reasons, especially uneven coverage and errors, a *de novo* metagenome assembly is unlikely to assemble all of a biologist's genes of interest even if they exist in the dataset. However, using a profile HMM introduces biases due to the fact that profile HMMs are generally built using genes that have already been discovered. A homologous gene in a metagenomic sample (or any sample for that matter) could be part of an under-sampled outgroup phylogenetically or the gene may have undergone rapid selection compared to other genes in the family. These biases would cause some of the novelty in the original set of reads to be lost in favor of the biases of the profile HMM. Another shortcoming to the approach is that the Bloom filter does not incorporate coverage information. This makes it difficult to discern between a sequencing error and true novelty. Incorporating this information could be a way to overcome some of the biases with using profile HMMs that were mentioned previously.

2.6 Error Correction

Given that errors in reads is one of the most challenging aspects to address in assembly, many error correction algorithms have been developed in order to attempt to address these challenges. While many error correction algorithms are implemented as separate packages that can be incorporated into an assembly pipeline, many assemblers include their own error correction procedures. In this section, we will focus on the error correction algorithms that are most closely related to the error correction method that will be introduced in a later chapter. While these algorithms are primarily developed for genomic reads, we will address how they may be useful for metagenomic datasets as well.

2.6.1 Quake

One read error corrector that has been part of many assembly pipelines for evaluation purposes such as GAGE[44] and the Assemblathon[13] is Quake[23]. The central idea behind Quake is that it incorporates base quality values into the k-mer coverage distribution of the dataset in order to better discriminate between erroneous k-mers and low coverage true k-mers. To determine the k-mer coverage distribution, all of the k-mers must first be counted. However, the count for a k-mer is not simply incremented each time it is discovered. Instead, for each base in the k-mer, the probability that the base is correct based on the quality value is multiplied. Initially, Quake used a Hadoop-style algorithm to quickly count the k-mers. However, after the Jellyfish[32] k-mer counting software package was released they switched to using it by default. Quake uses a mixture model on the k-mer coverage distribution to predict whether or not a k-mer is erroneous. For the true k-mers, a Gaussian distribution was used for coverage, and the zeta distribution was used for copy number. For the error k-mer distribution, the gamma distribution was used. In order to correct a read, each k-mer is determined to be either "trusted" or not. The end goal is to correct the reads so that all of the k-mers are trusted. By looking at the pattern of erroneous k-mers in a particular read, Quake localizes the errors and then uses maximum likelihood to find the most probable set of trusted k-mers.

One of the biggest advantages to using Quake is that it converts a regular k-mer into "q-mers" by incorporating base quality values to prevent low coverage real k-mers from being falsely changed. However, as is, Quake would be ineffective in correcting metagenomic datasets. The primary reason is that erroneous and real k-mers are much more difficult to separate by using the k-mer coverage distribution in these datasets. Since the abundance levels for species in environmental samples differ, it is unclear what the expected k-mer coverage level will be. Furthermore, it seems unlikely that the mixture distribution used in Quake would be effective for metagenomes, particularly the use of the Zeta function to estimate copy number. In order to make the use of a Quake-style error correction algorithm, one possibility would be to incorporate digital normalization to even out the coverage level.

2.6.2 Hammer

Not all error correction software packages use the k-mer coverage distribution to correct reads. One such package, called Hammer[33], uses a Hamming (edit-distance) graph to attempt to handle datasets with non-uniform coverage such as single-cell sequencing, mRNAseq, and metagenomics.

In the first step of the error correction algorithm, Hammer finds the set of distinct k-mers in the reads. Then, it finds all of the components in the Hamming graph using a "spaced seed" approach since comparing each pair of k-mers would be computationally intractable. A union-find data structure is used to quickly determine which k-mers are part of the same components. Next, each component is processed in order to attempt to find a consensus k-mer. If the size of the component is 1, then the k-mer is either correct or it has too many errors to connect within the edit-distance parameter that was set. In a manner similar to Quake, Hammer weights the score of a k-mer using its quality value. If the size of the component is greater than 1, then the highest occurring k-mer in the cluster is considered to be the consensus.

Hammer was shown to perform better than other error correctors on a single-cell sequencing dataset, but it did not perform as well as Quake on a regular *E. coli* dataset. One advantage to Hammer compared to other k-mer based approaches is that it is able to reconstruct k-mers that are not present in the dataset. This is particularly useful for Hammer's intended purpose, single cell sequencing datasets. While approaches that use Hamming graphs are useful, there are many drawbacks to consider. First, if one were to take into account indels, the Hamming graph is likely not a suitable candidate approach because the problem becomes more computationally prohibitive. A drawback to Hammer is that it assumes that no more than one k-mer in a cluster is correct. This calls into question the use of the software package on non-microbial genomes that are large and/or highly repetitive.

2.6.3 SEQuel

While most error correctors are generally employed in an assembly pipeline prior to assembly, some correction methods are used to refine an assembly and thus are placed after the assembly step in a pipeline. This is the idea behind SEQuel[43], which is a post-assembly error correction tool specifically tailored for single and multi-cell sequencing datasets. As with Hammer, SEQuel specializes in datasets with non-uniform coverage.

SEQuel goes through each contig and finds the reads that have a unique mapping to that contig. From here, SEQuel constructs what they coin a positional de Bruijn graph. This is similar to a de Bruijn graph except that it also accounts for the k-mer's position within a contig as well, so the same k-mer could occur in different vertices in the graph. Along with the k-mer sequence and position is the count that is obtained from the original mapping of the read to the contig. Since the k-mers are anchored to the contig, the graph is much simpler, which makes refining the assembly easier. Given the simplified graph, common errors such as bubbles can be corrected, and indeed SEQuel is shown to correct a large number of substitution and indel errors in the final assembly.

One of the biggest shortcomings to SEQuel is that it relies on a read mapper. Unfortunately, this is not easy to overcome but it limits the capabilities of the algorithm to the sensitivity and specificity of the mapper and the parameters chosen for that mapper. Another issue is that it is unclear how well the algorithm would work with more complex genomes because the paper only used microbial genomes as examples. However, what was perhaps most relevant to this dissertation is the use of a positional de Bruijn graph, which can place a k-mer at separate positions in a read and can simplify the overall structure of the assembly graph.

2.6.4 Musket

A more recent addition to read error correction software packages is Musket[29]. Musket uses many of the features from previously mentioned error correctors along with parallelization support that make it highly effective in both accuracy and performance. There are two stages to the error corrector: k-mer counting and coverage cut-off analysis, and then the actual error correction stage.

In the first stage, Musket uses a Bloom filter to eliminate as many distinct k-mers as possible. Thus, only k-mers that have already been seen by the Bloom filter will make it into the hash table. Using a Bloom filter produces false positives, but no non-distinct k-mers will be lost during the process. To parallelize this process, a master-slave model is used where k-mers are dissected from reads by master nodes, which then distribute the k-mers to slave nodes. Each slave handles a different set of possible k-mers. Once the hash table in each slave has been loaded with k-mers (after having distinct k-mers filtered out by the Bloom filter), all other distinct k-mers are removed from the hash tables, and the result is the k-mer abundance distribution of the dataset with all distinct k-mers removed. Finally, Musket uses the k-mer abundance distribution to estimate the cut-off between erroneous and trusted k-mers. Rather than using a mixed model as with Quake, Musket simply finds the point with the lowest k-mer count between the peaks though this can be overridden by a parameter.

In the second stage, Musket combines three different methods to achieve error correction. First, it uses a "two-sided correction" that introduces the concept of trusted bases. A trusted base is simply a base that is part of any trusted k-mer, which was determined from the previous stage. Next, Musket attempts to replace the untrusted bases with ones that would result in trusted k-mers. To speed up the process, only the end k-mers for an untrusted base is used so that multiple combinations do not need to be tested. However, this approach is too conservative when used on its own and only works when there is a single error in a k-mer. To address this, Musket adds another step to their correction workflow that attempts to correct untrusted k-mers next to trusted k-mers so that the k-mer has the highest count of any of the given alternatives. Musket has further steps to their workflow to quality check the corrections made and make changes as needed.

Being one of the most recent error correctors published, Musket offers a combination of features from other read error correction packages that give it good scaling characteristics as well as accuracy. The use of a Bloom filter is an effective way to screen out distinct k-mers that are likely to be erroneous. While some distinct k-mers make it through the first pass, they are taken care of in the next stage. Another interesting component in Musket is using the point of lowest density between the two peaks in the k-mer abundance distribution as the error cut-off. Other error correctors such as Quake use more complicated mixed models, but this simple approach seems to work just as well. While Musket is a viable option for correcting sequence datasets for genomes, it has yet to be proven for more complicated scenarios such as metagenomes. It is unlikely that the aforementioned cut-off method would work when examining the k-mer spectra of a metagenomic dataset. Thus, further pre-error correction filtering would be required to handle these situations.

Chapter 3

Probabilistic de Bruijn Graphs

In this chapter, we introduce the idea of a probabilistic de Bruijn graph. This novel graph representation utilizes a Bloom filter data structure for storing a k-mer assembly graph. We start by describing useful properties of de Bruijn graphs, then we break down how a Bloom filter can be used to store an assembly graph. We introduce an algorithm called partitioning that finds graph components in an assembly graph and use it on an actual soil metagenome. Next, we analyze the point at which the partitioning approach breaks down due to the increase in the false positive rate.

3.1 Properties of de Bruijn Graphs

In graph theory, de Bruijn graphs[12] are a well-studied class of graphs due to their interesting properties and application to computer networks. Recently, they have been shown to be useful for analyzing DNA sequence datasets. In this section, we will introduce some basic properties of de Bruijn graphs and also define a DNA de Bruijn graph and how they are used as DNA sequence assembly graphs.

A (s, k)-de Bruijn graph is a graph where s different symbols are used to form words of length k. If these words share a $k-1$ overlap, they are considered to be adjacent. Though a directed formulation is used in some literature, we will consider only simple de Bruijn graphs where there are no self-loops or directed edges. In total, there are s^k vertices in the graph where each vertex can have up to $2s$ adjacent vertices. While most vertices indeed have $2s$ neighbors, two special cases exist. For vertices that contain only the same symbol (e.g. 0000 in a binary de Bruijn graph with $k = 4$ and $s = 2$), there are $2s - 2$ adjacent vertices since we do not include the self-loop that would exist in a directed de Bruijn graph. Another case is where a word contains an alternating sequence (e.g. 0101 in a binary de Bruijn graph with $k = 4$ and $s = 2$). These words (i.e. k-mers) have $2s - 1$ because the directed version would contain parallel edges between that k-mer and another alternating sequence word. One property of de Bruijn graphs that make them a topic of study in computer networking theory is the short diameter length, k. The diameter is generally defined as the "longest shortest" path. In other words, when comparing the shortest path between each possible pair of vertices, the diameter is the longest of these paths.

We will now consider the application of de Bruijn graphs to DNA. In this case, we set $s = 4$ and let our alphabet only allow the symbols $a \in A, C, G, T$. The word size, k, can vary depending on the DNA sequence dataset of interest. Since DNA is double stranded, we may want to consider the reverse complement of a k-mer as being the vertex itself. In this case, rather than having a graph of 4^k vertices, the graph will contain either $\frac{4^k}{2}$ vertices if k is odd or $\frac{4^k}{2} - 4^{\frac{k}{2}}$ if k is even. The difference is due to the presence of k-mers that are their own reverse complement. They can be easily counted by dividing k by two and constructing the reverse complement of that sequence and concatenating it to the original sequence. For example, if $k = 4$, we can choose GT to be the first two letters of a sequence. The reverse complement is AC, so the full sequence is GTAC, which is it's own reverse complement.

3.2 Probabilistic de Bruijn Graphs

To better scale assembly, we used a Bloom filter[3] data structure to store k-mers originating from next-generation sequencing datasets. A Bloom filter is a hash-style probabilistic data structure used for storing sets. In the khmer software package, the Bloom filter is implemented as multiple hash tables with each having a different hash function. To add an element in the data structure, the corresponding bin in each hash table where the k-mer maps is set to 1. To query whether a k-mer is present or not, one must find a 1 in each bin for the k-mer. If there are one or more 0s, then the k-mer is definitely absent from the set. However, if we find all 1s, then the k-mer might be present. Thus, while false positives are a possibility, false negatives are not. It is possible to tune the false positive rate by optimizing the number of hash tables used depending on the expected number of k-mers to be inserted and the total amount of memory that one is willing to allocate for the Bloom filter.

To load the Bloom filter, we step through each read and break it up into k-mers and insert each into the Bloom filter. After loading the set of reads, it is possible to traverse the de Bruijn graph by checking for the presence of each of the up to eight possible neighbor k-mers (that share a $k - 1$ overlap). Due to the one-sided error caused by using a Bloom filter, we call this graph a probabilistic de Bruijn graph, an example of which is illustrated in Figure 3.1. If we use an ancillary data structure that tags at least one k-mer to each read, we can employ a graph traversal algorithm to discover which reads transitively connect to one another by finding components in the de Bruijn graph.

It is important to note that the data structure used in khmer is not a Bloom filter by definition. A classic Bloom filter consists of one hash table with k different hash functions. Each hash function is assumed to be independent as this keeps the false positive rate as

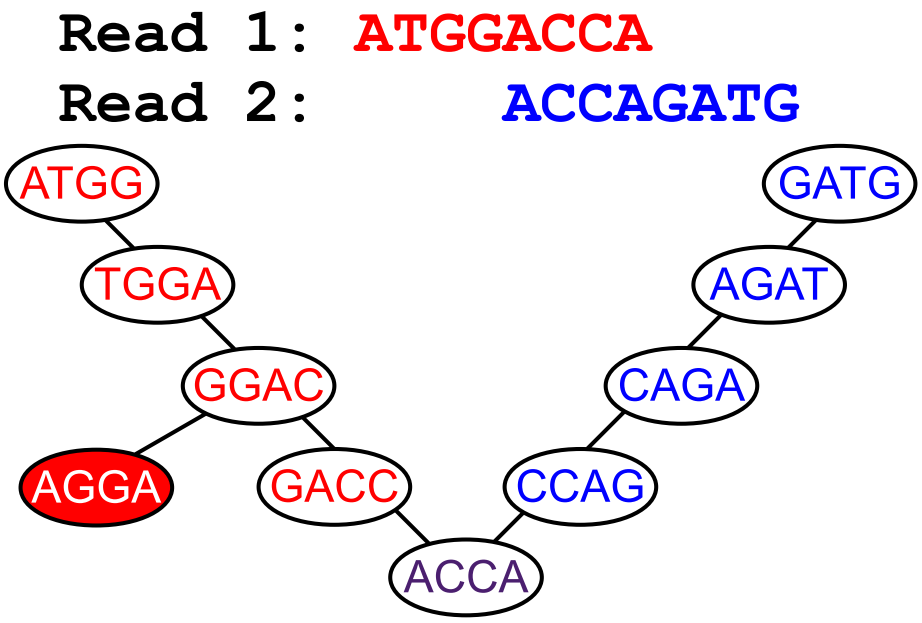


Figure 3.1: Example of a simple probabilistic de Bruijn graph with k=4, consisting of two reads. Due to the false positive rate, an erroneous k-mer, AGGA, exists though it doesn't exist in the set of reads. For interpretation of the references to color in this and all other figures, the reader is referred to the electronic version of this dissertation.

low as possible. In contrast, the data structure in khmer consists of k different hash tables with a different hash function in each. Furthermore, the hash functions used in khmer are not necessarily independent. Figure 3.2 demonstrates the difference between the two data structures where two k-mers are inserted into a classical Bloom filter (right side) and the Bloom filter in khmer (left side). Both require three different hash functions each (in this particular example).

Currently, khmer uses prime numbers above the given minimum hash table size parameter as the size for each hash table. The hash function is then the modulo of that prime with the number a k-mer is converted to. By mapping A to 00, C to 01, G to 10 and T to 11, we can convert the k-mer to its binary number form. If we count the reverse complement of the k-mer as the same, we can obtain both numbers and choose whichever is smaller.

We will show two ways that the Bloom filter variant data structure used in khmer has

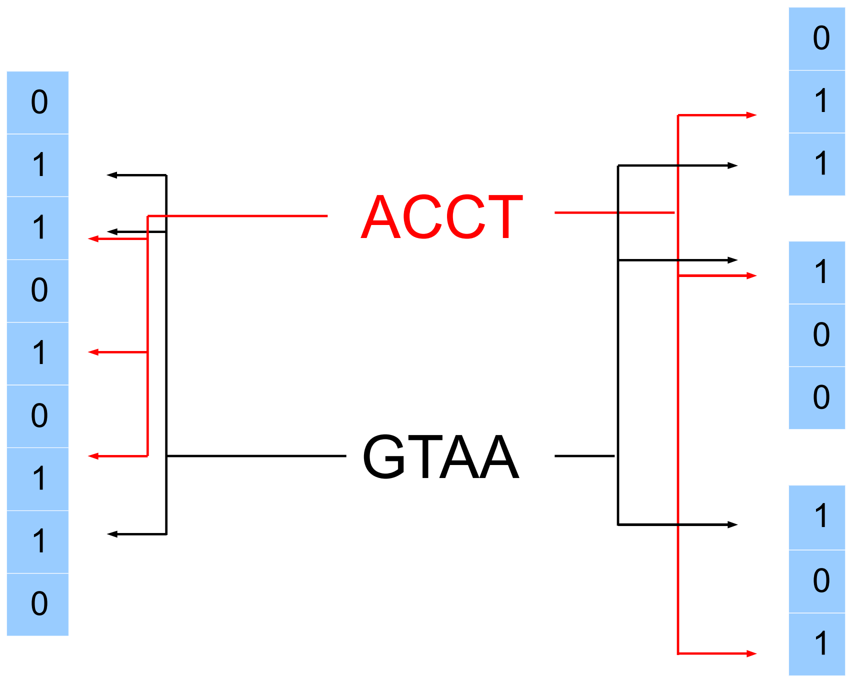


Figure 3.2: Illustration between a classical Bloom filter (left side) and the Bloom filter in the khmer software package (right side).

similar properties to the theoretical properties of a classical Bloom filter. First, we will compare the simple analytic derivatives that show they are equivalent (assuming independent hash functions) and show that the optimal number of hash tables is equivalent to the optimal number of hash functions for a classical Bloom filter.

For the Bloom filter variant in khmer, we will begin by considering the probability that a particular bin in one of the hash tables is occupied or not. First, let us assume that the size of each hash table equivalent, and let m be the total amount of memory allocated to the hash tables, k be the number of hash tables, h be the size of each hash table, and n be the number of k-mers inserted into the data structure. Furthermore, we assume that each hash function is independent of one another.

Examining one hash table, the probability that a bin is occupied after one k-mer is inserted is

$$\frac{1}{h}. \tag{3.1}$$

Thus, the probability that a bin is unoccupied after a single insertion is

$$\left(1 - \frac{1}{h}\right). \tag{3.2}$$

Extending this to n insertions, we see that the probability that a bin is unoccupied after n insertions is

$$\left(1 - \frac{1}{h}\right)^n. \tag{3.3}$$

Thus, we find that the false positive rate for a single hash table to be

$$1 - \left(1 - \frac{1}{h}\right)^n. \tag{3.4}$$

Since the data structure has k hash tables, we find that the false positive rate for the given variables is

$$\left(1 - \left(1 - \frac{1}{h}\right)^n\right)^k. \tag{3.5}$$

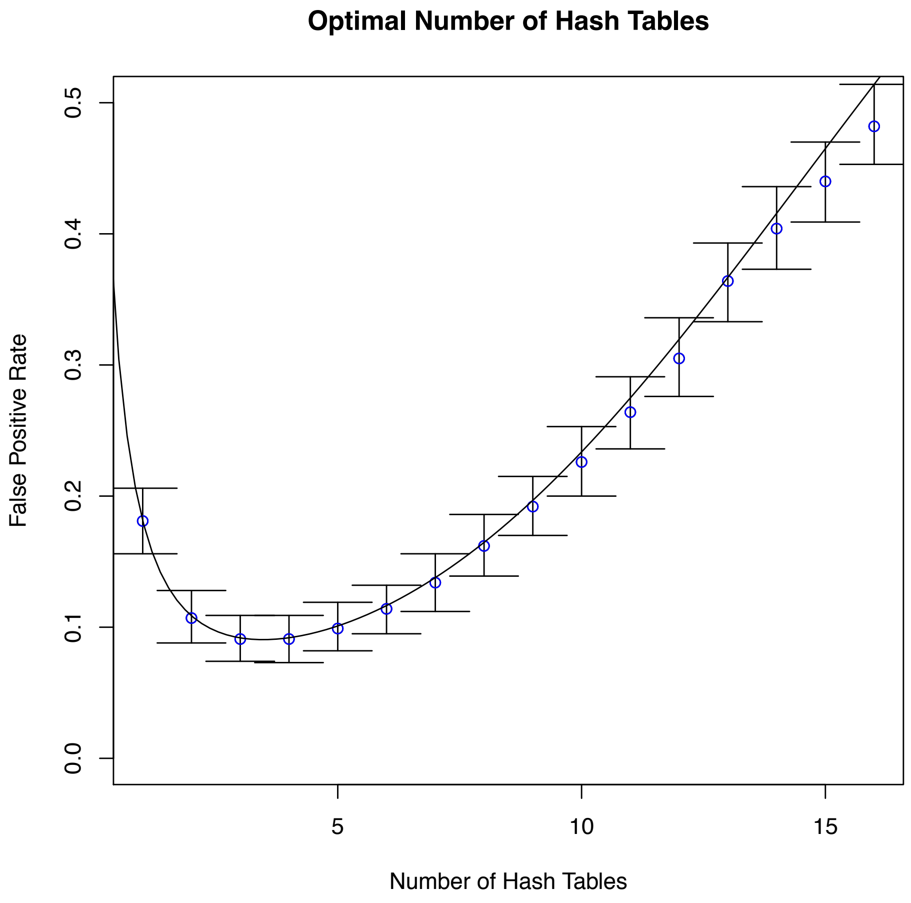


Figure 3.3: Comparison between the optimal number of hash tables in khmer with the theoretical optimal for Bloom filters.

Since $(1 - \frac{1}{h})^n$ approximates to $e^{-\dfrac{n}{h}}$, we obtain

$$\left(1 - e^{-\frac{n}{h}}\right)^k . \tag{3.6}$$

The false positive rate for a classical Bloom filter is obtained in a similar manner, and given the same variables is $(1 - e^{-\dfrac{kn}{m}})^k$[14]. Since $m = kh$, we see that the false positive rate properties for both data structures are analytically the same.

While the analytic derivation matches, it makes the critical assumption that the Bloom

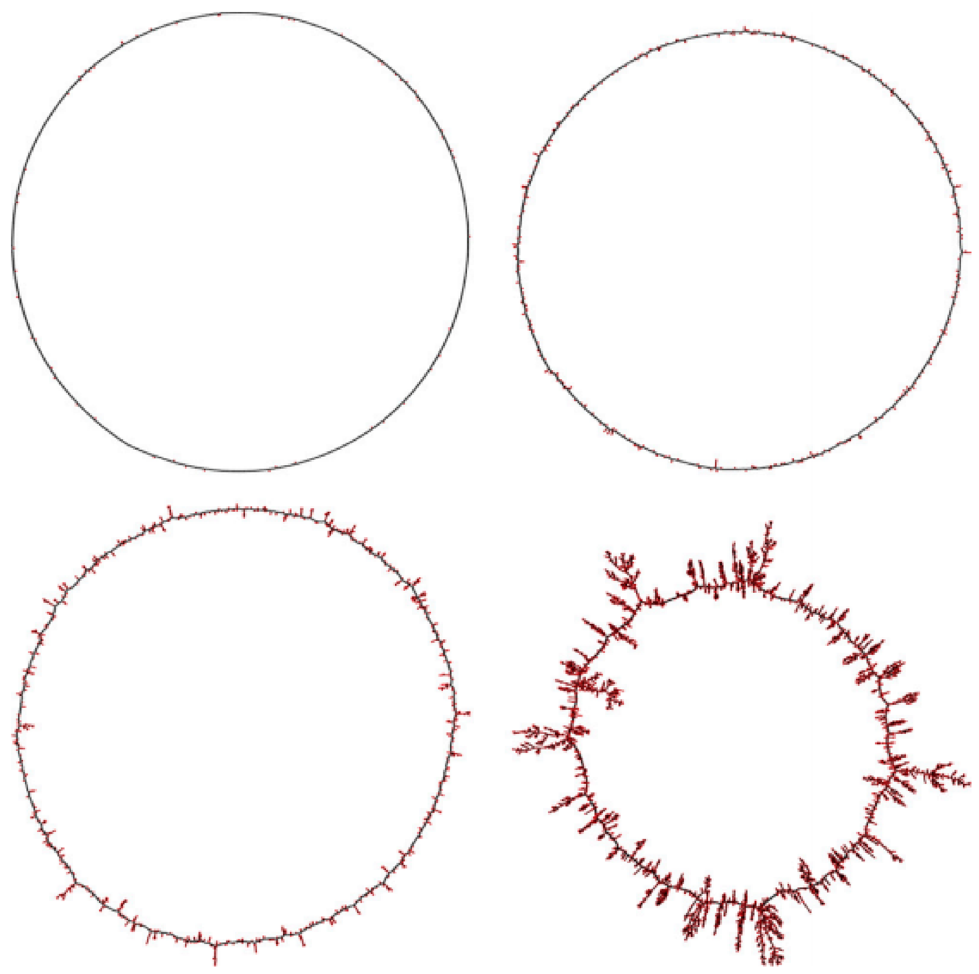


Figure 3.4: Graph visualization of a circular chromosome at four different false positive rates (1%, 5%, 10%, and 15%).

filter used in khmer has independent hash functions. Thus, it is important to compare the theoretical properties with the actual properties in the software implementation. Given a fixed amount of memory we allocate to our Bloom filter-variant, we can vary the number of hash tables and determine the false positive rate for each. As Figure 3.3 shows, the false positive rate given a various number of hash tables matches the theoretical number of hash functions for a classical Bloom filter for the given dataset. Given that our variant matches the theoretical properties of a classical Bloom filter, we will henceforth refer to the data structure in khmer as a Bloom filter.

3.3 Percolation of DNA de Bruijn graphs

Intuitively, it is clear that traversing a graph at a high enough false positive would result in erroneous k-mers connecting together in such a way that false paths could connect components in the graph. We visualized this effect by inserting a randomly generated 1,000bp circular chromosome into a Bloom filter with four different false positives. By starting at the first k-mer and then exploring the graph using the Bloom filter, we can see the effects of false positives as the false positive rate grows larger (Figure 3.4). Figure 3.5 shows as the false positive rate increases linearly, the average number of clusters appears to rise non-linearly and approaches a vertical asymptote. As we approach this asymptote, it becomes computationally intractable to calculate average cluster sizes for large k because the number of possible k-mers is so large and false positives erroneously connect clusters together.

The problem of determining when clusters begin to connect together has a strong theoretical framework called percolation theory. In graphs, the percolation threshold is the point at which a unique, giant cluster exists. In infinite graphs, this cluster is infinitely-sized. For finite graphs such as de Bruijn graphs, the giant cluster is of a size proportional to the number of vertices in the graph, whereas below the percolation threshold the largest clusters are of size proportional to the logarithm of the number of vertices. For a particular graph or lattice, there are actually two percolation problems: site and bond. In site percolation, the vertices are randomly "on" with a probability p. If an edge is shared between two vertices that are "on," then they are considered to be part of the same cluster. In bond percolation, the edges are randomly "on" with probability p.

The most well-studied lattice for percolation analysis is the 2D lattice (Figure 3.6), which consists of an infinite number of vertices in a grid layout. Thus, each vertex has an adjacent

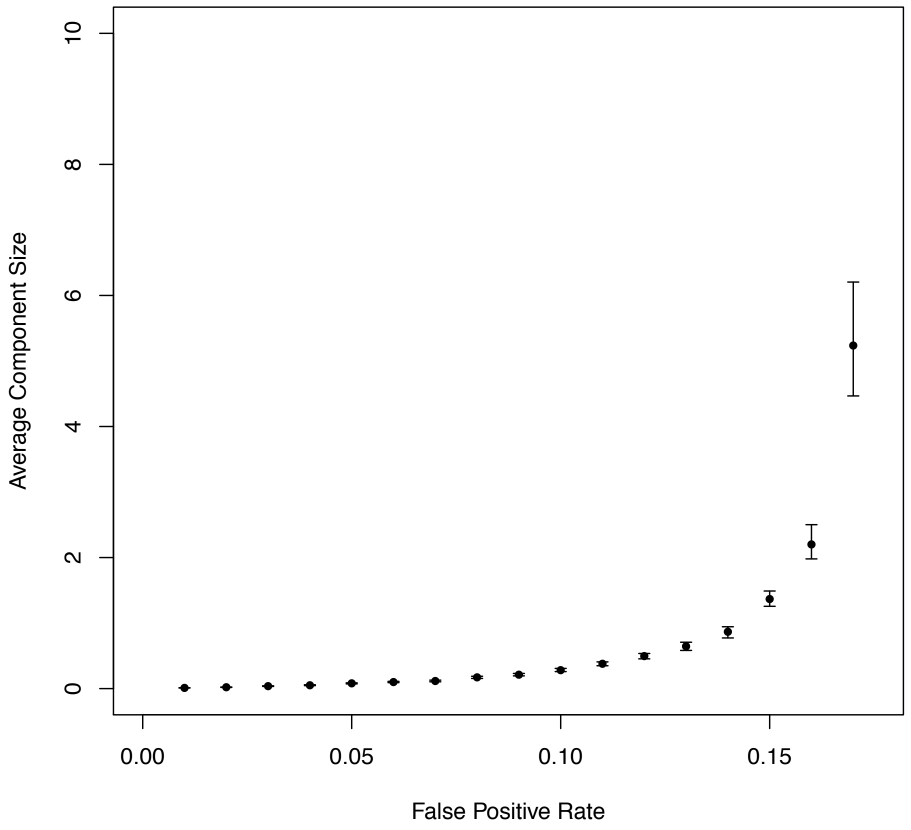


Figure 3.5: Average cluster size versus false positive rate. The average cluster appears to approach a vertical asymptote.

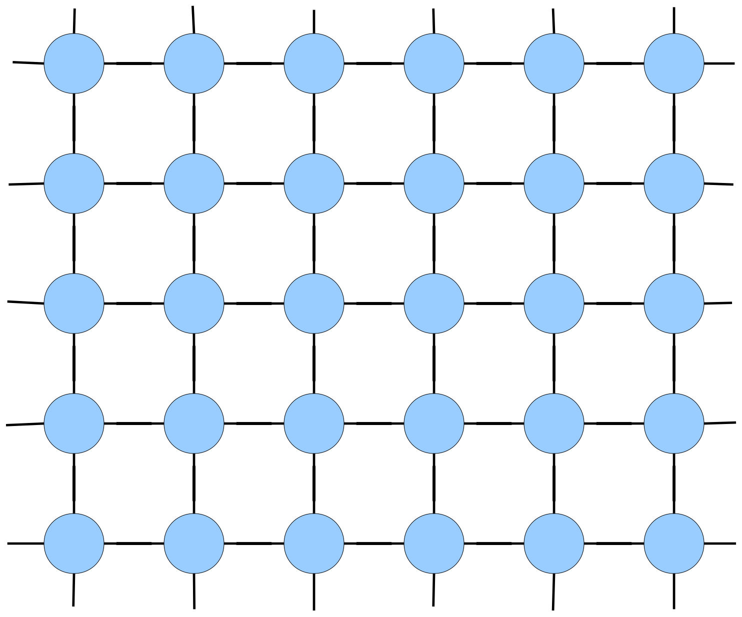


Figure 3.6: A 2D Lattice.

vertex above, below, to the left, and to the right. Though analytic solutions have been attempted for both the site percolation threshold and the bond percolation threshold, only the bond percolation threshold has been proven to be 0.5[24]. Through simulation, the site percolation threshold for 2D lattices has been found to be around .59[15]. Simulating either the site or bond percolation threshold for 2D lattices is considerably easier than DNA de Bruijn graphs because of the structures between both graphs. An effective way to find the percolation threshold for 2D lattices is to create a large, finite-sized grid and find at what p there exists a path from the left border to the right border. This calculation is made easier by using a union-find data structure to determine whether two vertices are part of the same component. However, de Bruijn graphs do not appear to have natural boundaries that can be easily tested. In order to test the DNA de Bruijn site percolation threshold, a different type of simulation must be used.

The site percolation threshold can be estimated by examining the cluster size distribution as vertices are randomly flipped on. As p increases, the distribution is fitted using both linear and quadratic regression. The percolation threshold is found by finding the point where quadratic regression fits better than a linear one due to the appearance of the giant component[48]. We calculate this by using the F-statistic

$$F = \frac{RSS_1 - RSS_2}{q_2 - q_1} \times \frac{n - q_2}{RSS_2}$$

where RSS_i is the residual sum of squares for a model i, q_i is the number of parameters for model i, and n is the number of simulations run for a particular probability. The percolation threshold, p_c, is found by finding the second local maxima. In order to account for finite sampling error, we employ the threshold binning method[1]. Using this technique, we found that the percolation threshold for DNA de Bruijn graphs is approximately 0.183.

3.4 Effect of False Positives on Long-Range Graph Connectivity

Though the percolation threshold results suggest that components in the graph will not fuse together below a specific percolation threshold, we wanted to explore how false positives affect long-range graph structure. To do this, we examined how the diameter, or "longest shortest path" in the graph changes as the false positive rate increases. Because we wanted to find when long-range connections occur beyond the percolation threshold, it is computationally difficult to use a large k. To remedy this, we simulated 50bp circular chromosomes at $k = 8$

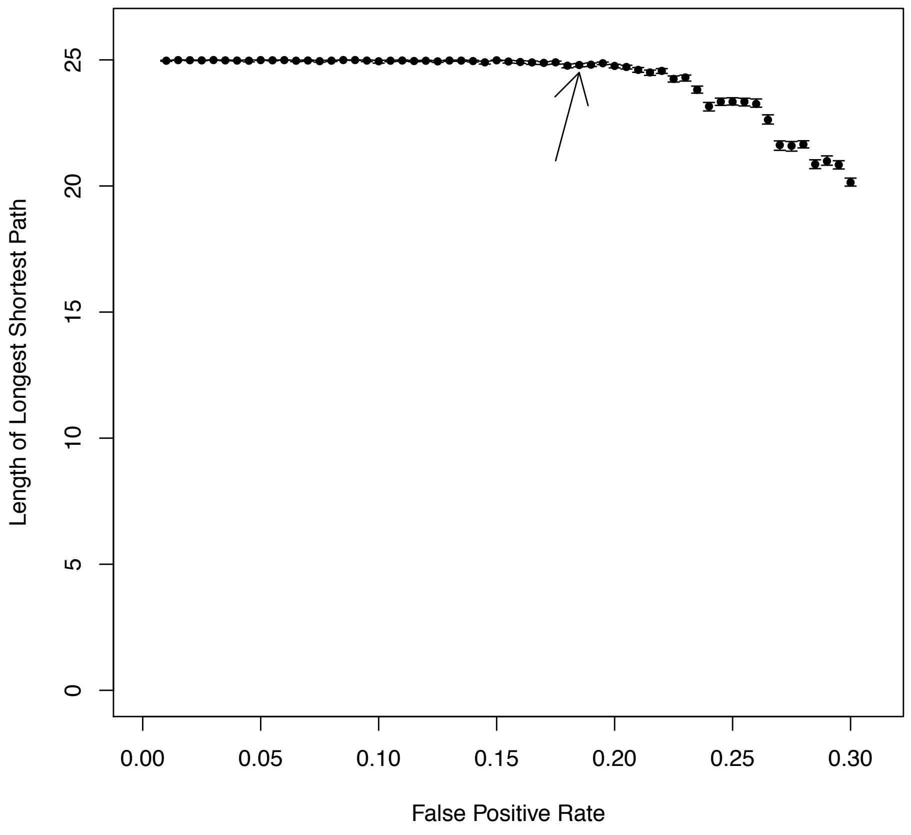


Figure 3.7: The diameter (i.e. longest shortest path) decreases abruptly after the percolation threshold, suggesting long-range connections within the graph structure.

(i.e. 58bp reads with the last 8bp equivalent to the first 8) and then calculated the shortest path between each pair of true k-mers with the longest of these being the diameter. The experiment was repeated 500 times for each false positive rate. From these experiments, we conclude that long-range false connections do not occur until at or beyond the percolation threshold (Figure 3.7).

3.5 Partitioning a Metagenomic Dataset

Using the probabilistic de Bruijn graph, it is possible to separate the graph into components that are likely to assemble together, which is called read partitioning. Using a probabilistic de Bruijn graph, we are able to partition the reads with far less memory than with exact (i.e. non-probabilistic data structures) approaches as long as we remain below the percolation threshold.

The partitioning algorithm works as follows. A set of reads is inserted into a Bloom filter. When each read is inserted, at least one k-mer is tagged in a C++ STL map so that the k-mer is mapped to a read ID. The next step is to perform a local traversal of the probabilistic de Bruijn graph based on the distance between tagged k-mers in the reads. Tagging few k-mers requires less memory, but more computation to traverse through a larger portion of the graph. However, doing a local traversal only captures the reads that share k-mers with a particular read. To determine which reads belong together in the same graph component, a merging step occurs where subsets are merged together. Finally, these separate sets of reads (i.e. partitions), can be assembled separately on different nodes to reduce memory requirements.

One concern with the partitioning algorithm is how it performs as the false positive rate increases. Though we determined a specific percolation threshold, this only applies when vertices are randomly turned on and off. However, this clearly does not apply with genomic DNA since large chromosomes will consist of large components with many k-mers. If two chromosomes share a single k-mer, they will be part of the same graph component. This may not present a problem when this happens infrequently, but if the false positive rate is high enough, erroneous connections could occur in the graph if false paths are introduced

Table 3.1: Partitioning results on a soil metagenome at k=31.

False positive rate	Total memory use (improvement)	Largest partition size in reads
1%	1.75GB (18.8x)	344,426
5%	1.20GB (27.5x)	344,426
10%	0.96GB (34.37x)	344,426
15%	0.83GB (39.75x)	344,426

between components. To test whether or not this occurs, we randomly generated 1,000 DNA sequences 1,000 base pairs long. Varying the false positive rate with k=31, we used the partitioning algorithm to determine the resulting number of partitions. Though the time to partition increases in a similar fashion to the expected average cluster size, the number of partitions does not change (Figure 3.8).

Using a small soil metagenomic dataset, we demonstrated the memory efficiency and utility of the partitioning algorithm in the khmer software package. To do this, we utilized a small script called memusg (available at `https://gist.github.com/netj/526585`) that checked the memory usage every 0.1 seconds. With all of the partitioning and assembly steps added to an sh script, the memusg script reported the highest memory measurement. Assembling the entire dataset without partitioning requires approximately 33 GB of RAM, but partitioning at a 15% false positive rate requires almost 40 times less memory. The memory requirements for partitioning and assembly are roughly linear and match the per k estimates (Table 3.1).

3.6 Sequencing Errors Outnumber False Positives

No sequencing platform produces error-free reads, and we explored how the number of erroneous k-mers resulting from sequencing errors compared to the erroneous k-mers that resulted from the false positives in a Bloom filter. We used two different datasets, the *E.*

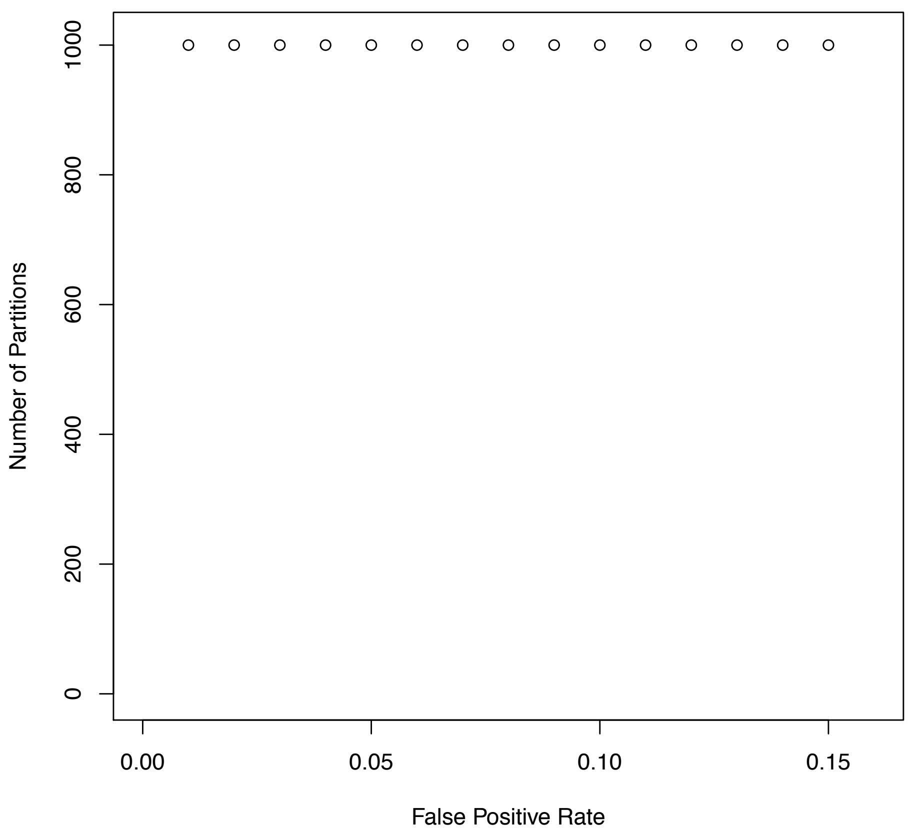


Figure 3.8: After partitioning a simulated dataset at increasing false positive rates up to 15%, the number of partitions found does not change. Closer to the percolation threshold, the partitioning process becomes more difficult computationally.

Table 3.2: Effects of loading *E. coli* data at different false positive rates

Graph	Total k-mers	False connected k-mers	% Real	Deg > 2	Mem (bytes)
E. coli at 0%	4,530,123	0	100	50,605	2.1×10^9
E. coli at 1%	4,814,050	283,927	94.1	313,844	5.4×10^6
E. coli at 5%	6,349,301	1,819,178	71.3	1,339,102	3.5×10^6
E. coli at 15%	31,109,523	26,579,400	14.6	10,522,482	2.2×10^6
Reads at 0%	45,566,033	41,036,029	9.9	7,699,309	2.1×10^9
Reads at 1%	48,182,271	43,652,265	9.4	31,600,208	5.4×10^7
Reads at 5%	62,019,545	57,489,537	7.3	42,642,203	3.6×10^7
Reads at 15%	231,687,063	227,157,037	1.9	113,489,902	2.3×10^7

coli genome as well as a set of Illumina GA-II reads that were obtained from sequencing the same strain of *E. coli.* We loaded both the reads and the genome into separate "exact" Bloom filters with K=17. It is exact in the sense that there is a bin for each possible 17-mer, so no collisions are possible. We then loaded each of the datasets into Bloom filters with 3 different false positive rates (1%, 5%, and 15%). As Table 3.2 shows, even with no false positives, the number of erroneous k-mers is almost 10x larger than the number of real k-mers. Inserting the *E. coli* genome into a Bloom filter with a 1% false positive rate yields approximately 5.9% of the k-mers as being false positives. However, it should be noted that as the false positive rate approaches the percolation threshold, the percentage of real k-mers falls dramatically. Another important comparison is between the number of k-mers with a degree greater than 2. This metric is used as a proxy for graph complexity. When reads are loaded into an exact Bloom filter, there are 7.7 million k-mers with 3 or more degrees. On the other hand, storing the genome with a 1% false positive rate yields only 313,844 false k-mers, implying that for low false positive rates, the erroneous k-mers generated from sequencing errors yields a comparatively more complex graph than false positive k-mers.

3.7 Memory Efficiency vs. Exact Data Structures

Assuming that the percolation threshold for DNA de Bruijn graphs is .183, we can compress an assembly graph down to approximately 4 bits per k-mer, independent of the size of k. Given a set of n total possible items and x items to be inserted, it is possible to represent the set in $log_2\binom{n}{x}$ total bits. This problem has been considered in the context of DNA assembly graphs previously[10]. While many efforts have been undertaken to create a succinct representation of these types of datasets that match the lossless informatic-theoretic lower bound, none have solved this problem in the general case. Regardless, a probabilistic data structure such as a Bloom filter or Counting Bloom Filter can push below this lower bound. Using the equation above, we compared the Bloom filters with four different false positive rates (1%, 5%, 10%, and 15%) with the information-theoretic lossless lower bound. Figure 3.9 shows that for sparsely populated k-mer bitmaps (typical of any sequencing dataset with a reasonably high k), the Bloom filters have better memory efficiency than any exact data structure. It is also important to note that the lossless lower bound is dependent on the size of k chosen: when a larger k value is chosen, there is a larger number of possible k-mers that could be part of the set. However, a Bloom filter is independent of k, but begins to lose its advantage over lossless data structures as the set becomes more dense. Intuitively, this makes sense because when a bitmap is used to represent a set, it is impossible to further compress when half of the bits are randomly occupied.

3.8 K-mer Counting with Counting Bloom Filters

One limitation to Bloom filters is that they can only provide the likely presence or certain absence of a particular item in a set. Since many genomics analyses use coverage information

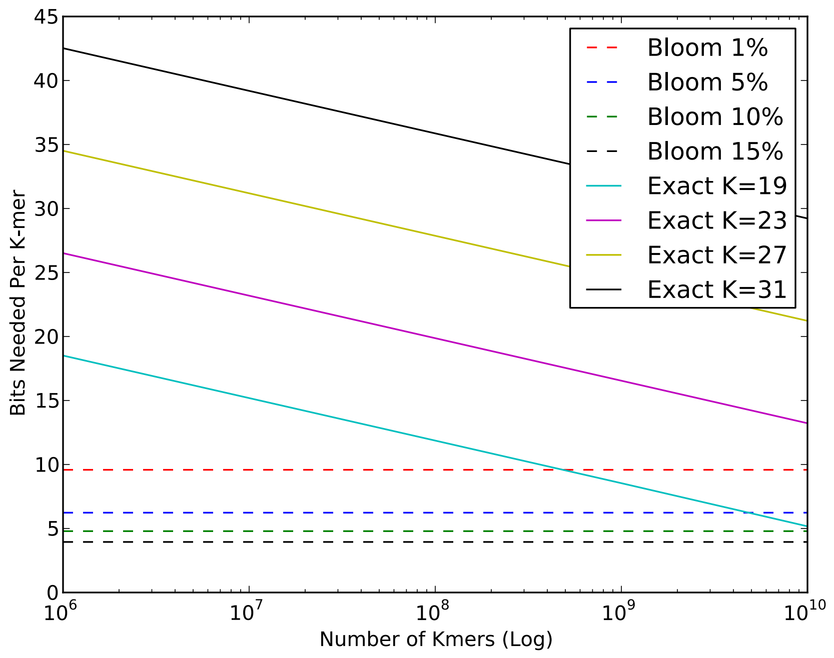


Figure 3.9: Comparison between a Bloom filter at four different false positives with the information-theoretic lower bound for four values of k.

to help find erroneous, repetitive, or other types of sequence, it is sometimes necessary to use a data structure that can store k-mer counts rather than simply presence or absence. A natural extension of the Bloom filter is to increase the bin size from 1 bit to the largest count one would like to store (e.g. 8 bits would allow one to count from 0 to 255). Then, to obtain the count for a particular k-mer, we look up the count for that particular k-mer in each hash table and take the minimum value. As with Bloom filters, false negatives are not possible, but false positives are. However, for k-mer counting there is yet another one-sided error that must be considered. That is, if a k-mer exists in the dataset, we will correctly see that it is present, but its count may be over-inflated. This only occurs in the case when each bin that corresponds to the k-mer of interest collides with another k-mer. This type of error depends largely on the count distribution as well as the false positive rate.

3.9 Implications for Non-Probabilistic de Bruijn Graphs

In this chapter, we examined the false positive rate at which point components in a probabilistic de Bruijn graph begin to erroneously connect together and form a giant cluster. However, the idea of percolation can further be considered for non-probabilistic graphs. When choosing a low k value, one must consider that if the occupancy rate of possible vertices is above the percolation threshold, then it is highly likely that the graph is already percolated and that most of the reads will belong to one large cluster. This, of course, makes assembly more difficult and should be taken into consideration when selecting a value for the k parameter of assemblers. Given that there are 4^k possible k-mers for a selected k value (without taking into account reverse complements), this becomes less of an issue as k is greater than 25. On the other hand, selecting too high a value for k will lower the sensitivity since fewer overlaps will be found between k-mers. Furthermore, even for non-probabilistic de Bruijn graphs, it is still possible that components will share different species or chromosomes even if they are not sampled from the same place. The classic Birthday Problem suggests that even for higher k values, say between 30 and 40, it is still possible that these types of problems can occur.

3.10 Data

3.10.1 Genomic Data

The data used to demonstrate the effect of false positives compared to sequencing errors in a de Bruijn graph was the *E. coli* K-12 MG1655 genome, which is available on NCBI under accession number NC_000913.2. The Illumina reads datasets that were used for comparison were a 200bp insert library (SRA0016655) and a 500bp insert library (SRA001666). Both were sequenced using the GA-II platform.

3.10.2 Metagenomic Data

The dataset used for the demonstration of the partitioning algorithm and its memory efficiency was a small soil metagenome sample named MSB2. It was sequenced on one lane of Illumina's GA-II platform. It contains approximately 35 million reads and totals around 2.8 Gbp. Since soil is so highly diverse, this dataset does not contain near enough coverage to assemble the entire metagenome. The dataset exists in NCBI's Short Read Archive under accession number SRA050710.

Chapter 4

Read Error Correction

Though assembling a set of reads together is challenging, it is even more difficult due to the errors present in reads. Rather than detecting exact overlaps, one must develop algorithms that are able to efficiently find inexact overlaps between reads. In the context of a de Bruijn graph, error k-mers greatly complicate assembly. A single base pair error in a sequence read can create up to k erroneous k-mers. Given a greater than 1% error rate, it is clear that erroneous k-mers generate the bulk of memory requirements for assemblers when not properly handled. Furthermore, they can make the graph so complex in certain regions that it is impossible to traverse the region in the graph. This is particularly troublesome in repetitive sequence. Due to the problems that errors create in assembly, read error correctors have become an ubiquitous part of assembly pipelines.

In this chapter, we introduce a two-pass algorithm for error correcting metagenomic sequencing datasets. Currently, few options exist for error correcting metagenomics datasets. Many effective algorithms exist for correcting genomic reads, but these generally rely on a more uniform coverage level across the sample. Because metagenomes can contain millions of species of varying abundance levels, it is difficult to use k-mer abundance information without conducting an additional round of filtering. We take advantage of the k-mer abundance distribution that results from applying the digital normalization algorithm[6] to a dataset to error correct metagenome datasets.

4.1 K-mer Abundance Distributions

Before we proceed with an implementation of a read error correction algorithm, we must first consider the k-mer abundance distribution for reads containing erroneously called bases if we incorporate such information in the algorithm. However, we must first make a number of unrealistic assumptions to make an analysis feasible. Such assumptions are not unfounded, and we will make similar ones used by Lander and Waterman in their paper outlining the statistics behind shotgun sequencing[25].

One assumption we will make is that that there are no repetitive or low complexity regions in the genome such that every k-mer in the genome is unique. This is clearly false as not only will k-mers in different locations of the genome be the same by random chance for lower k values, but repetitive and low complexity regions will mean that even for large k values there will be many identical k-mers. Another assumption is that sequencing errors occur uniformly across a read. This, too, is an unrealistic assumption as sequencers tend to have different error profiles. In Illumina, it is well known that most errors occur on the ends of reads with the majority happening on the 3' end. We will also assume that indels do not occur. These are relatively uncommon compared to SNPs in Illumina reads, and have a similar effect to them in the k-mer abundance distribution. However, by not including them, we can assume the length of each read is the same. The last assumption we will make is that when a sequencing error occurs, the new erroneous k-mers generated will be distinct. This simplifies calculating the k-mer abundance distribution with sequencing errors since any erroneous k-mer will be given a count of 1.

4.1.1 K-mer Abundance Distribution of One Genome

Let G be the number of base pairs in a circular genome. Since it is circular, G is also the number of k-mers where k is the chosen word size. Let e_r be the probability that a particular base in a sequence read is erroneous. For a given sequence read, let R be the length of the read in bases, and let $M = R - K + 1$ be the number of k-mers in a read. Lastly, let N be the number of reads in a dataset.

To determine the k-mer abundance distribution, let us consider the probability that a particular k-mer occurs in c reads without any errors. First, we will consider the probability that a k-mer occurs in a particular read. Since a sequence read contains M k-mers, the probability is $\frac{M}{G}$. The probability that the k-mer does not exist in the read is then $\frac{G-M}{G}$. Putting this together, the probability that a k-mer occurs in c reads without any errors is

$$\binom{N}{c}\left(\frac{M}{G}\right)^{c}\left(\frac{G-M}{G}\right)^{N-c}. \tag{4.1}$$

This means that for a genome of size G, the k-mer abundance distribution is

$$G \cdot \binom{N}{c}\left(\frac{M}{G}\right)^{c}\left(\frac{G-M}{G}\right)^{N-c}. \tag{4.2}$$

Now we will incorporate errors into the distribution. The probability that a k-mer in a read is error-free is $(1 - e_r)^k$. This implies that the probability that a k-mer contains at least one error (and is thus an error k-mer) is $(1-(1-e_r)^k)$. Thus, the expected number of correct k-mers per read is $M(1 - e_r)^k$ and the number of error k-mers is $M(1-(1-e_r)^k)$.

To simplify the notation, let $L = (M(1 - e_r)^k)$. We can then replace M with L from the previous equation. Due to our assumption that all error k-mers are unique, we find that the piece-wise function for the k-mer abundance distribution for a single circular genome with sequencing errors is

$$F(G, R, e_r, k, c) = \begin{cases} G \cdot \binom{N}{c} \left(\frac{L}{G}\right)^c \left(\frac{G-L}{G}\right)^{N-c} + M(1(1-e_r)^k), & \text{if } c = 1 \\ G \cdot \binom{N}{c} \left(\frac{L}{G}\right)^c \left(\frac{G-L}{G}\right)^{N-c}, & \text{otherwise.} \end{cases} \tag{4.3}$$

Due to the presence of the binomial distribution, this function simplifies to

$$F(G, r, e_r, k, c) = \begin{cases} G \cdot \operatorname{bin}(c, N, \frac{L}{G}) + M(1(1-e_r)^k), & \text{if } c = 1 \\ G \cdot \operatorname{bin}(c, N, \frac{L}{G}), & \text{otherwise} \end{cases}. \tag{4.4}$$

4.1.2 Digital Normalization Convergence

An important issue when applying the digital normalization algorithm to a dataset is if the k-mer coverage converges towards a specific value (ideally, the C parameter). If convergence can be expected, then it can be useful for predicting when a dataset has saturated or if more novelty can be expected. For the case where the length of the read is equal to K, it is simple to show that the k-mer coverage approaches C, the parameter for digital normalization. We define k-mer coverage to be the expected number of reads in a dataset that “contains” a particular k-mer.

Intuitively, we can see that this is the case since the genome will continue to be sampled until each possible k-mer has a count of C. We can also consider the probability mass function for the k-mer abundance distribution in this special case given the size of a genome in k-mers (G), the number of reads processed so far (N), and the digital normalization parameter (C), To do this we must consider three cases: where the k-mer count is less than C, the k-mer count is currently equal to C, and the count is greater than C. In the last case, we know that the number of k-mers having a count greater than C must be zero because if a read equal to the size of a k-mer is introduced that is of size C, it will be discarded. In the first case, the count will be the same as for the binomial case, so any k-mer that does not have a count of less than C must be equal to C. Putting this all together, we obtain

$$pmf(N, C, G, X) = \begin{cases} G \cdot \binom{N}{X} \left(\frac{1}{G}\right)^X \left(\frac{G-1}{G}\right)^{N-X}, & \text{if } X < C \\ G \cdot \left(1 - \sum_{i=0}^{C-1} \binom{N}{i} \left(\frac{1}{G}\right)^i \left(\frac{G-1}{G}\right)^{N-i}\right), & \text{if } X = C \\ 0, & \text{if } X > C \end{cases} \tag{4.5}$$

Furthermore, we can use the probability mass function to derive the number of reads that are kept by the digital normalization algorithm in the same case where the length of the read is K. Since no k-mer can have an abundance greater than C, we only consider adding a read (i.e. k-mer) when the count is less than C, which we see is

$$readskept(N, C, G) = \sum_{i=1}^{N} \sum_{j=0}^{C-1} \binom{i}{j} \left(\frac{1}{G}\right)^j \left(\frac{G-1}{G}\right)^{i-j}. \tag{4.6}$$

Though it is straight-forward to analyze the probability mass function and number of reads kept with the digital normalization in the case where there are no errors and the length of the read is equal to k, the analysis becomes considerably more complicated when extended to the length of the read. The most challenging part of this analysis is the use of the median function in the algorithm. To determine the count of a particular k-mer after N reads have been processed, one must consider the ordering of the reads that covered a particular k-mer. Each of these reads may not have been retained because the median k-mer count of the read exceeded C. To do this analysis, one would need to take into account the median, rather than the expectation at each point in time. Since the digital normalization algorithm skews the original reads distribution, this complicates the analysis. Regardless, it is possible to evaluate the convergence of the algorithm by creating a simulation of the digital normalization algorithm. We simulated a set of 200,000 reads where each read contains 20 k-mers. Given a circular genome with 100,000 k-mers, we apply the digital normalization algorithm with $C = 10$. After processing the reads with the algorithm, we see that the mode of the k-mer abundance distribution is 10, but that some k-mers have an abundance greater than C due to how the reads are stacked along the genome as they are processed (Figure 4.1). Furthermore, the algorithm clearly causes the number of reads to converge to a specific value (Figure 4.2).

However, it is clear that there is no convergence towards the C parameter in the presence of errors. Considering the error model introduced earlier, there is a possibility that a read will contain only error k-mers as long as there is at least one base error every k or fewer bp in distance. Given an infinite stream of reads, this means that any possible read can be generated, and thus any k-mer. This implies that the data structure used would collect every possible k-mer, but the k-mer coverage would still end up around C (since the erroneous k-mers would dominate the expectation calculation at this point). However, because these

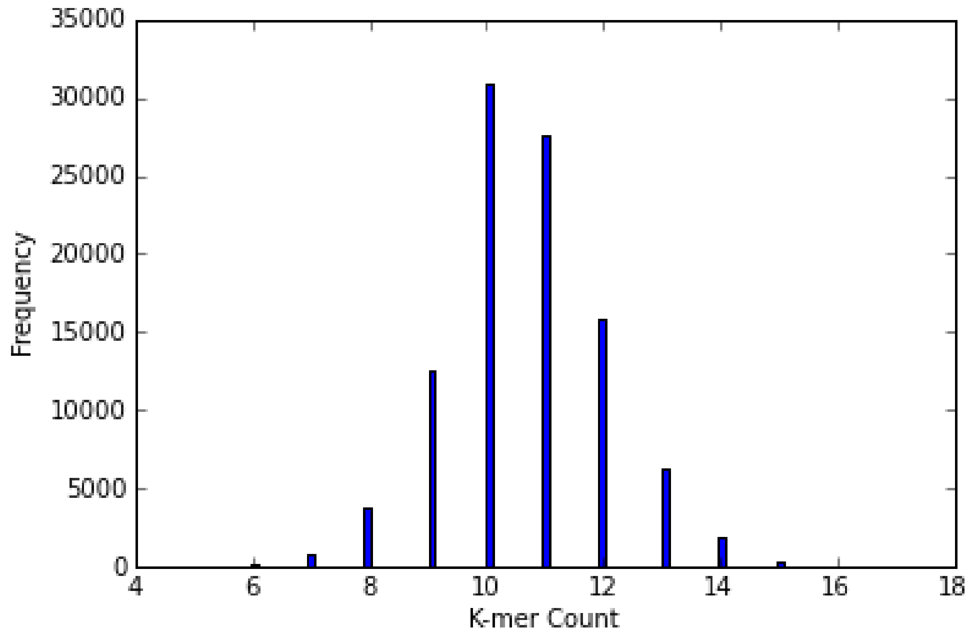


Figure 4.1: K-mer abundance distribution using simulated reads with 20 reads per k-mer, a circular genome size of 100,000 k-mers and 200,000 reads.

error k-mers would begin to match the coverage of trusted ones, the algorithm would need to terminate much sooner and would need to be modified heavily to form the basis for an online, streaming algorithm. Due to the use of a Counting Bloom filter, we instead must use a multi-pass approach that terminates well before these errors cause such an issue.

4.2 Error Correction Algorithm Components

Currently, few methods exist for reliably error correcting metagenomic reads due to the uneven abundances among "species" within an environmental sample and the deep sequencing required to sufficiently sample the least abundant organisms. As mentioned previously, several error correction methods such as Quake and what is used in the EULER assembler rely on the k-mer coverage distribution to estimate which k-mers can be trusted and which are

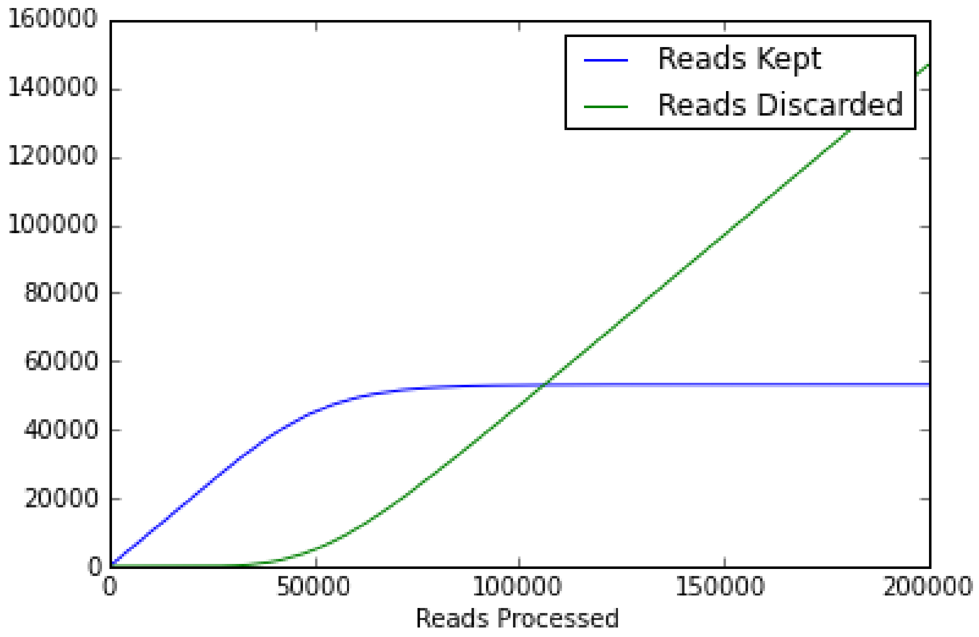


Figure 4.2: The number of reads kept and discarded as reads are processed through a simulation of the digital normalization algorithm in the same experiment as Figure 4.1.

likely erroneous. However, the k-mer coverage of metagenomic data is different from genomic datasets.

Digital normalization[6] can be used to even out the coverage level in a metagenomic sample. This results in a k-mer coverage distribution from which one can separate erroneous k-mers from non-erroneous. However, using previous methods would require multiple passes through the sequence dataset, which can be time consuming when there are potentially billions of reads. To remedy this, we will use an error correction method that requires using two passes through the dataset by combining elements of probabilistic de Bruijn graph traversal and alignment with the digital normalization algorithm.

The error correction algorithm applies digital normalization to the dataset of interest and uses a Counting Bloom filter as the underlying data structure for k-mer counting. As the algorithm walks through the reads, the k-mer abundance distribution is built up using the digital normalization algorithm as a filter. We will use a mixture model containing two Poisson distributions to discern between erroneous and non-erroneous k-mers, but better fitting distributions and models can be used in circumstances where the distribution is better characterized.

4.2.1 Read to Graph Alignment

In order to pull out the best candidate alignment from the graph based on a sequence read, we use a similar strategy to what is in Xander (Jordan Fish, personal communication). Given a Counting Bloom Filter with a word size of k, we start at the beginning of each sequence read until we reach a k-mer that exists in both the Counting Bloom Filter and the read. To avoid aligning from each k-mer, we do not align k-mers that exist in alignments that were previously obtained. We consider this k-mer to to be the starting vertex in an A* search algorithm. We maintain a closed set of nodes that have been explored as a set and an open set of nodes as a priority queue (implemented as a heap in C++'s STL library). For each vertex, we check if any of its possible neighbors exist in the Counting Bloom Filter. However, unlike with a regular de Bruijn graph, we must account for up to four insertions and a deletion vertex in addition to the one possible exact match vertex and up to three mismatch vertices. These vertices are added to the open set, and by employing the A* graph algorithm, we can find the optimal alignment between the read and the graph. It is important to note that for each non-end k-mer (i.e. k-mers that are not at the end of a read), we must do two traversals: one for the forward direction and one for the backward. Figure 4.3 demonstrates an example of a read containing one error being aligned to the graph, which contains both the erroneous k-mers of the read (with only a k-mer coverage of one) and the correct sequence where all of the k-mers have a coverage of either 19 or 20. The algorithm

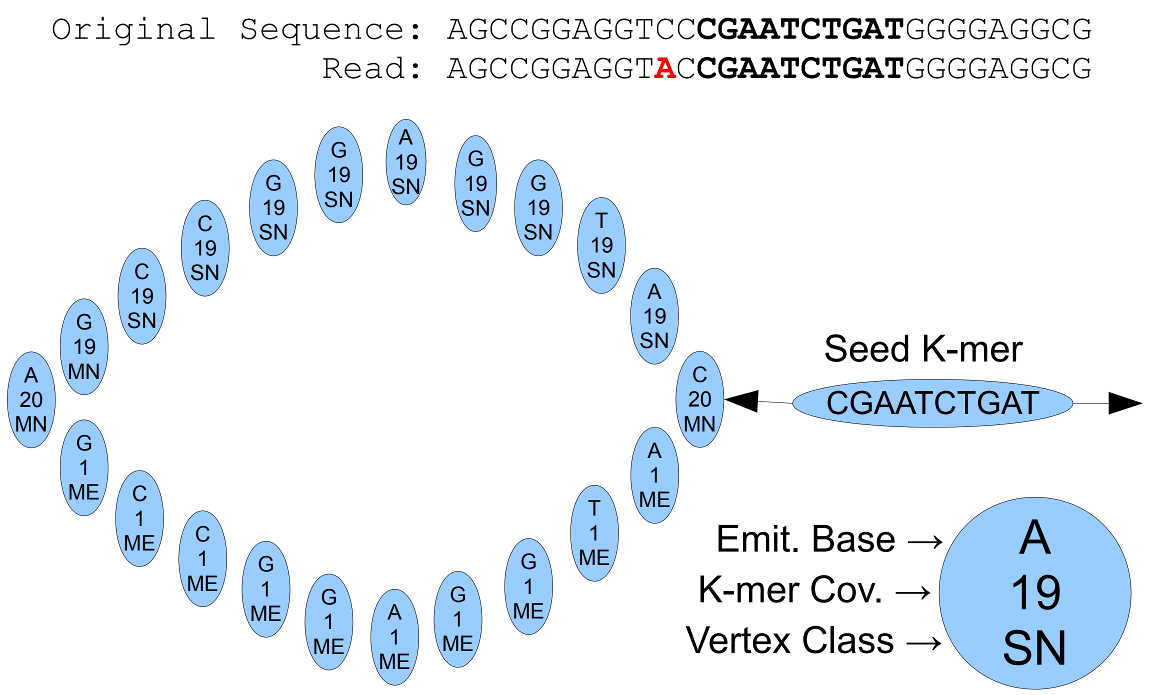


Figure 4.3: Simplified example of a graph alignment beginning at a seed k-mer. Two paths exists due to a single SNP, which forms a bubble structure. The goal is to correct the read by replacing the substituted SNP for the original base. The abbreviations for the vertex classes are as follows: MN = match and non-erroneous, ME = match and erroneous, SN = substitution and non-erroneous.

must prefer the path that is more likely to contain correct sequence as opposed to the path that more closely matches the read with erroneous k-mers.

To score the alignment, we assign scores for matches, substitutions, insertions, and deletions. However, due to the nature of the real biological sequence data, erroneous k-mers are certain to exist in the dataset. Rather than traverse down paths that contain error k-mers, we choose to stop traversal and not consider such error k-mers. We will discuss the method for determining whether or not a k-mer can be trusted later on. However, if the algorithm were to consider erroneous paths (e.g. to retain, rather than discard, error-prone reads), one could assign separate scores for erroneous matches, SNPs, and indels. These scores must

Table 4.1: Scores for error correction graph alignment.

Type	Score
Match	0
Substitution	8
Insertion	4
Deletion	4

necessarily be lower than any of the scores for correct k-mers that exist in the graph. The set of scores that were used for the experiments in this paper are listed in Table 4.1.

4.2.2 Two-pass Error Correction

To handle metagenomic data, we must first be able to successfully call an erroneous k-mer from a correct one. However, given errors in reads, the mode of the distribution is generally 1 with another peak occurring at the expected k-mer coverage. Figure 4.4 illustrates this effect with an *E. coli* reads sequencing dataset (see Data section). Due to the large number of error k-mers, the second peak is a small fraction of the size of the number k-mers of abundance one. Though this is typical, other types of sequencing datasets such as mRNAseq, MDA single cell, and metagenomic datasets are likely to have very different distributions from this. This highlights the importance of a pre-assembly filtering method that can effectively normalize the distribution. The problem is even worse with metagenomic dataset because with an ensemble of species in the sample, it is possible for no other easily discernible peaks to exist.

To ameliorate this issue, we employ the digital normalization algorithm[6] to even out the coverage level among different species. By limiting the k-mer coverage to a specific value, we know where to look for the second peak in the dataset. As long as all "species" in the sample were sequence to a higher coverage level than digital normalization's C parameter, we can

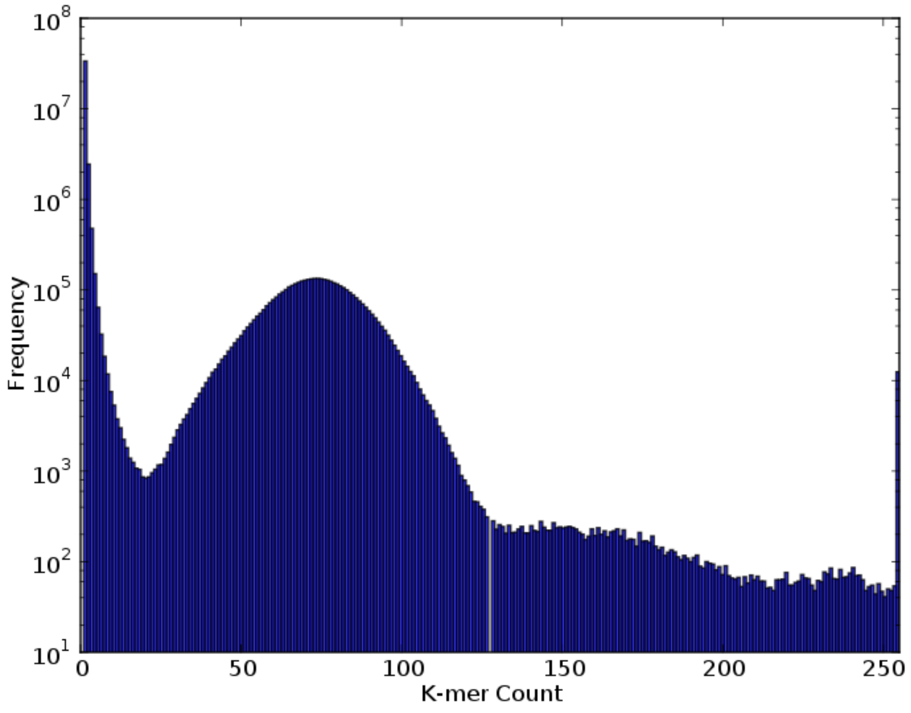


Figure 4.4: The raw k-mer abundance distribution on an *E. coli* reads dataset. Note that the y-axis uses logarithmic scaling.

retain the novelty from the sample while creating a peak from which we can discern trusted k-mers from ones likely to be erroneous. Figure 4.8 shows the effect of applying the digital normalization algorithm to the *E. coli* reads dataset. Clearly, a peak occurs around 20, which can be used to predict whether a particular k-mer is erroneous or correct. Knowing where the peak will be due to the C parameter is also useful because we do not need to keep track of the entire k-mer abundance distribution. Rather, we know that a peak at an abundance level of 1 due to error k-mers as well as a peak near C.

We will now outline the two-pass error correction algorithm. A two-pass algorithm means that the sequence reads dataset will be processed twice. In the first pass, we build a Counting Bloom filter while applying the digital normalization algorithm, which keeps any read where the median k-mer count is below C and discards any other read. For the purposes of error correction, the set of reads that are retained by the digital normalization algorithm are not needed. Instead, the Counting Bloom filter that is used to obtain the count is saved so that it can be used for the second pass of the algorithm.

In the second pass, we utilize the read to graph alignment strategy outlined earlier. To properly score the possible alignments, we must be able to determine whether a k-mer is likely to be erroneous or correct. We use the post-digital normalization abundance distribution as a guide. Though it is possible to use a bimodal distribution analytically obtained by making the assumptions made from previous sections, in practice there are many issues with this approach. First, as the assumptions about repetitive and low complexity regions are unreasonable, there is a possibility that the model may underestimate the number of erroneous k-mers. Ideally, an error correction algorithm should limit the number of "false corrections" as much as possible, so a more conservative approach is key. Second, the model requires many parameters that may not be known, so a more simple solution may be desirable. For these reasons, we choose to use a mixed model containing two Poisson distributions with one distribution belonging to error k-mers and the other to correct k-mers. The lambda parameter for the correct k-mer distribution can be directly tied to the C parameter of the digital normalization algorithm, which means that the k-mer abundance distribution of the dataset after applying digital normalization does not have to be obtained before hand. To determine whether a k-mer is classified as either erroneous or correct, we simply pick which distribution the k-mer of interest fits best in.

4.2.3 Error Region Heuristics

A pressing limitation to a graph-based alignment strategy is that in complex regions of the assembly graph, graph search algorithms have a difficult time quickly finding the lowest cost path. Furthermore, aligning an entire read to the graph exacerbates the issue since this leads to more possible paths for the graph aligner to traverse. These issues can make the algorithm unreasonably slow for longer reads. To address this, we introduce a simple heuristic to shorten the size of the read that is being aligned.

Using the method described in the previous section to determine whether or not a k-mer is erroneous or not, we can go along a read and see the "error footprint" of a read. Since a single base pair error results in k erroneous k-mers, we expect that if an erroneous k-mer exists, there are likely to be several more in the same region. The longer this region, the longer the graph alignment and thus the more time it takes to complete an alignment. To avoid aligning an entire read, we adjust the graph alignment algorithm to only align from the last correct k-mer before an erroneous region to the next correct k-mer. Figure 4.5 visually demonstrates how an error footprint may look within a read. The binary numbers represent whether or not the k-mer is considered correct (1 for correct, 0 for erroneous). The red line

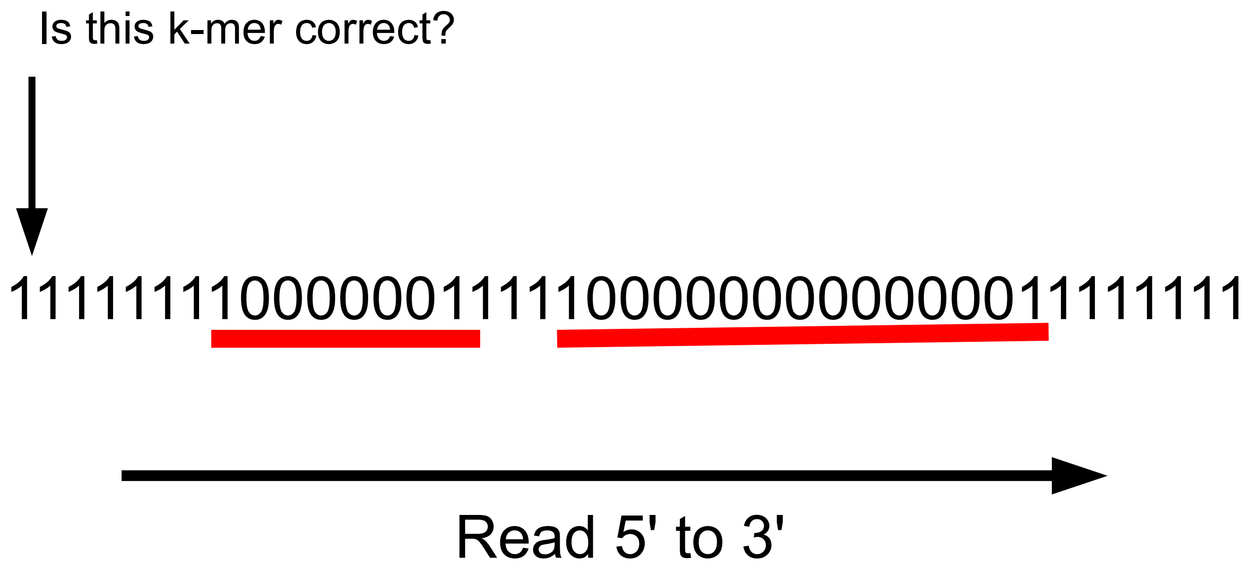


Figure 4.5: Illustration of the error footprint for a read. Each binary number represents whether a k-mer. If the number is 1, the k-mer is deemed correct; if 0, then the k-mer is considered erroneous. Rather than aligning an entire read, the graph search need only span an erroneous region in the graph.

illustrates the region that is aligned with the graph. To do an entire alignment, the stretches of 1's are added to the result along with the alignments of the 0 regions. This approach makes a necessary assumption that any so-called correct k-mers are in fact correct. To align the entire read becomes prohibitive for longer stretches of sequence, and thus this method is necessary. Furthermore, we assume that the first and last k-mer of the starting alignments are correct and exist in the Counting Bloom filter. If this is not the case, then the alignment cannot be made and the entire read is discarded.

Even when adding the heuristic to only align over erroneous regions, there are still likely to be issues when trying to align long stretches. Figure 4.6 shows a histogram of the lengths of the error regions in an *E. coli* reads dataset with the y-axis on a log scale. As the region size is larger, then number of these regions decrease exponentially. However, there are very

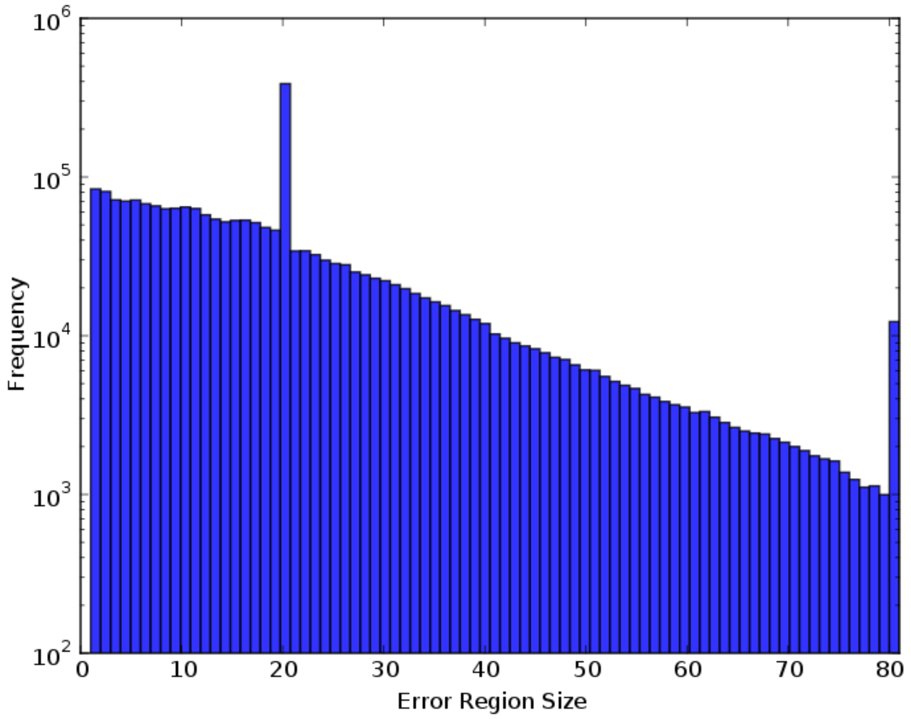


Figure 4.6: Histogram of the lengths of "error regions" in 100,000 reads of an *E. coli* reads dataset. The spike at 20 is due to a single base pair error causing k erroneous k-mers. In this case, $k = 20$. The spike at the end is due to reads with no k-mers called as correct.

noticeable blips at 20 and 80. The spike at 20 is due to SNPs in reads that generate k erroneous k-mers in a row, and in this case, k is set to 20. There is another spike at 80 that occurs when the entire read is erroneous. These reads are discarded since no seed k-mers exist that can be used to anchor the alignment(s) between the read and the graph.

When examining the largest error region size on a per read basis, it is worth noting that most reads have small error regions (Figure 4.7). Since the time to error correct a large error region becomes computationally prohibitive, we add another feature called max-error-region-size that will only error correct a read if the largest error region in the read is less than or equal to the max-error-region-size parameter. If the read's largest error region is higher than this parameter, then the read is discarded. Adding this ability to the read

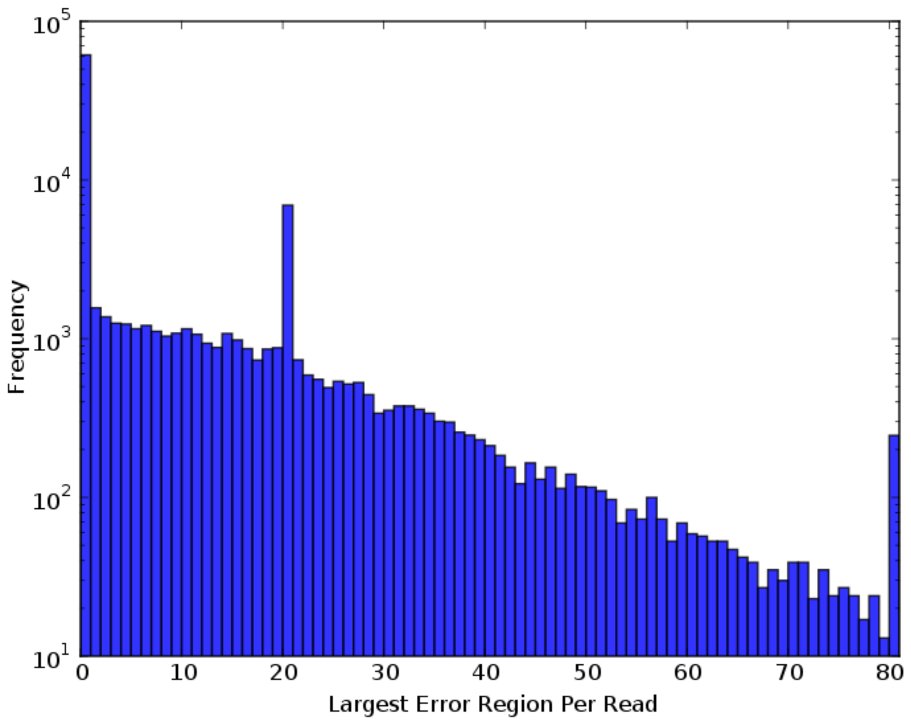


Figure 4.7: Histogram of the largest error region in a 10,000 read subset of the *E. coli* reads dataset.

error correction procedure allows for some flexibility between speed and the number of reads that are corrected. Furthermore, it is likely that aligning longer erroneous segments is less accurate than shorter alignments, so being conservative with this parameter may lead to a higher specificity.

4.3 Error Correction on Genome Sequence Data

Having introduced all of the ingredients to our read error correction algorithm, we will now use it to correct reads from a 5 million reads subset of an *E. coli* sequence dataset (see Data section of this chapter). We first run the digital normalization stage of the algorithm. After applying the digital normalization algorithm at $C = 20$, a peak appears around 20 in the

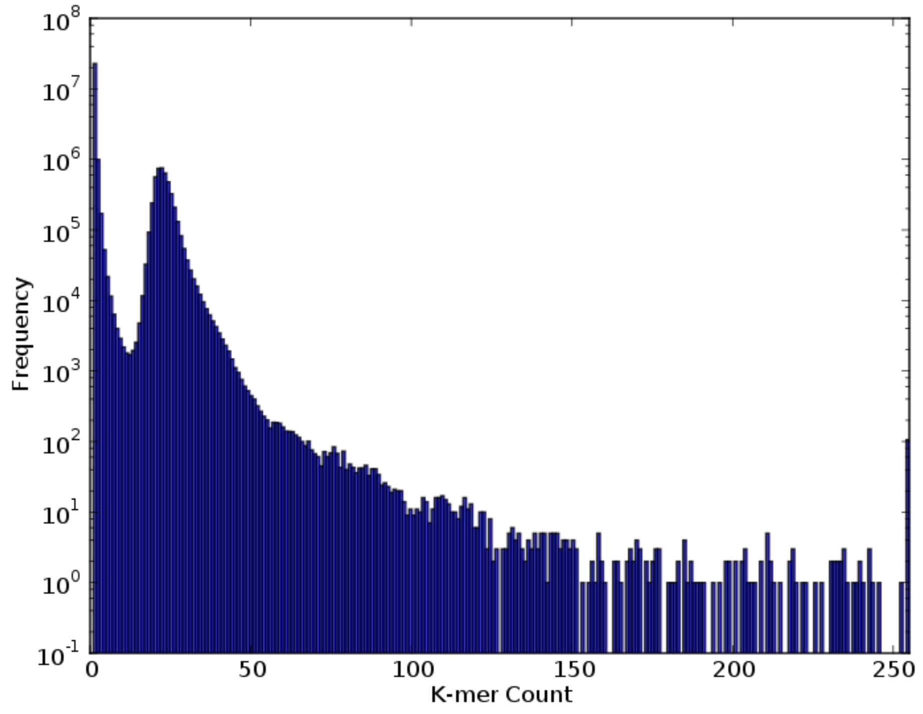


Figure 4.8: The k-mer abundance distribution of the *E. coli* dataset after running the digital normalization algorithm with C=20. Note that the y-axis uses logarithmic scaling.

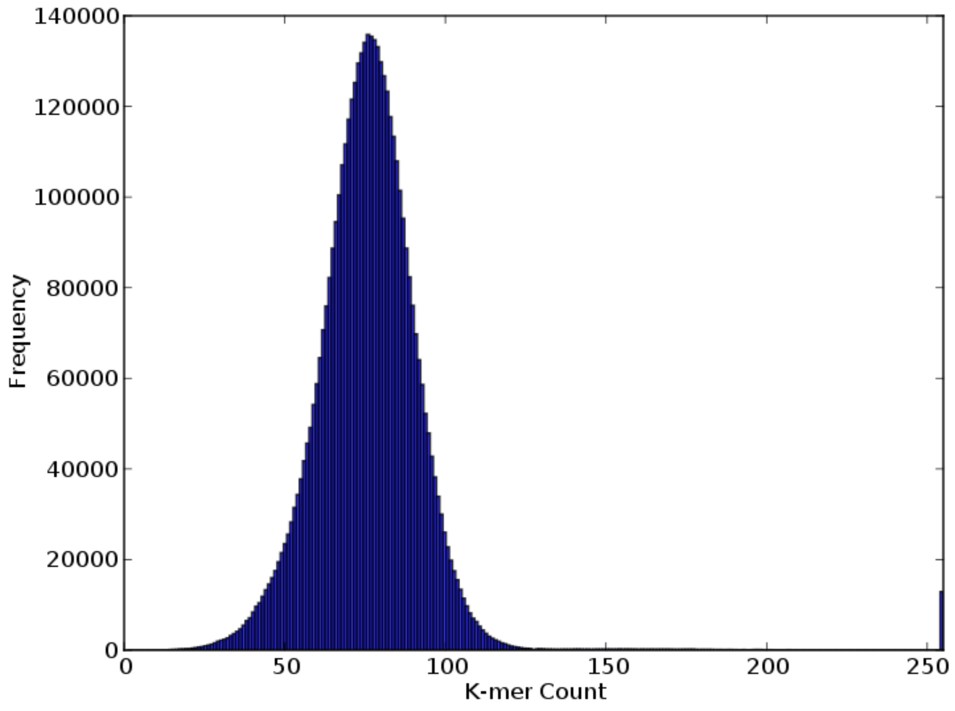
k-mer abundance distribution (Figure 4.8). Compared to the raw distribution (Figure 4.4), the number of erroneous k-mers is greatly reduced. However, the existence of the larger peak allows us to better separate the error k-mers from correct ones, so we can then proceed with the correction stage of the algorithm.

The top graph of Figure 4.9 shows the k-mer abundance distribution after it has been corrected using the genome itself. This is effectively the gold standard for the error correction algorithm since it is not possible for a k-mer that doesn't belong in the genome to be considered correct. Though it is still possible to have false corrections due to the same k-mer being in multiple locations on the genome, this type of error is difficult to mitigate. The graph shows a clear peak around 75, which is where the peak existed albeit slightly in the raw reads abundance distribution. We see that though the distribution does not quite match the “gold standard”, particularly at the left tail, the error correction algorithm performs well when used against the normalized data (shown in the bottom of Figure 4.9).

4.4 Evaluation on Human Gut Mock Dataset

We will now evaluate the error correction procedure on a metagenomic dataset. Because most metagenomic datasets have incomplete references where very few microbes in the sample are fully sequenced, it is difficult, if not impossible, to properly evaluate error correction in this circumstances. However, one mock metagenome dataset exists where microbes whose genomes have been fully sequenced were combined together into a sample and then sequenced. These data were sequenced by the Human Microbiome Project, and they offer two Illumina datasets. The first, an even sample, contains 20 different known genomes with close to equal abundance. The second, a staggered sample, contains the same 20 genomes but with varying levels of abundance. The Data section has more information about where to obtain the datasets.

The goal of the evaluation is to demonstrate that the error correction algorithm considerably reduces the error rate in the reads. Since we have the reference genomes available to us, we can map the reads back to the genomes and look at the number of mismatches in the alignment. However, there are some drawbacks to this method. First, since we will be using Bowtie, insertions and deletions will not be captured. Though this is not ideal, this is not a big issue since most errors from Illumina chemistries are SNPs rather than indels. Another issue is that Bowtie has a limit to the number of mismatches that it check for. This limit is in place presumably because the mapping problem becomes combinatorially more difficult as you increase the number of mismatches you are willing to tolerate. Because we



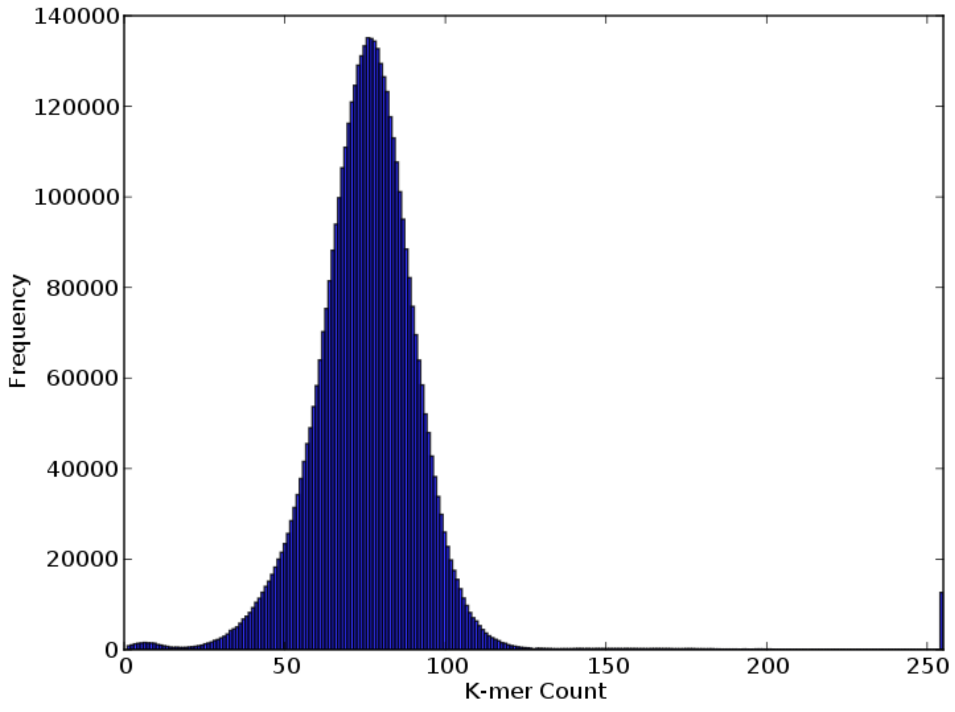


Figure 4.9: K-mer abundance distributions on the *E. coli* dataset after error correction using both the genome (top) and the dataset post-digital normalization(bottom) as the abundance distribution used for error correction.

Table 4.2: Error correction and mapping results on HMP Mock Dataset

	Even Dataset	Staggered Dataset
Total Reads	6433051	7686356
Mappable Reads	4598477 (71.48%)	3741334 (48.68%)
EC Reads	4245210 (65.99%)	5073565 (66.01%)
EC Mappable	4148620 (97.72%)	4740514 (93.44%)

are limited in the number of mismatches to which we can map a read, we are unable to map the most error-prone reads. Thus, this means that any error rate given as a result of the mappings are an underestimate. However, we can combine the percentage of reads that map to the reference genomes with the error rates of the ones that are mappable to get a good idea of the relative number of errors. Though we could change the parameters of Bowtie to get as many reads as possible, we used the default parameters since we are only interested in comparing reads that the error correction algorithm is likely to keep and correct. Since the error correction algorithm detects and discards particularly error-prone reads, we are unlikely to find them in the reads file that is outputted.

To do the evaluation, we will first map the raw reads of the even and staggered HMP datasets to a collection of reference genomes using the default parameters of Bowtie. Then, we will run the digital normalization algorithm against the datasets with $C = 10$ and $K = 23$. Then, we run the second pass of the error correction algorithm with the max-error-region parameter set to 25. We set it to be just above k because we wanted to make sure to include reads that only had error regions caused by a single base pair errors, which is ultimately a large number of reads. By 25, we choose speed over sensitivity, but since we are interested more in obtaining high-quality reads than correcting as many reads as possible, this approach is more favorable. Finally, we map the reads that were successfully error corrected back to the reference genomes.

We could then summarize the results by counting the number of reads that mapped to the reference genomes. With each mapping, we can see the number of mismatches that occurred between the read and the reference, and by counting these we can see the error rate of the mappable reads. Though the even and staggered datasets use the same reference genomes, the percentage of mappable reads differ greatly, at 71.48% and 48.68%, respectively (Table 4.2). After correcting each of the reads datasets, 65.99% of the even reads were kept and corrected along with 66.01% of the staggered. Even though fewer staggered reads were mappable, a similar proportion were able to be corrected. Even more promising, 97.72% of reads in the error-corrected even dataset were successfully mapped back to the reference genomes while 93.44% of the reads in the error corrected staggered dataset mapped back. This strongly implies that the reads are being accurately corrected and that highly error-prone reads are eliminated. Though we do not know how many mismatches existed between the unmappable reads, the specificity is clearly much better compared to the raw dataset.

Tables 4.3 and 4.4 summarize the mapping results when broken down by genome. The first surprising result is that the even dataset is not very evenly distributed by abundance level. Approximately 35.69 % of the raw reads mapped to the *D. radiodurans* genome. In fact, only four genomes had more than 5% of the raw reads map to it. Because of the unevenness of this sample, it is likely that the digital normalization algorithm filtered out most of the reads of many genomes because there was not enough coverage for even $C = 10$. Low abundance organisms like *L. gasseri* and *S. agalactiae* were almost completely filtered out. Though concerning, this does suggest that the error correction procedure works as intended because it is likely that the vast majority of the k-mers in those low abundance genomes would be called as erroneous due to the low coverage. Due to the lower abundance genomes being filtered out, the high abundance organisms were corrected to have many more mappable reads to the genome than the raw dataset. The number of errors between the raw and the error corrected dataset is quite dramatic, with some genomes showing several factors of fewer error bases. The most abundant species, *D. radiodurans* had a more than 10x decrease in the number of errors. However, the difference is actually likely to be much more extreme as the result of Table 4.2 suggest that the percentage of reads that map to the reference genomes in the error corrected dataset is much higher than the raw reads. Since the most error-prone reads are excluded from this analysis, the gap in errors between the raw and error corrected datasets is much larger.

Table 4.3: Even HMP Mock Dataset: raw versus error corrected.

Genome	# Reads (Raw)	# Reads (EC)	Errors (Raw)	Errors (EC)	% Reads (Raw)	% Reads (EC)	% Errors (Raw)	% Errors (EC)
Acinetobacter baumannii	603321	669106	169100	4978	0.1312	0.1613	0.00049031	0.00001644
Actinomyces odontolyticus	105273	49946	50145	254	0.0229	0.0120	0.00014540	0.00000084
Bacillus cereus	40404	922	10642	39	0.0088	0.0002	0.00003086	0.00000013
Bacteroides vulgatus	464746	409685	138186	1629	0.1011	0.0988	0.00040067	0.00000538
Clostridium beijerinckii	195538	49529	47656	233	0.0425	0.0119	0.00013818	0.00000077
Deinococcus radiodurans	1641112	2076680	802391	68753	0.3569	0.5006	0.00232654	0.00022701
Enterococcus faecalis	64467	9536	17883	61	0.0140	0.0023	0.00005185	0.00000020
Escherichia coli K12	74713	6965	26375	65	0.0162	0.0017	0.00007647	0.00000021
Helicobacter pylori	137925	110753	37924	379	0.0300	0.0267	0.00010996	0.00000125
Lactobacillus gasseri	1419	63	398	13	0.0003	0.0000	0.00000115	0.00000004
Listeria monocytogenes	95721	24576	26173	84	0.0208	0.0059	0.00007589	0.00000028
Methanobrevibacter smithii	35121	3846	8912	15	0.0076	0.0009	0.00002584	0.00000005
Neisseria meningitidis	116171	68039	44164	1501	0.0253	0.0164	0.00012805	0.00000496
Propionibacterium acnes	186478	144171	80143	828	0.0406	0.0348	0.00023238	0.00000273
Pseudomonas aeruginosa	31009	492	15761	53	0.0067	0.0001	0.00004570	0.00000017
Rhodobacter sphaeroides	135655	44006	71492	330	0.0295	0.0106	0.00020729	0.00000109
Staphylococcus aureus	139129	78098	69307	22721	0.0303	0.0188	0.00020096	0.00007502
Staphylococcus epidermidis	258945	225277	64960	1518	0.0563	0.0543	0.00018835	0.00000501
Streptococcus agalactiae	4521	278	1175	21	0.0010	0.0001	0.00000341	0.00000007
Streptococcus mutans	88149	34929	23603	122	0.0192	0.0084	0.00006844	0.00000040
Streptococcus pneumoniae	178648	141723	49017	1021	0.0388	0.0342	0.00014213	0.00000337

Table 4.4: Staggered HMP Mock Dataset: raw versus error corrected.

Genome	# Reads (Raw)	# Reads (EC)	Errors (Raw)	Errors (EC)	% Reads (Raw)	% Reads (EC)	% Errors (Raw)	% Errors (EC)
Acinetobacter baumannii	19219	419	7219	11	0.0051	0.0001	0.00002573	0.00000003
Actinomyces odontolyticus	305	3	165	0	0.0001	0.0000	0.00000059	0.00000000
Bacillus cereus	13209	465	4771	58	0.0035	0.0001	0.00001700	0.00000017
Bacteroides vulgatus	1014	9	426	5	0.0003	0.0000	0.00000152	0.00000001
Clostridium beijerinckii	59670	3250	20078	68	0.0159	0.0007	0.00007155	0.00000019
Deinococcus radiodurans	8562	160	5722	6	0.0023	0.0000	0.00002039	0.00000002
Enterococcus faecalis	336	179	171	39	0.0001	0.0000	0.00000061	0.00000011
Escherichia coli K12	309905	306980	149326	2771	0.0828	0.0648	0.00053217	0.00000793
Helicobacter pylori	4186	8	1541	3	0.0011	0.0000	0.00000549	0.00000001
Lactobacillus gasseri	358	97	140	10	0.0001	0.0000	0.00000050	0.00000003
Listeria monocytogenes	3313	271	1320	46	0.0009	0.0001	0.00000470	0.00000013
Methanobrevibacter smithii	183680	204721	63333	627	0.0491	0.0432	0.00022571	0.00000179
Neisseria meningitidis	7495	3759	4208	1344	0.0020	0.0008	0.00001500	0.00000385
Propionibacterium acnes	8071	23	4846	4	0.0022	0.0000	0.00001727	0.00000001
Pseudomonas aeruginosa	75960	10827	52546	462	0.0203	0.0023	0.00018726	0.00000132
Rhodobacter sphaeroides	754540	1249509	542721	41796	0.2017	0.2636	0.00193414	0.00011964
Staphylococcus aureus	896931	1191682	511311	357082	0.2397	0.2514	0.00182221	0.00102213
Staphylococcus epidermidis	1019837	1306522	352521	22359	0.2726	0.2756	0.00125631	0.00006400
Streptococcus agalactiae	21450	1383	7779	53	0.0057	0.0003	0.00002772	0.00000015
Streptococcus mutans	352806	460061	129918	2779	0.0943	0.0970	0.00046300	0.00000795
Streptococcus pneumoniae	475	186	207	32	0.0001	0.0000	0.00000074	0.00000009

Unsurprisingly, the difference in mappable raw reads between the most highly abundant is the greatest in the staggered dataset, but there was not a single most dominant species. Three genomes, *R. sphaeroides*, *S. aureus*, and *S. epidermidis*, had between 20-30% of the mappable raw reads while the rest had below 10%. Compared to the even dataset, more genomes were low abundance and were discarded by the error correction algorithm. In general, the trend appears to be that when the error correction algorithm is run, more reads from high abundance organisms become mappable while fewer reads from lower abundance organisms are. Despite this, as long as the abundance of the species is high enough, the number of reads that are kept and mappable either increases or remains steady compared to the raw dataset. Furthermore, the number of errors between the raw and error corrected datasets is quite large even though the number of mappable reads in both datasets are quite similar. Once again, since the percentage of mappable reads in the raw dataset is far lower than the error corrected dataset, the number of errors is far greater in the raw dataset than what the tables present.

4.5 Data

4.5.1 E. coli Dataset

The *E. coli* dataset originates from [9] but is a subset containing only five million reads. This dataset can be obtained from `https://s3.amazonaws.com/public.ged.msu.edu/ecoli_ref-5m.fastq.gz` and was used in the digital normalization paper[6].

4.5.2 Human Gut Mock Dataset

The human gut mock dataset was obtained from the NCBI Short Read Archive with accession numbers SRX055380 (6.6 million reads) and SRX055381 (7.9 million reads) where the former corresponds to the even sample and the latter corresponds to the staggered. The genomes of the microbes used in the mock dataset were obtained from `ftp://ftp.hgsc.bcm.tmc.edu/pub/misc/HMP/mock/mock.all.genome.fa`.

Chapter 5

Conclusions

We have established a framework for scaling metagenome sequence assembly. This dissertation covers an overview of existing approaches and how they scale. Building on this foundation, we described a probabilistic de Bruijn graph and described its properties. Furthermore, we evaluated the utility of a partitioning algorithm that can be used as a precursor to assembly to bring down memory requirements. In the next approach, we established a metagenome error correction algorithm. In this chapter, we briefly summarize how these approaches help to tackle the metagenome assembly problem and discuss possible avenues for future work.

5.1 Summary

The overarching goal of the projects discussed in this dissertation is to better scale metagenome sequence assembly. Scaling can be applied in a number of different ways, but we focus on two. In the first, we improve the memory requirements for assembly by constant factor. In the second, we reduce the number of errors on the assembly graph, which speeds up the assembly process, improves accuracy, and further lowers the memory footprint. These approaches combined work to effectively scale metagenome assembly in terms of both memory and running time.

By using a simple probabilistic data structure, a Bloom filter, we are able to store k-mers in a much more memory efficient manner compared to exact methods. However, this added efficiency comes at the cost of false positives where k-mers that are not present in the dataset are declared present when queried. When adding the false positive k-mers to a de Bruijn graph constructed from a set of reads, we call the resulting graph a probabilistic de Bruijn graph. We found that below a false positive rate of approximately 0.183, graph components do not erroneously fuse together, which enables for a component-separation step prior to assembly. Assuming extremely large components do not exist in the original graph, this can lead to memory savings in later stages of an assembly pipeline. We successfully tested this algorithm on a real soil metagenomic dataset, and compared the false positive k-mers generated by the Bloom filter with error k-mers caused by sequencing technology.

By increasing the number of bits in a Bloom filter, we obtain a Counting Bloom Filter that can be used to count the number of k-mers in a sequencing dataset. In addition to the false positives where a k-mer that is not in the dataset is declared present, there is an overestimation error. If each has at least one collision, there will be an overestimate. Regardless, the generalization of the Bloom filter to larger bin sizes enables fairly accurate k-mer counting, especially for metagenome datasets that have many low abundance k-mers.

Using a Counting Bloom filter, we can obtain the k-mer abundance distribution for a DNA sequence dataset. For a simple genome, the distribution will be bimodal, with many low frequency k-mers due to errors and then a second peak with the mode at the expected k-mer coverage. This distribution does not generally occur from metagenomic datasets because of the variability in abundance levels among different microbial "species" in the sample. However, by applying the digital normalization algorithm, one can normalize the abundance level to even out the frequencies within the sample. With a new k-mer abundance distribution, we can appropriately discriminate between erroneous and correct k-mers by creating a mixture model. Then, we can align each read in a dataset to the post-digital normalization graph to error correct the reads. By comparing the k-mer abundance distribution between the *E. coli* dataset corrected with the genome and with digital normalization, we showed that the error correction method produces a much improved k-mer spectra.

We used a human gut mock dataset to show that the error correction procedure fixes the vast majority of reads while discarding those that would complicate assembly. By demonstrating the accuracy of the method on such a metagenomic dataset, we are able to rely on the reference genomes of the microbes that were sequenced. This allowed us to accurately map reads back to the reference genomes and determine that the error correction algorithm successfully discarded a large number of erroneous reads and greatly improve the accuracy of the existing ones.

Altogether, these memory reduction and error correction methods can be combined together as part of a pre-assembly pipeline to improve the accuracy of a metagenomics dataset. Highly diverse samples require deep sampling to discover the novelty in sufficient detail, so a buildup of error k-mers is to be expected, especially for highly abundance species. The memory reduction filtering approaches effectively remove erroneous k-mers without thoroughly examining them (though sufficient coverage of these regions are assured through the digital normalization algorithm) while the error correction method examines the k-mer abundance distribution to correct erroneous k-mers to trusted one. The elimination and correction of erroneous k-mers is ultimately what makes these approaches successful.

5.2 Future Work

Though the approaches detailed in this dissertation reduce the memory requirements for metagenome sequence assembly as well as improve accuracy, there is still plenty of room for improvement. Due to the speed at which new sequencing technologies are being developed and improved upon, it is likely that all of the datasets used in these works are from chemistries several generations old. It is important to develop algorithms that generalize well among different sequencing platforms. While it is difficult to predict how sequencing will continue to improve in the next 2-5 years, it is reasonable to expect that read lengths will continue to be longer and error rates will continue to decrease. With these points in mind, there are many areas worth exploring to continue to improve metagenome sequence assembly.

One area of improvement is to develop more sensitive metagenome sequence error correction algorithms. Though the approach elaborated in this work successfully improves the quality of metagenome assemblies, it is not sensitive to rare variant detection. It is possible to map the reads back to the metagenome assembly to detect these variants, but ideally they would be detected and kept rather than potentially discarded as redundant sequence due to the effects of the digital normalization algorithm.

The error correction algorithm implemented in this chapter could be greatly improved by a new seeding strategy. Currently, the method requires that at least one k-mer in the read is considered correct. However, for larger values of k, multiple sequencing errors can result in large error regions that are difficult to align via the graph. Currently, the maximum error region size heuristic drastically improves the performance at the expense of discarding exceptionally error-prone reads. This heuristic is undesirable for longer reads and any sequencing technologies with a higher error rate. To address this, any new seeding strategy would not require exact matching. One possibility is to employ the Hamming distance graph in a similar manner to the Hammer read error corrector to find seeds along the read that can then be bridged using the graph alignment strategy. On the other hand, with deep enough sequencing, it may make sense to to willingly discard exceptionally error-prone reads.

Another way the two-pass error correction method could be improved is to use a one-pass, online formulation. In order to do this, one would need to apply the digital normalization algorithm while simultaneously error correcting reads. There would likely need to be a "burn-in" period where many errors are discarded, but eventually there would be enough information to correct reads based on the post-digital normalization k-mer abundance distribution. Another issue that would arise with an online formulation is the use of Bloom filters. The basic type of Bloom filter used in the partitioning and error correction approaches do not remove k-mers over time. To be able to handle any size dataset, a Bloom filter variant (or a streaming data structure) would need to be used that can remove k-mers over time. By putting this framework together, it would be possible to error correct a dataset of arbitrary size. However, even with this method it would be difficult to extract low abundance species from a dataset if the data structure is not able to hold enough novelty. This implies that the bottleneck would continue to be memory and that a memory-efficient approach is required.

Along similar lines, online metagenome sequence assembly is a worthy goal for Bioinformaticians to pursue. Deep sampling of environmental samples is required to fully capture the diversity within a microbial ecosystem, so longer reads and fewer errors will not help to sequence the least abundance microbes. Thus, it will be necessary to develop algorithms that treat a reads dataset as a stream where only one pass through is possible. This would require incorporating memory efficiency and error correction into the assembly process. As the more abundance sequences are assembled, these parts of the assembly graph must be removed over time and screened from being added again. This will help to keep memory requirements low while allowing low abundance organisms to be sampled, which can be functionally important.

APPENDIX

This appendix contains a glossary of commonly used terms within this dissertation.

Assembler: A computer program that takes multiple DNA sequence reads, finds overlaps between them, and outputs an assembly.

Assembly: A collection of contigs generated by an assembler.

Contig: Short for contiguous sequence, a contig is a collection of zero or more overlaps from one or more reads.

Coverage: The coverage at a particular location in a genome is the number of reads that map to that location. This definition can be extended to assembly graphs where the k-mer coverage is considered the number of reads that contain a given k-mer.

de Bruijn Graph: A type of graph that is used for DNA sequence assembly. A de Bruijn graph can have an alphabet size of 2 or larger and a word size of 1 or more. In the case of DNA, the alphabet size is 2 and the word size varies with the dataset. A vertex is a DNA word of fixed size k, and its neighbors are any k-mers that share a $k-1$-mer overlap.

Genome: The collection of DNA sequences that make up a cell, organism, or species. An eukaryotic species like human does not possess a single genome, but rather one in each cell. Each individual within a species also has unique genomes even though the genome for a species may be collectively referred to as the genome for that species (e.g. the Human Genome).

Indel: When comparing a two aligned sequences, an indel is an insertion of one or more base pairs and a corresponding deletion in the other sequence.

K-mer: A k-mer is a DNA word of a fixed size k. If $k = 4$, an example of a 4-mer is CGAT.

Mapper: A mapper is a computer program that aligns a DNA sequence read to a reference genome or sequence.

Metagenome: A collection of genomes obtained from sequencing an environmental sample.

OTU: Short for Operational Taxonomic Unit. It is used to bin similar organisms together based on similarity as a way to form an arbitrary definition of a species in a microbial context.

Overlap: Where a prefix of one read is the same as the suffix of another. Usually, a putative overlap must be of a certain size to be considered a true overlap. Inexact overlaps, where mismatches exist between reads, are also possible.

Paired-End Read: When two sides of a longer DNA sequence fragment is sequenced. The middle part of the sequence is not known, but the length between the two sequenced reads are known. This added information assists in generating and validating sequence assemblies.

Read: A single sequence obtained from a sequencer.

Reverse Complement: Given a DNA sequence, the reverse complement is the opposite strand of the DNA sequence read in opposite order since a DNA strand is generally read in only one direction. The reverse complement is a combination of a string reverse operation and a DNA complement operation (substitute A for T, T for A, C for G, and C for G).

SNP: A single nucleotide polymorphism. This is a single substitution between two or more aligned sequences.

Scaffold: The gap between two contigs. That is, the sequences between the contigs are not known, but the direction and length between the contigs is.

Supercontig: A collection of contigs where their relative positions to on another are known due to scaffolds.

Shotgun Sequencing: A method for sequencing where DNA sequencing reads are essentially randomly sampled from a genome (or multiple genomes).

Transcriptome: The collection of mRNA transcripts from a cell or organism.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] C. Adami and J. Chu. Critical and near-critical branching processes. *Physical Review E*, 66(1):011907, 2002.

[2] S. Batzoglou, D. Jaffe, K. Stanley, J. Butler, S. Gnerre, E. Mauceli, B. Berger, J. Mesirov, and E. Lander. Arachne: a whole-genome shotgun assembler. *Genome Research*, 12(1):177, 2002.

[3] B. Bloom. Space/time tradeoffs in hash coding with allowable errors. *CACM*, 13(7):422–426, 1970.

[4] S. Boisvert, F. Laviolette, and J. Corbeil. Ray: Simultaneous assembly of reads from a mix of high-throughput sequencing technologies. *Journal of Computational Biology*, 17(11):1519–1533, 2010.

[5] S. Boisvert, F. Raymond, E. Godzaridis, F. Laviolette, and J. Corbeil. Ray meta: scalable de novo metagenome assembly and profiling. *Genome Biology*, 13(12):R122, 2012.

[6] C. Brown, A. Howe, Q. Zhang, A. Pyrkosz, and T. Brom. A reference-free algorithm for computational normalization of shotgun sequencing data. in review at PLoS One, july 2012. *Preprint at http://arxiv. org/abs/1203.4802*, 2012.

[7] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. 1994.

[8] T. Chen and S. Skiena. Trie-based data structures for sequence assembly. In *Combinatorial Pattern Matching*, pages 206–223. Springer, 1997.

[9] H. Chitsaz, J. L. Yee-Greenbaum, G. Tesler, M.-J. Lombardo, C. L. Dupont, J. H. Badger, M. Novotny, D. B. Rusch, L. J. Fraser, N. A. Gormley, et al. Efficient de novo assembly of single-cell bacterial genomes from short-read data sets. *Nature biotechnology*, 2011.

[10] T. C. Conway and A. J. Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479–486, Feb 2011.

[11] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[12] N. de Bruijn and P. Erdos. A combinatorial problem. *Koninklijke Netherlands: Academe Van Wetenschappen*, 49:758–764, 1946.

[13] D. Earl, K. Bradnam, J. St John, A. Darling, D. Lin, J. Fass, H. O. Yu, V. Buffalo, D. R. Zerbino, M. Diekhans, N. Nguyen, P. N. Ariyaratne, W. K. Sung, Z. Ning, M. Haimel, J. T. Simpson, N. A. Fonseca, . Birol, T. R. Docking, I. Y. Ho, D. S. Rokhsar, R. Chikhi, D. Lavenier, G. Chapuis, D. Naquin, N. Maillet, M. C. Schatz, D. R. Kelley, A. M. Phillippy, S. Koren, S. P. Yang, W. Wu, W. C. Chou, A. Srivastava, T. I. Shaw, J. G. Ruby, P. Skewes-Cox, M. Betegon, M. T. Dimon, V. Solovyev, I. Seledtsov, P. Kosarev, D. Vorobyev, R. Ramirez-Gonzalez, R. Leggett, D. MacLean, F. Xia, R. Luo, Z. Li, Y. Xie, B. Liu, S. Gnerre, I. MacCallum, D. Przybylski, F. J. Ribeiro, S. Yin, T. Sharpe, G. Hall, P. J. Kersey, R. Durbin, S. D. Jackman, J. A. Chapman, X. Huang, J. L. DeRisi, M. Caccamo, Y. Li, D. B. Jaffe, R. E. Green, D. Haussler, I. Korf, and B. Paten. Assemblathon 1: a competitive assessment of de novo short read assembly methods. *Genome Res.*, 21(12):2224–2241, Dec 2011.

[14] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000.

[15] X. Feng, Y. Deng, and H. Blöte. Percolation transitions in two dimensions. *Physical Review E*, 78(3):031136, 2008.

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

[17] J. Gilbert, F. Meyer, J. Jansson, J. Gordon, N. Pace, J. Tiedje, R. Ley, N. Fierer, D. Field, N. Kyrpides, F. Glockner, H. Klenk, K. Wommack, E. Glass, K. Docherty, R. Gallery, R. Stevens, and R. Knight. The earth microbiome project: Meeting report of the '1 emp meeting on sample selection and acquisition' at argonne national laboratory october 6 2010. *Stand Genomic Sci*, 3(3):249–53, 2010.

[18] D. Hernandez, P. François, L. Farinelli, M. Østerås, and J. Schrenzel. De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer. *Genome Research*, 18(5):802–809, 2008.

[19] M. Hess, A. Sczyrba, R. Egan, T. W. Kim, H. Chokhawala, G. Schroth, S. Luo, D. S. Clark, F. Chen, T. Zhang, R. I. Mackie, L. A. Pennacchio, S. G. Tringe, A. Visel,

T. Woyke, Z. Wang, and E. M. Rubin. Metagenomic discovery of biomass-degrading genes and genomes from cow rumen. *Science*, 331:463–467, Jan 2011.

[20] X. Huang and S. Yang. Generating a genome assembly with pcap. *Current Protocols in Bioinformatics*, 2005.

[21] Z. Iqbal, M. Caccamo, I. Turner, P. Flicek, and G. McVean. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nat. Genet.*, 44(2):226–232, Feb 2012.

[22] V. Iverson, R. M. Morris, C. D. Frazar, C. T. Berthiaume, R. L. Morales, and E. V. Armbrust. Untangling genomes from metagenomes: revealing an uncultured class of marine Euryarchaeota. *Science*, 335(6068):587–590, Feb 2012.

[23] D. R. Kelley, M. C. Schatz, and S. L. Salzberg. Quake: quality-aware detection and correction of sequencing errors. *Genome Biol.*, 11(11):R116, 2010.

[24] H. Kesten. The critical probability of bond percolation on the square lattice equals 1/2. *Communications in Mathematical Physics*, 74(1):41–59, 1980.

[25] E. S. Lander and M. S. Waterman. Genomic mapping by fingerprinting random clones: a mathematical analysis. *Genomics*, 2(3):231–239, 1988.

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

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

[28] R. Li, H. Zhu, J. Ruan, W. Qian, X. Fang, Z. Shi, Y. Li, S. Li, G. Shan, K. Kristiansen, S. Li, H. Yang, J. Wang, and J. Wang. De novo assembly of human genomes with massively parallel short read sequencing. *Genome Res.*, 20(2):265–272, Feb 2010.

[29] Y. Liu, J. Schröder, and B. Schmidt. Musket: a multistage k-mer spectrum-based error corrector for illumina sequence data. *Bioinformatics*, 29(3):308–315, 2013.

[30] R. Mackelprang, M. P. Waldrop, K. M. DeAngelis, M. M. David, K. L. Chavarria, S. J. Blazewicz, E. M. Rubin, and J. K. Jansson. Metagenomic analysis of a permafrost microbial community reveals a rapid response to thaw. *Nature*, 480(7377):368–371, Dec 2011.

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

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

[33] P. Medvedev, E. Scott, B. Kakaradov, and P. Pevzner. Error correction of high-throughput sequencing datasets with non-uniform coverage. *Bioinformatics*, 27(13):i137–141, Jul 2011.

[34] F. Meyer, D. Paarmann, M. D'Souza, R. Olson, E. M. Glass, M. Kubal, T. Paczian, A. Rodriguez, R. Stevens, A. Wilke, J. Wilkening, and R. A. Edwards. The metagenomics RAST server - a public resource for the automatic phylogenetic and functional analysis of metagenomes. *BMC Bioinformatics*, 9:386, 2008.

[35] J. Miller, S. Koren, and G. Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327, 2010.

[36] E. Myers, G. Sutton, A. Delcher, I. Dew, D. Fasulo, M. Flanigan, S. Kravitz, C. Mobarry, K. Reinert, K. Remington, et al. A whole-genome assembly of Drosophila. *Science*, 287(5461):2196, 2000.

[37] E. W. Myers. The fragment assembly string graph. *Bioinformatics*, 21 Suppl 2:79–85, Sep 2005.

[38] T. Namiki, T. Hachiya, H. Tanaka, and Y. Sakakibara. MetaVelvet: An extension of Velvet assembler to de novo metagenome assembly from short sequence reads. *ACM Conference on Bioinformatics, Computational Biology and Biomedicine*, 2011.

[39] Y. Peng, H. Leung, S. Yiu, and F. Chin. Meta-IDBA: a de novo assembler for metagenomic data. *Bioinformatics*, 27(13):i94–i101, 2011.

[40] P. A. Pevzner, H. Tang, and M. S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proc. Natl. Acad. Sci. U.S.A.*, 98:9748–9753, Aug 2001.

[41] J. Qin, R. Li, J. Raes, M. Arumugam, K. S. Burgdorf, C. Manichanh, T. Nielsen, N. Pons, F. Levenez, T. Yamada, D. R. Mende, J. Li, J. Xu, S. Li, D. Li, J. Cao, B. Wang, H. Liang, H. Zheng, Y. Xie, J. Tap, P. Lepage, M. Bertalan, J. M. Batto, T. Hansen, D. Le Paslier, A. Linneberg, H. B. Nielsen, E. Pelletier, P. Renault, T. Sicheritz-Ponten, K. Turner, H. Zhu, C. Yu, S. Li, M. Jian, Y. Zhou, Y. Li, X. Zhang,

S. Li, N. Qin, H. Yang, J. Wang, S. Brunak, J. Dore, F. Guarner, K. Kristiansen, O. Pedersen, J. Parkhill, J. Weissenbach, P. Bork, S. D. Ehrlich, J. Wang, M. Antolin, F. Artiguenave, H. Blottiere, N. Borruel, T. Bruls, F. Casellas, C. Chervaux, A. Cultrone, C. Delorme, G. Denariaz, R. Dervyn, M. Forte, C. Friss, M. van de Guchte, E. Guedon, F. Haimet, A. Jamet, C. Juste, G. Kaci, M. Kleerebezem, J. Knol, M. Kristensen, S. Layec, K. Le Roux, M. Leclerc, E. Maguin, R. M. Minardi, R. Oozeer, M. Rescigno, N. Sanchez, S. Tims, T. Torrejon, E. Varela, W. de Vos, Y. Winogradsky, and E. Zoetendal. A human gut microbial gene catalogue established by metagenomic sequencing. *Nature*, 464:59–65, Mar 2010.

[42] M. Rho, H. Tang, and Y. Ye. FragGeneScan: predicting genes in short and error-prone reads. *Nucleic Acids Res.*, 38(20):e191, Nov 2010.

[43] R. Ronen, C. Boucher, H. Chitsaz, and P. Pevzner. SEQuel: improving the accuracy of genome assemblies. *Bioinformatics*, 28(12):i188–196, Jun 2012.

[44] S. L. Salzberg, A. M. Phillippy, A. Zimin, D. Puiu, T. Magoc, S. Koren, T. J. Treangen, M. C. Schatz, A. L. Delcher, M. Roberts, G. Marcais, M. Pop, and J. A. Yorke. GAGE: A critical evaluation of genome assemblies and assembly algorithms. *Genome Res.*, 22(3):557–567, Mar 2012.

[45] J. Simpson and R. Durbin. Efficient construction of an assembly string graph using the fm-index. *Bioinformatics*, 26(12):i367–i373, 2010.

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

[47] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. Jones, and I. Birol. ABySS: a parallel assembler for short read sequence data. *Genome Res.*, 19:1117–1123, Jun 2009.

[48] D. Stauffer. Scaling theory of percolation clusters. *Physics Reports*, 54(1):1–74, 1979.

[49] K. Stefan, N. Apurva, S. Joshua, and W. Doreen. A new method to compute k-mer frequencies and its application to annotate large repetitive plant genomes. *BMC Genomics*, 9, 2008.

[50] G. W. Tyson, J. Chapman, P. Hugenholtz, E. E. Allen, R. J. Ram, P. M. Richardson, V. V. Solovyev, E. M. Rubin, D. S. Rokhsar, and J. F. Banfield. Community structure and metabolism through reconstruction of microbial genomes from the environment. *Nature*, 428(6978):37–43, Mar 2004.

[51] J. C. Venter, K. Remington, J. F. Heidelberg, A. L. Halpern, D. Rusch, J. A. Eisen, D. Wu, I. Paulsen, K. E. Nelson, W. Nelson, D. E. Fouts, S. Levy, A. H. Knap, M. W. Lomas, K. Nealson, O. White, J. Peterson, J. Hoffman, R. Parsons, H. Baden-Tillson, C. Pfannkoch, Y. H. Rogers, and H. O. Smith. Environmental genome shotgun sequencing of the Sargasso Sea. *Science*, 304(5667):66–74, Apr 2004.

[52] P. Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11. IEEE, 1973.

[53] D. Wu, M. Wu, A. Halpern, D. Rusch, S. Yooseph, M. Frazier, J. Venter, and J. Eisen. Stalking the fourth domain in metagenomic data: searching for, discovering, and interpreting novel, deep branches in marker gene phylogenetic trees. *PLoS One*, 6(3):e18011, 2011.

[54] D. R. Zerbino and E. Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res.*, 18:821–829, May 2008.