December 2017

Error Correction and de novo Genome Assembly of DNA Sequencing Data

Michael Z. Molnar
The University of Western Ontario

Supervisor
Dr. Lucian Ilie
The University of Western Ontario

Graduate Program in Computer Science

A thesis submitted in partial fulfillment of the requirements for the degree in Doctor of Philosophy

Recommended Citation

Molnar, Michael Z., "Error Correction and de novo Genome Assembly of DNA Sequencing Data" (2017). *Electronic Thesis and Dissertation Repository*. 5050.
https://ir.lib.uwo.ca/etd/5050

Abstract

The ability to obtain the genetic code of any species has caused a revolution in biological sciences. Current technologies are capable of sequencing short pieces of DNA with very high quality. These short pieces of DNA determint the sequence of bases in the genome of any species. This information is key in understanding many of the aspects of how life functions.

The accuracy of sequencing is extremely important since the differences between individuals of the same species are caused by very few changes. All sequencing technologies make errors, and before the data can be used for downstream applications it is usually best to correct the errors first. I present an error correction program called RACER that is an error correction program that aims to correct substitution sequencing errors.

There are many substitution error correction programs available for DNA sequencing technologies, so it is important for biologists to know which program is best to use for their sequencing technology. I present a comprehensive survey of substitution error correction programs for DNA sequencing data to address this issue. I also present two programs to evaluate the performance of error correcting programs.

Since the current dominant platform in the market can only obtain small pieces of DNA, software is needed to assemble these pieces to determine the full sequence of the sampled genome. Current genome assembly programs are not capable of assembling the entire genome of most species due to the repetitive nature of genomes and the uneven coverage of the sampled genome. I present a genome assembly program called SAGE2 that improves upon the current state-of-the-art.

Keywords: de novo genome assembly, substitution error correction, Illumina, next-generation sequencing, bioinformatics, HiSeq, MiSeq, RACER, SAGE2

It has been a long and difficult path to complete my education, and I could not have done it without the support of my family. I dedicate this work to them. Thank you Mom, Dad, and my sisters Debby and Judy. I could not have completed this work without the support of my wife Kimber-Lee and our four children; Kallie, Cooper, Cole, and our new baby Kamryn.

Acknowlegements

My path to a PhD. was long and difficult but it would not have been possible without the guidance of my supervisor Dr. Lucian Ilie. Without his commitment to me I would not be where I am today. I owe my deepest gratitude to him for taking a chance on me as a graduate student and guiding me throughout the years. I am also grateful to Dr. Roberto Solis-Oba for accepting me as a graduate student and providing guidance through my time at the University of Western Ontario.

I am thankful to Dr. Md. Bahlul Haider for his guidance and advise in the beginning of my graduate career, and helping me understand genome assembly. I would like to thank Ehsan Haghshenas for helping me with the development of SAGE2. I would also like to thank Nilesh Khiste for helping me with parts of SAGE2 and for our great talks about school and life.

I would like to thank the staff in the Computer Science department at the University of Western Ontario for all their help through my graduate studies. In particular the support of Janice Wiersma and Cheryl McGrath whom I bothered many times, and they were always there to help.

Finally I would like to thank the University of Western Ontario for providing me with the Western Graduate Research Scholarship, and the Ontario government for awarding me the Queen Elizabeth II Graduate Scholarships in Science and Technology. Without the financial support I would not have been able to complete my studies.

Contents

List of Figures

List of Tables

List of Appendices

Chapter 1

Introduction

Biological sciences have been revolutionized by the information obtained from DNA sequencing technologies over the last two decades. The enormous amounts of data that can now be produced in a short amount of time and at a relatively low cost needs advanced algorithms to make use of the information. With the amount of data that needs to be processed, it is important that these algorithms not only be highly accurate but also time efficient. In this work I present fast and accurate algorithms for processing DNA sequencing data. There is a great deal of data that needs to be searched quickly, and the algorithms I have developed rely heavily on hash tables to find information at near constant time.

My work has addressed three main problems with DNA sequence analysis. The first problem is the correction of errors that are produced by DNA sequencing platforms. Accuracy is extremely important when working with DNA sequencing data, and much of the information produced by sequencing machines contain some errors. It is important for most applications to have the DNA sequencing data corrected before using it in order to achieve a high degree of accuracy.

The second problem is the need to determine the best error correction application for different types of genomes and sequencing platforms. There are many error correction applications available now, and researchers do not have the time or domain knowledge needed to determine the best application to use to correct their data. I present a comprehensive analysis of the best available applications, for the most frequently used sequencing platforms, from a wide range of genomes. I also present two applications I have developed for researchers to asses the performance of their own error correction.

The final problem is that of *de novo* genome assembly, which is the process of assembling the sequence of a sampled genome from sequencing data without a reference genome to assist the assembly process. Genome assembly is one of the most important problems in bioinformatics, and one of the main uses of DNA sequencing data in biological sciences.

The remainder of this chapter will give an overview of DNA sequencing technologies, DNA error correcting software, and *de novo* genome assembly software. The following chapters will outline my work on error correction, evaluation of error correcting software, and *de novo* genome assembly.

1.1 DNA sequencing

One of the most important discoveries in biological sciences was the description of the DNA double-helix by James Watson and Francis Crick in 1953. This discovery led to our understanding of how organisms are able to store the information needed to build and sustain the cells that make up their structure. DNA is composed of four different deoxyribonucleotidess that are linked together by a sugar-phosphate backbone, as shown in Figure 1.1. DNA is double-stranded, and the two strands are held together by hydrogen bonds between the deoxyribonucleotides on each strand. Two complementary deoxyribonucleotides on opposite strands that are hydrogen bonded are referred to as *base pairs* (bp). The deoxyribonucleotide adenine (A) always pairs with its complement deoxyribonucleotide thymine (T), and the deoxyribonucleotide cytosine (C) always pairs with its complement deoxyribonucleotide guanine (G). The genome sequence of a species is the order of all the DNA bases on each of the chromosomes. Our ability to determine the genome sequence of any species has revolutionized our

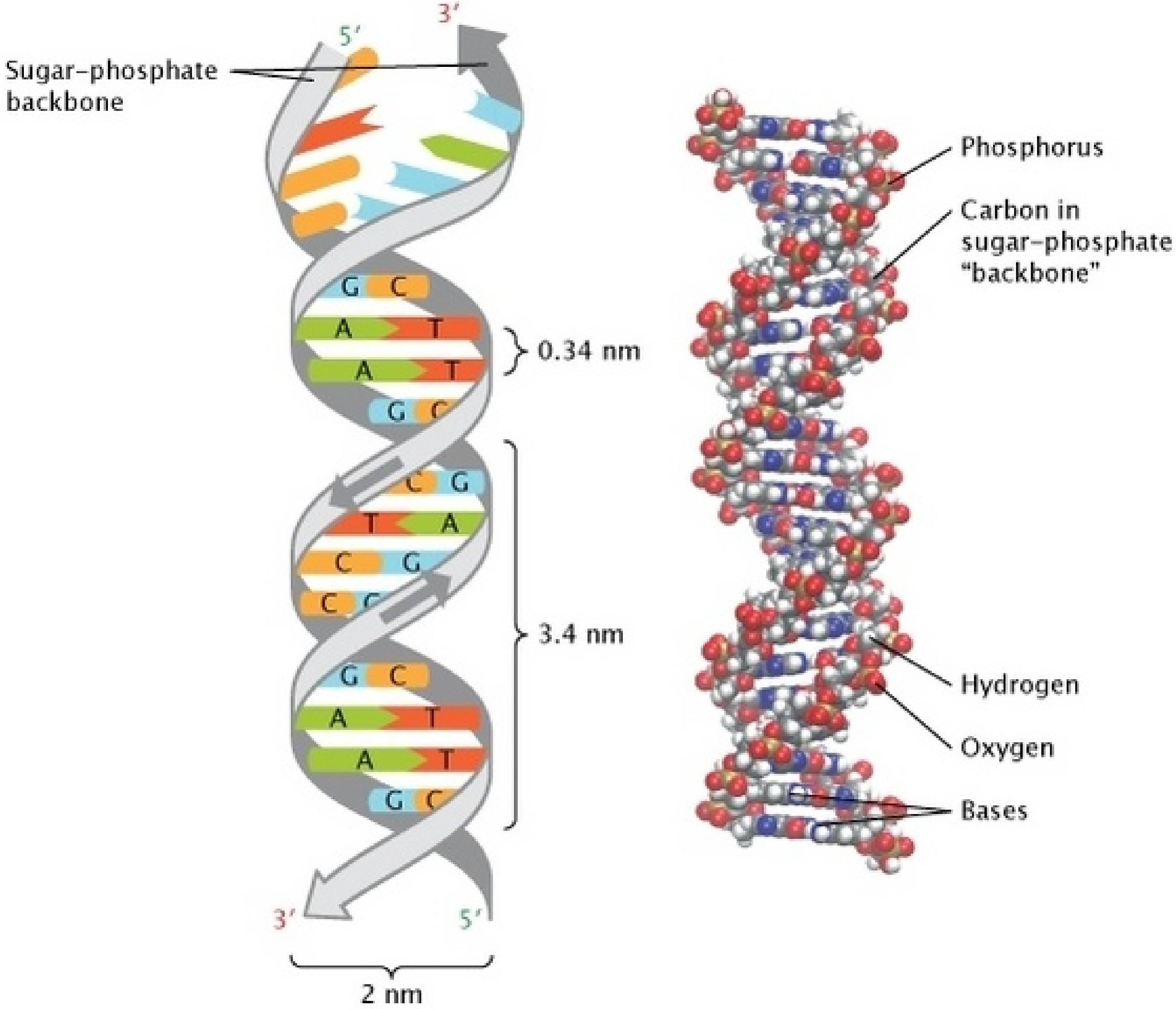


Figure 1.1: An example of the structure of DNA. The sugar-phosphate backbone links deoxyribonucleotides together on each strand which run in opposite directions. The two strands are held together by hydrogen bonds between complementary base pairs on opposite strands [31].

understanding of biology.

1.1.1 Sanger method

The Sanger method [33] of DNA sequencing, developed by Frederick Sanger in the 1970's, was the first widely used method to determine the genome of a species. This method relies on copying DNA fragments with a mixture of regular deoxyribonucleotides, and chain-terminating deoxyribonucleotides initiated from a defined starting point. When the chain-terminating deoxyribonucleotides are incorporated into a DNA fragment it stops the fragment from being copied. This leaves a mixture of DNA fragments with varying lengths that are copies from the original DNA fragment, which can then be separated by their lengths with a technique called *gel-electrophoresis*. This procedure is performed four times for each fragment of DNA, with each experiment using a single chain-terminating deoxyribonucleotide from the four deoxyribonucleotides in DNA. The separation of the fragments creates a ladder of DNA fragments from largest to smallest, and it is then possible to determine the DNA sequence of the original DNA fragment. An example of this ladder is shown in the image on the left of Figure 1.2.

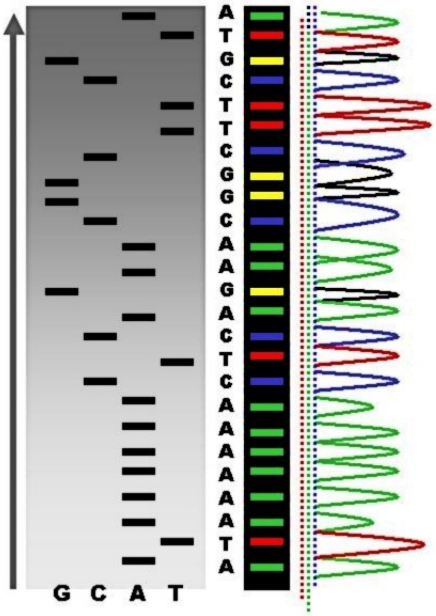


Figure 1.2: The image on the left is an example of the Sanger method of DNA sequencing using the gel-electrophoresis ladder. The order of the deoxyribonucleotides from the bottom to the top represents the sequence of the DNA fragment. The image on the right is the automated Sanger method of DNA sequencing using florescent labels through a capillary tube. Similarly, the order of the deoxyribonucleotides from the bottom to the top represents the sequence of the DNA fragment [36].

The Sanger method was an expensive and time consuming technique, so the initial sequencing projects were limited to small RNA viruses, and bacteria with very small genomes. The technique was eventually improved by automating much of the process [39]. The gel-electrophoresis and radioactive chain-terminated deoxyribonucleotides were replaced by a capillary tube and florescent chain-terminated deoxyribonucleotides. This allows computers to capture the images of the florescent dyes as they passed through the capillary tube to determine the sequence of the DNA fragment. An example of the output from this procedure is shown in the image on the right of Figure 1.2.

The automated Sanger method ushered in a new era of genome sequencing projects. The first free-living organism to have the complete genome sequenced was *Haemophilus influenzae* in 1995 [5], with a genome length of approximately 1.8 million bp. The first unicellular eukaryotic genome sequenced was that of *Saccharomyces cerevisiae* in 1996 [6], with a genome length of approximately 12 million bp. The first multicellular eukaryotic genome sequenced was that of *Caenorhabditis elegans* in 1998 [41], with a genome length of approximately 100 million bp. All of this work would eventually lead to the determination of the human genome, which is approximately 3.2 billion bp long.

The projects mentioned above, and many other similar projects, had significant impacts in our understanding of biology and the process of evolution. During this time two projects were proposed to sequence the human genome. One project was an international consortium of government and educational resources, and the other was a privately funded project from the Celera corporation. Although they had the same goal, they took different approaches to achieving their goals. The publicly funded Human Genome Project used bacterial artificial chromosomes to sequence DNA fragments, and then tried to find overlaps in the sequences. The privately funded project used the newest sequencing techniques to determine short piece of DNA, and then tried to find overlap in the short DNA sequences. In 2001 both projects published their initial drafts of the human genome [19, 40].

Recently, the use of Next Generation Sequencing (NGS) technologies has replaced Sanger sequencing as the preferred method of DNA sequencing. NGS technologies have a much lower cost per DNA base than the Sanger method, and produces much more information per run. The dominant technology is the Illumina platform which can sequence fragments of DNA, called *reads*, that are between 100 bp to 300 bp long. New technologies have been developed that produce much longer reads than the Illumina technology, but at a much higher cost, and there are many more errors in the reads.

1.1.2 Next generation sequencing

NGS technologies have provided an immense increase in DNA sequencing throughput over the last few years. Although there have been many technologies produced over the last two decades, the Illumina technology has taken over as the dominant sequencing platform due to its low cost per base, high throughput, and high quality of reads. The details of the current Illumina technologies are listed in Table 1.1.

Table 1.1: Output information for Illumina large scale sequencing platforms. (a) Run in high-output mode. (b) Run in rapid run mode.

Sequencing Platform	Maximum Read Length	Maximum Reads Per Run	Maximum Output
NextSeq	2 x 150 bp	400 million	120 Gb
HiSeq 2000	2 x 100 bp	2 billion	200 Gb
HiSeq 2500 (a)	2 x 125 bp	4 billion	1000 Gb
HiSeq 2500 (b)	2 x 250 bp	600 million	300 Gb
HiSeq 3000	2 x 150 bp	2.5 billion	750 Gb
HiSeq 4000	2 x 150 bp	5 billion	1500 Gb
HiSeqX Ten	2 x 150 bp	6 billion	1800 Gb
NovaSeq	2 x 150 bp	20 billion	6000 Gb

The Illumina technologies rely on an adapter library that is attached to a surface called a *cluster station*. The fragments are amplified in a process called *bridge amplification*, which uses DNA polymerase to make copies of the DNA fragments. Each cluster of a DNA fragment contains approximately one million copies of the original fragment. An example of the bridge amplification process is depicted in Figure 1.3. This is needed in order to produce an image that is strong enough to detect with a camera.

The system tries to add any of the four deoxyribonucleotides simultaneously to the fragments in each step. Three of the four deoxyribonucleotides contains a unique fluorescent label, and a chemical that blocks any other base from being incorporated. An image is taken of the

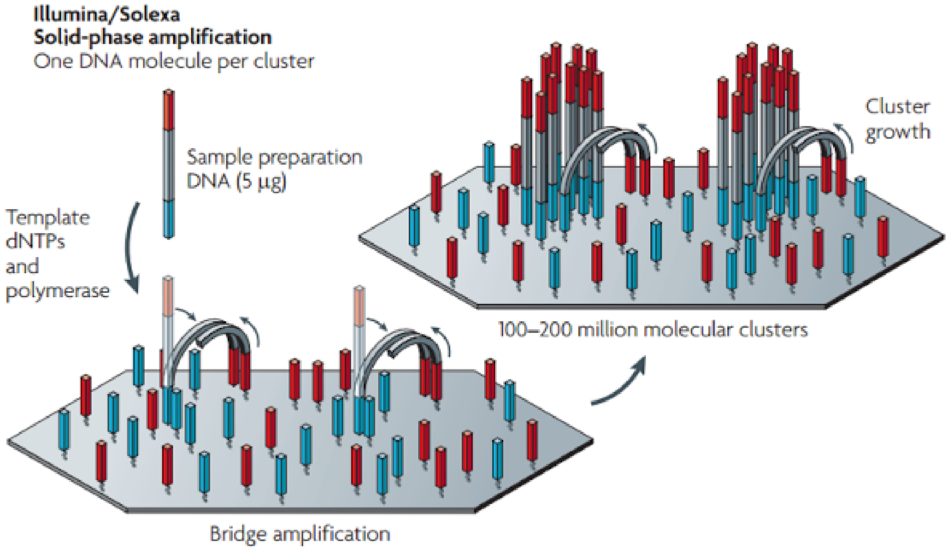


Figure 1.3: Bridge amplification of DNA fragments in the Illumina technologies. Primers are attached to single stranded DNA fragments, which then attach to the cluster station. Many copies of the fragment are made into a cluster of the same DNA fragment which can then be sequenced. [25].

incorporated base before the next deoxyribonucleotide is added. After each imaging step, the chemical block is removed so that the next deoxyribonucleotide can be added. An example of the imaging process is depicted in Figure 1.4. This procedure is repeated to determine the sequence of the fragment.

Another important feature of NGS technologies is the use of *paired-end* reads. Two reads are paired-end reads when they are sequenced from a single fragment of DNA from opposite ends of the fragment. The fragments will vary slightly in length, but the length of the reads taken from each side of the fragment will be known, and the DNA that is between the two reads is unknown. Repetitive regions in genomes are difficult to assemble, and paired-end reads are used in genome assembly to resolve the assembly around the repetitive regions.

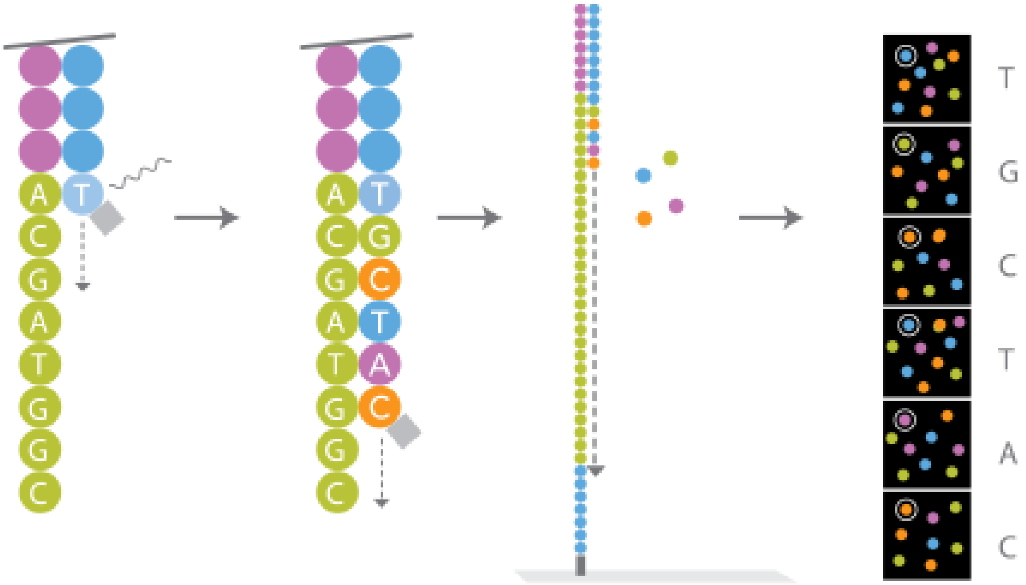


Figure 1.4: Imaging of incorporated bases in the fragment clusters of the Illumina technologies. Each deoxyribonucleotide is added one at a time and then the fluorescent label is excited and an image is taken of the incorporated deoxyribonucleotide. The chemical block is removed so the next deoxyribonucleotide can be added, and then the process is repeated. [3].

1.2 Error correction

There are two types of errors that are predominant in NGS technologies; *substitution* errors happen when a single base has been changed to a different base, and *indels* are bases in a read that have been inserted or deleted. Correcting errors in the reads can greatly increase the performance of applications that use NGS data such as genome assembly and variant calling. The Illumina platform predominantly makes substitution errors, and since it is the dominant technology most software developed for correcting errors focuses on substitution errors. Some of the most successful error correction programs that correct substitution errors include BLESS [10], Coral [32], HiTEC [13], Musket [23], RACER [12], SGA [37], and SHREC [35].

All error correction programs make use of "*k*-mers" in a read to find and correct errors. A *k*-mer is a subsequence of a read with length *k*. For example, consider a read with the DNA sequence ATCTCTG. If we chose a *k* value of 3, then this read would have five 3-mers (ATC, TCT, CTC, TCT, CTG), and four unique 3-mers (ATC, TCT, CTC, CTG).

All read error correction programs require a certain level of *coverage* in order to make accurate corrections. If G is the length of the genome being sequenced, and N is the total number of bases in the data set, then the coverage of the data set is the integer value of N/G. Theoretically, if the coverage of a data set is c, then each DNA base in the genome should be represented c times in the data set, assuming the coverage is uniform. Although, real data sets do not have a uniform coverage of the sample genome, some regions will be covered more than others, and some regions may not have any coverage at all.

There are three main approaches that are used to correct substitution errors; k-mer spectrum, k-mer counting, and multiple sequence alignment. k-mer spectrum correction algorithms correct errors in reads by trying to maximize the k-mers in a read to those that appear most often in the data set. Both BLESS and Musket use the k-mer spectrum approach to correct errors. k-mer counting algorithms count the number of times each k-mer appears in the data set, and they correct the low frequency k-mers in a read to k-mers that appear more than a set threshold. HiTEC, SGA, SHREC, and RACER all use the k-mer counting approach to correct errors. The multiple sequence alignment approach relies on aligning multiple reads that are similar, and correcting each read based on which base appears most often in the alignment at a particular position in each read. Coral uses the multiple sequence alignment method to correct errors in reads, which is also capable of correcting indels. SGA has an option to use inexact overlaps in reads to correct errors, but it is not the default method used.

1.3 Genome assembly

Current DNA sequencing technologies are not capable of sequencing the entire genome of a species, only small pieces of the genome can be sequenced, and these pieces must be assembled to determine the genome of the sampled species. It is possible to use a reference genome to map the reads to the reference genome. Although, many genomes do not have a reference genome yet, and some that have a reference genome are not reliable. Using *de novo* genome assembly is important in finding the sequence of genomes that have no reference, and to identify the genome sequences of individuals without the bias of a reference genome.

The first *de novo* genome assemblers used a *greedy method* to assemble genomes from reads. In this method overlaps are found between reads and scored based on the number of bases that overlap. The overlapping reads with the highest score would be merge first and then the process would be repeated. This approach makes many mistakes if there are a large amount of errors in the reads, or if there are many repeat regions in the genome.

A method was proposed by Kececioglu and Myers [16] to make more accurate assemblies that was a variation of the shortest common substring problem. Their approach made use of an overlap graph where each read is represented by a vertex in the graph, and each edge between two vertices represents an overlap between the reads of each vertex. The algorithm had four phases of assembly; graph construction from overlapping reads, assigning orientation of the reads, finding sets of overlaps that form a consistent layout with the orientated reads, and merging the reads based on a consensus multiple sequence alignment. This approach is called the overlap-layout-consensus (OLC) model of *de novo* genome assembly.

The first program developed to deal with whole genome assembly of mammalian sized genomes was the Celera assembler [29]. The Celera assembler was first used to assemble the *Drosophila melanogaster* genome, and it was used to assemble the first draft of the human genome. The Celera assembler has been updated to use different sequencing technologies and released under the name CABOG [26]. Other *de novo* assemblers that use the OLC method include ARACHNE [1], Edena [11], SGA [37], and SAGE [14].

For over 20 years the OLC approach was the only method used for *de novo* assembly until a new approach was proposed by Pavel Pevzner [30] that was based on k-mers in reads. The vertices in the graph are the k-mers in the reads, and the edges between the vertices are k-mers with exact overlaps of $k - 1$. This method is referred to as the *de Bruijn graph* (DBG) method of *de novo* genome assembly because it is based on graphs used by Nicolaas de Bruijn [4].

Figure 1.5 illustrates the difference between the OLC and DBG methods. For the given

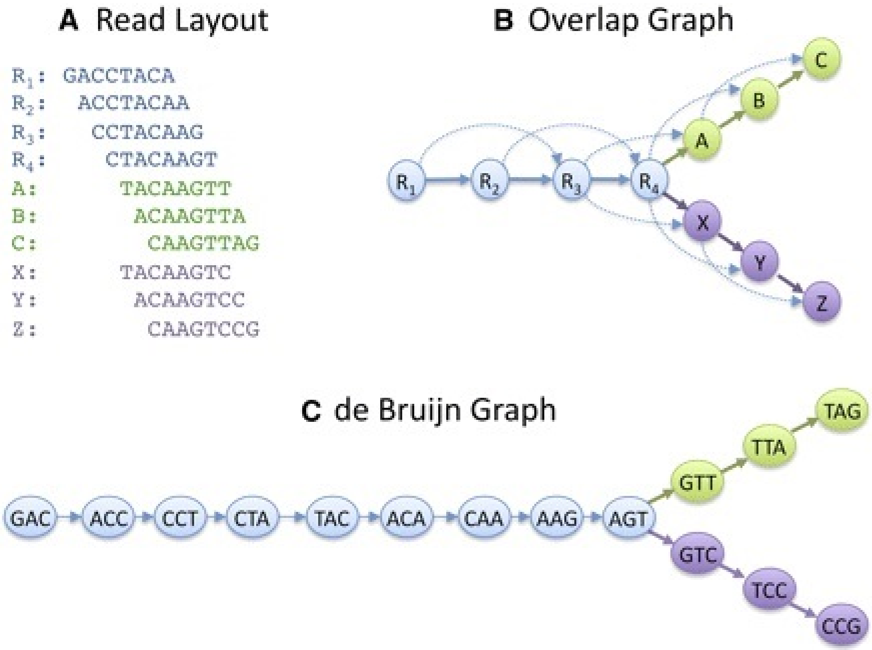


Figure 1.5: An example of the OLC and DBG methods for building a graph from overlaps the reads. (A) The layout of overlaps between the reads. (B) The overlap graph from the reads in (A), with transitive edges represented by the curved lines. (C) The de Bruijn graph of the reads in (A). [34].

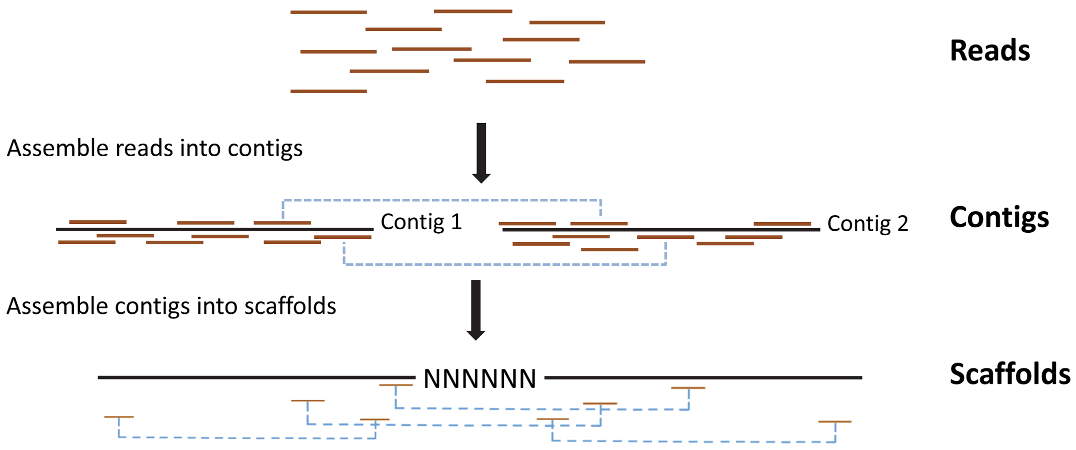


Figure 1.6: Overlaps in the reads or k-mers are used to build contiguous sequences called contigs. Paired-end reads or mate-pairs are used to link and orient the contigs to build scaffolds. Unknown gaps between contigs are filled with N's in the scaffolds.[15].

set of reads in Figure 1.5 (A), the overlap graph in Figure 1.5 (B) is build by finding overlaps between reads of at least a minimum length of 6 bases. The curved edges in Figure 1.5 (B) are called *transitive edges*. Transitive edges in the overlap graph are redundant because the strings spelled by the straight edges in Figure 1.5 (B) can be used to find the strings in the sampled genome without the transitive edges. OLC assemblers remove transitive edges in order to significantly reduce the space needed to store the overlap graph.

Figure 1.5 (C) shows the DBG approach for a k-mer length of 3 given the set of reads in Figure 1.5 (A). k-mers that overlap by $k-1$ have an edge in the DBG, and it should be noted that there are no transitive edges in a DBG. Since its inception the DBG has been the main strategy used by *de novo* genome assemblers including ABySS [38], ALLPATHS [2], SOAPdenovo [21], and Velvet [45].

Both approaches try to build contiguous sequences of overlapping reads or k-mers, which are referred to as *contigs*. The contigs are then orientated and connected with gaps between them using paired-end reads, with each of the two reads appearing in different contigs. The linked and oriented contigs are referred to as *scaffolds*. This process is shown in Figure 1.6.

The more paired-end reads that connect two contigs, the more likely the two contigs should be linked into a scaffold. The length of the gaps between contigs in a scaffold are estimated using the average distance between the paired-end reads on each contig.

1.4 Overview of my work

In chapter 2 I describe the implementation of an error correction program for DNA sequencing data called RACER. The algorithm uses binary storage of the DNA bases using two bits per base into 8-bit integer arrays to reduce space usage. Bitwise operations are used to correct a data set to reduced the runtime of the program, and a hash table is used to quickly access information about the *k*-mers in a data set. I show an evaluation of RACER against leading error correction programs using real sequencing data from a variety of organisms.

In chapter 3 I explain the details of an evaluation of error correction software, the methods used, and the implementation of two programs to perform the evaluation. I explain the need for a standardized methodology for comparing the results from error correction software, and the four measures introduced in the evaluation which provide a thorough and unbiased way to compare error correction programs. I outline the algorithms of the two programs that perform the evaluation. One program evaluates the *k*-mers in the reads using a hash table to find *k*-mers in a reference genome, and the other program evaluates whole reads using a suffix array to find the reads in a reference genome. The programs are evaluated using a variety of data sets from both Illumina HiSeq and MiSeq machines, including three human data sets from the Illumina HiSeq machine.

In chapter 4 I describe the implementation of a *de novo* genome assembly program for DNA sequencing data called SAGE2. It is a reimplementation of a previous program called SAGE, which addresses some of the key issues of the SAGE algorithm. The bottleneck in the assembly process of SAGE is the building of the overlap graph in serial, and I outline the details of a parallel implantation of overlap graph construction. I explain the details of the new algorithm for graph cleaning and merging of edges in the overlap graph. I then show the results of the new implementation compared to leading *de novo* genome assembly programs, including the assemblies of six whole human DNA sequencing data sets.

Finally, in chapter 5 I summarize the conclusions of the work outlined in this paper, and the possibilities for future research for the programs detailed throughout this work.

Chapter 2

Error Correction: RACER

The work outlined in this chapter is a continuation of the work from our previously published program called RACER [12]. RACER is an error correction program designed to correct substitution errors from NGS technologies. The previous implementation of RACER was unable to correct high coverage data sets from large genomes, such as the human genome. I first introduce the RACER algorithm, and the details of its automatic parameter selection. I then outline the details of the methods used to store the information in a data set in binary, the bitwise operations used to correct *k*-mers in the reads, and the implementation of a hash table which can store an retrieve information about the *k*-mers in the reads quickly and efficiently. Next, I explain the algorithm to detect and attempt to correct errors in the *k*-mers of reads for a data set. Finally, I show the results of RACER compared to leading error correcting programs using real DNA sequencing data sets from a variety of organisms for reads that are both mapped and unmapped to a reference genome. Due to the work outlined in this chapter RACER is now capable of correcting high coverage data sets from large genomes.

2.1 RACER algorithm

The algorithm for RACER has remained the same from the published version. The modifications since the publication have been made to the implementation of the functions and the automated parameter selection for correction. The algorithm for RACER is listed in Algorithm 1. The following sections will outline how RACER works and the changes that have been made.

RACER detects errors in reads by utilizing a time and space efficient hash table. This allows RACER to search for information about each k-mer in near constant time. The hash table stores the k-mers in each read, and the total number of times each base appears before and after each k-mer. The optimal k-mer lengths and threshold values are calculated automatically so that users do not have to determine them on their own. After the k-mers and their occurrences have been calculated, the reads are corrected based on the parameters that were calculated.

Algorithm 1 RACER: Correct errors in reads.

```
Input: Uncorrected data set $U$, estimated genome length $L$.
Output: Corrected reads.
$BRA \leftarrow BinaryReadsArray(U)$                                ▷ Store the reads in binary.
$MaxReadLength \leftarrow FindMaxReadLength(U)$                     ▷ Find maximum read length.
$KmerLengths(k, K, L, MaxReadLength)$                             ▷ Compute $k$-mer lengths $k$ and $K$.
for $1 \le i \le 10$ do
    if $i = 1 \,||\, i = 2$ then
        $kmerLength = k + 1$
    if $i = 3 \,||\, i = 4$ then
        $kmerLength = K + 1$
    if $i = 5 \,||\, i = 6$ then
        $kmerLength = k - 1$
    if $i = 7 \,||\, i = 8$ then
        $kmerLength = K - 1$
    if $i = 9 \,||\, i = 10$ then
        $kmerLength = K + 1$
    if $i = 1 \,||\, i = 3 \,||\, i = 5 \,||\, i = 7 \,||\, i = 9$ then
        $T \leftarrow ComputeThreshold(L, kmerLength, MaxReadLength)$
        $HT \leftarrow BuildHashTable(BRA, MaxReadLength)$
    $CorrectErrors(HT, BRA, T, kmerLength, MaxReadLength)$
$OutputCorrectedReads(BRA)$
```

To save space RACER encodes the input sequences using two bits to represent each base. Storing the bases as characters would require eight bits per base, so this approach is up to four times more space efficient at storing the reads. A k-mer and its reverse complement are considered the same so we only need to store one of them, and in RACER only the *k*-mer with the smaller encoding key is stored. This decreases the amount of space used in the hash table by half.

To save even more space in the hash table, RACER splits high coverage data sets into equal sections of a coverage cutoff of 45. This value was experimentally determined to be high enough that each split would have enough information to properly correct each split. Data sets with a coverage level at or below 45 are corrected all at once without splitting the data.

RACER begins correcting the reads after finding the *k*-mers in the reads and counting the number of times the bases appear before and after each *k*-mer. Correction starts by finding the *k*-mers in a read with the same technique described previously. The counters for each *k*-mer are used to determine if a correction needs to be made in the base before and after the *k*-mer, and to decide what is the correct base. If the count for the before or after base is less than the threshold then it is considered an error, and the counters are searched for a base that is above the threshold. If there is another base above the threshold then the erroneous base is replaced with the correct base. If there is more than one base that is above the threshold then no corrections are made, since this is likely from a heterozygous part of the sampled genome. If the total count for the before or after base is above the threshold it is considered correct.

The hash table and the corrected read are updated before the next *k*-mer in a read is considered for correcting. An advantage of this implementation is that once an erroneous base is corrected, the next *k*-mer that is considered will contain the corrected base. This allows RACER to correct more than one error in a read in one iteration.

2.2 Automated parameter selection

In order to correct the reads with high accuracy RACER requires three parameters to be calculated; a threshold for determining if an error should be corrected, a *k*-mer length *K* that will minimize the number of false positives, and a *k*-mer length *k* that will maximize the number of corrections. RACER uses statistical analysis similar to HiTEC to determine the best possible values of the parameters. Unlike HiTEC, all statistical computations used in RACER are performed by the program itself, eliminating the need for additional software to be installed. The reader is referred to [13] for the details of the statistical analysis used for the parameter selection.

One of the main reasons that RACER is able to produce such high accuracy is that it varies the *k*-mer lengths used to correct the reads. The combination of *k*-mer lengths was chosen experimentally based on extensive testing on a wide range of data sets. The combination of *k*-mer length values has been modified since the publication of RACER to optimize all genome lengths and coverage levels. The order of the *k*-mer lengths that are used is listed in Equation 2.1. The original version of RACER would stop any time after four iterations of corrections if the number of corrections for any iteration was below a set threshold. The latest version of RACER uses ten iterations of corrections regardless of how many corrections are made in each iteration.

$$k+1, k+1, K+1, K+1, k-1, k-1, K-1, K-1, K+1, K+1 \tag{2.1}$$

One thing to note about the selected *k*-mer lengths is that RACER performs two iterations of correction for each *k*-mer length. This is because RACER has been implemented so that the hash table can be used repeatedly without rebuilding it if the *k*-mer length stays the same. RACER uses the same *k*-mer length for two iterations to reduce the number of times it builds the hash table in half. Rather than building the hash table ten times for each iteration of correction, RACER only needs to build the has table five times.

2.3 Efficient *k*-mer storage and retrieval

RACER requires that the input files are either in *FASTA* or *FASTQ* format. If the reads are in FASTA format then there are two lines for each read. The first line of each read always begins with the ">" symbol, followed by a string that identifies the read. The second line contains the DNA letters of the read that was sequenced. An example of a FASTA file with two reads is shown in Figure 2.1.

If the reads are in FASTQ format then each read will occupy four lines. The first two lines for each read are similar to FASTA files, except that the first character of the first line is the "@" symbol instead of the ">" symbol. The first character in the third line of each read in a FASTQ file is either a "+" or "-" symbol to indicate if the read has been sequenced in either the $5' \rightarrow 3'$ direction, or the $3' \rightarrow 5'$ direction. The fourth line for each read is the quality score of each base and can start with any character above the ASCII offset of the read quality. The quality score is a quantitative way of determining how likely the base at that position is correct. RACER does not use quality values to correct reads, so this information is not stored. An example of a FASTQ file is shown in Figure 2.2.

```
>SRR065202.1 length=42
GAGCGTTAATCGGAATAACTGGGNGTNAAGGGCACGCAGGCG
>SRR065202.2 length=42
CTAATCCTGTTTGCTCCCCACGCTTTCGCACATGAGCGTCAG
```

Figure 2.1: An example of two reads from a FASTA file. The first line for each read starts with the ">" symbol followed by the identifier for the read and the second line for each read is the DNA sequence of the read.

```
@SRR065202.1 length=42
GAGCGTTAATCGGAATAACTGGGNGTNAAGGGCACGCAGGCG
+SRR065202.1 length=42
7BBC>CCBCCBBB>>CCCBC@@B!A@!BB2;3AAB>@;>+@#
@SRR065202.2 length=42
CTAATCCTGTTTGCTCCCCACGCTTTCGCACATGAGCGTCAG
+SRR065202.2 length=42
BCBACBCBB=BAABBA@BBBBABC>BB@BBB>B?=+A>?=A@
```

Figure 2.2: An example of two reads from a FASTQ file. The first line for each read starts with the "@" symbol followed by the identifier. The second line for each read is the DNA sequence of the read. The third line for each read starts with either the "+" or "-" symbol followed by the same identifier from the first line for the read. The forth line for each read are ASCII characters that represent the quality scores for each base in the read.

The reads that are sequenced from NGS technologies contain the four letters of the DNA alphabet, and bases that could not be determined are represented by the letter N. Because of the encoding used in RACER only the four letters in the DNA alphabet can be stored in the hash table. For this reason, any character in the reads that is not from the DNA alphabet is randomly replaced by a letter from the DNA alphabet. This allows RACER to correct all of the data in the data set, and since the ambiguous base must be one of the four bases in the DNA alphabet, RACER has a $1/4$ chance of randomly selecting the correct base at that position. If it is not the correct base that is replaced then it will most likely be changed to the correct base during the correction process.

To save space RACER encodes the reads in two bits per base instead of a string of characters. The 2-bit encoding of the DNA bases in RACER represents A as 00, T as 11, C as 01, and G as 10. As mentioned previously, any ambiguous base is randomly converted to one of the four DNA bases before encoding it. We performed testing of RACER to replace ambiguous bases with a non-random DNA letter, but it was determined that this can create problems when correcting, and when using downstream applications. If there are many long stretches of ambiguous bases in the reads, then replacing them by all A's can cause a k-mer with all A's to be above the set threshold. Therefore, the stretch of A's will not be corrected even when they should, causing problems for programs that use the corrected data, such as *de novo* genome assemblers. This is avoided by randomly replacing the ambiguous bases before correction.

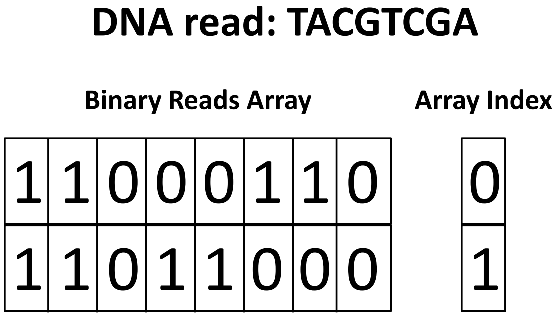
Not only does using 2-bit encodings save space, it also increases the speed of the program. This is because performing bit operations on data is much faster than using operations on characters. The logical bit operations used in RACER are AND, OR, and NOT. The AND operation is used in RACER with a mask to extract k-mers from the reads. A mask is a sequence of bits used to filter out unwanted bits from another sequence of bits. A mask contains zeros in positions that are not wanted, since the results will always be zero regardless of what is in the other sequence of bits. A mask is set to one in positions that are wanted, since the results will be whatever was in that position in the sequence of bits.

The bases are encoded in such a way that finding the reverse complement of a k-mer can be found quickly by using the NOT logical operation. A NOT operation flips all the bits so that 0's become 1's, and 1's become 0's. The DNA letter A is the complement of T, so performing a NOT on 11 will results in 00, and from 00 to 11. The same is true for G and C, which flips the bits from 10 to 01, and from 01 to 10. After the bits are flipped the bits just need to be reversed to give the reverse complement. This implementation is more time efficient than using a character array, since bit operations are faster than comparing characters to determine the reverse complement.

The reads are stored sequentially in an unsigned 8-bit integer array, which is initialized to zero for each element. This means that each unsigned 8-bit integer array can store up to four DNA bases from a read. Most machines are byte addressable, which means the smallest number of bits they can access is eight bits, so storing each 2-bit base requires a mask and bit shifting.

To understand this concept consider a read that contains the DNA letters TACGTCGA. Each element in the array that will contain the encoded read initially contains all 0's. The first letter is a T and is encoded to 11, then shifted left 6 positions and logically OR'd with the array element. The OR operation will put a 1 in the result if either of the operands contains a 1, and 0 otherwise. This array element will now contain 1100 0000. The next base is an A and is encoded to 00, then shifted left four positions and logically OR'd with the array element. This array element will now contain 1100 0000. The next base is a C and is encoded to 01, then shifted left two positions and logically OR'd with the array element. This array element will now contain 1100 0100. The next base is a G and is encoded to 10, then logically OR'd with the array element. This array element will now contain 1100 0110.

At this point the first 8-bit integer array holds four bases and is now full, so the index value for the array of 8-bit integer arrays is incremented to the next 8-bit integer array. The process is repeated until all the bases in the read are encoded. If the read length is not a multiple of eight then the last index of the array will not be filled. The rest of the bits of that element will



DNA read: TACGTCGA

Binary Reads Array								Array Index
1	1	0	0	0	1	1	0	0
1	1	0	1	1	0	0	0	1

Figure 2.3: An example of the 2-bit encoding used in RACER of the DNA bases TACGTCGA. The letter T is stored as 11, the letter A is stored as 00, the letter C is stored as 01, and the letter G is stored as 10. An 8-bit integer array stores the 2-bit encoding of the bases in each read, and another array stores the index to keep track of the locations of the bases in each read.

be left as 0's. At most there will be six unused bits for each read, which is still a much more space efficient way to store reads compared to using a character array. An integer array is used to store the length of each read so that the number of bits not used in the last array element for each read can be calculated. The result of storing TACGTCGA in the binary reads array is shown in Figure 2.3.

Many data sets contain reads that are mostly ambiguous bases. These reads do not contain much reliable information from the sampled genome, so correcting them would waste time and space. To deal with these types of reads in a data set, RACER does not correct reads if more than half the bases in the read are ambiguous bases. In order to mark these reads in the data set, the integer array that is used to store the read lengths of each read is set to a positive value if the read has < 50% ambiguous bases, and a negative value if the read has ≥ 50% ambiguous bases. The read is not removed from the final output, leaving the decision to use it or not to the downstream applications.

2.4 Hash table and hashing function

In order to make corrections to the reads RACER needs to find information about the *k*-mers quickly. A hash table is used to find information about each *k*-mer in near constant time. A hash table uses a *key* for each element that is input to a *hashing function* to find the element in the hash table. The 2-bit representation of each *k*-mer is used as a unique binary number that represents the input key for the hashing function used in RACER. The hashing function in RACER takes the input key modulo the hash table size, and the result is the index in the hash table to store the *k*-mer. RACER uses a hash table size that is a prime number selected from a precomputed list of prime numbers. The initial size of the hash table in RACER is set to the first prime number that is greater than eight times the genome size. This was determined experimentally in order to optimize both the run time and memory usage.

Hash tables with good hashing functions can store and retrieve elements in near constant time. The main problem with hash tables is when a hashing function maps different keys to the same location in the hash table. When this happens we call it a *collision*, and it is a common issue with hash tables. One way to deal with collisions is by using open addressing, which requires a search of the hash table for an empty entry to store the element being inserted into the hash table. The three most common open addressing solutions to collisions are *linear probing*, *quadratic probing*, and *double hashing*. The most successful hashing techniques try to distribute values evenly throughout the hash table to avoid collisions.

Linear probing is the simplest way to find an empty location to store a new element, and it was the fist type of collision resolution used in RACER. The way it resolves collisions is to look at the next index in the hash table, or any constant value of index values, until an empty location is found. This approach is fast because of cache effects, but it tends to cluster elements in the hash table which can dramatically increase the search time when the clustering gets too large.

Quadratic probing is similar to linear probing, except that instead of searching the next index, or constant value of indexes in the hash table, a quadratic formula determines the next index searched. This method is slower than linear probing, since it does not take advantage of cache effects. Although, there is usually less clustering than the linear method, which can improve performance for bad hash functions.

Double hashing requires a second hash function to handle collisions. If there is a collision using the main hashing function, then the second hashing function is used repeatedly until an empty location is found. This is the technique that is used in the current version of RACER.

Another issue that must be handled when using hash tables is the *load factor*. If e is the number of elements inserted into the hash table, and n is the number of indexes in the hash table, then the load factor is e/n. If the load factor gets too large then the search time for elements in the hash table increases such that it is no longer close to a constant time search. Conversely, if the load factor is near zero then the search time will be near constant time, but there will be a large amount of wasted space.

RACER does not allow the load factor to go above 50%. Once the load factor of the hash table is greater than 50% RACER stops building the hash table, and then increases the size of the hash table to the prime number closest to twice the previous hash table size. The original hash table is deleted before creating the larger hash table to save space, and the k-mers and counters are recalculated.

2.5 Counting bases adjacent to the k-mers

The counters for the bases appearing before and after each k-mer are stored in an 8-bit unsigned integer array. Each block of eight elements in the counters array stores the number of times each of the four DNA bases appears before and after each k-mer. The first four elements represent the four possible bases before the k-mer, and the next four elements represent the four possible bases after the k-mer. Since the counters array is 8-bits, the maximum k-mer occurrences that can be counted is 255, which is well above any threshold that would be set for a base to be considered correct.

The process of finding *k*-mers and incrementing the counters starts by copying the current read into a temporary array. The size of the array is set to twice the size of the longest read, since two bits are needed for each letter in the read. The last 64 bits of the current read are loaded into the current 64-bit window. Another 64-bit window is used to store the reverse complement of the current 64-bit window. Once the end of the 64-bit window is reached, the next 64 bits of the read and its reverse complement are loaded into the temporary arrays.

Another *k*-mer window is created which is twice the *k*-mer length, since each base is encoded using two bits. This *k*-mer window is then aligned with the rightmost end of the 64-bit window. The *k*-mer is obtained with a logical AND operation between the *k*-mer mask and the 64-bit window. The *k*-mer mask contains bit values of 1 inside the *k*-mer window and 0 elsewhere.

A similar procedure is used for the reverse complement of the current 64-bit window. The bases that are before and after the *k*-mer are extracted using a similar masking procedure. Each mask is set to all 0's, except for the 2-bit locations where the bases before or after the *k*-mer are located. An example of this procedure is shown in Figure 2.4. After storing a *k*-mer and incrementing its counters, as shown in Figure 2.4(a) , the *k*-mer windows shifts two bits to find the next *k*-mer in the read, as shown in Figure 2.4(b). The hash function is then used to store or find the *k*-mer in the hash table. If the *k*-mer already exists then the appropriate counters are incremented, if it is not in the hash table it is added and the counters for the bases before and after the *k*-mer are set to 1.

The next *k*-mer in the read is found by shifting the *k*-mer window two bits to the left in the current 64-bit window, and two bits to the right in the reverse complement. The *k*-mer is stored if it is not in the hash table and the counters are incremented, then the window is shifted two bits again. This continues until all of the *k*-mers in the read are found. If l is the read length and k is the *k*-mer length, then the number of *k*-mers in a read is $l - k + 1$.

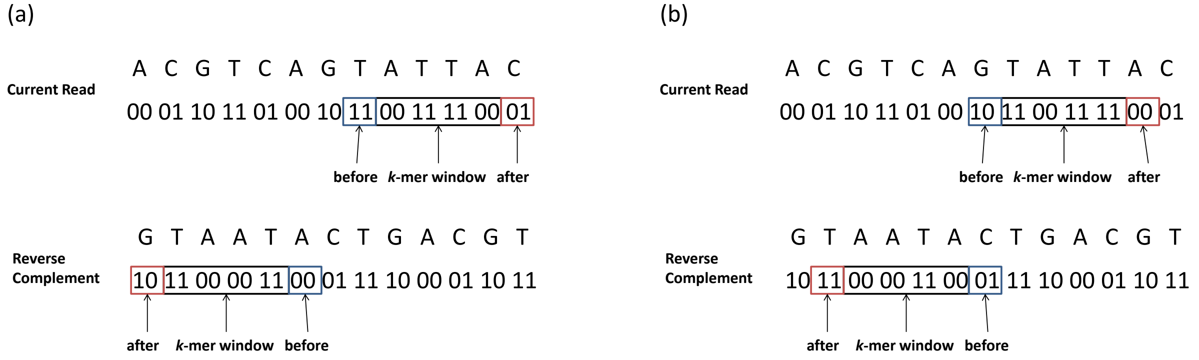


Figure 2.4: An example of finding and correcting adjacent bases for each *k*-mer in a read. (a) shows the *k*-mer window and bases before and after the *k*-mer window in the read and its reverse complement. (b) shows the *k*-mer window and bases before and after the *k*-mer window after the first *k*-mer was corrected. This process is repeated for every *k*-mer in each of the reads.

2.6 Correcting the reads

Once the *k*-mers have been stored in the hash table and the counters have calculated, the final step is to correct the errors based on the counts. The correction process begins by finding the *k*-mers in each read with the same procedure described in the previous section. The counters for each *k*-mer are used to determine if a correction needs to be made in both the base before and after each *k*-mer, and to decide what is the correct base.

If the total count for the base before or after the *k*-mer is above the threshold then it is considered correct. If the count for the base before or after the *k*-mer is less than the threshold then it is considered an error, and the counters are searched for a base that is above the threshold. If there is another base above the threshold then the erroneous base is replaced with the correct base. If there is more than one base that is above the threshold then no corrections are made since it is likely a heterozygous region of the sampled genome.

The hash table and the corrected read are updated before the next *k*-mer in the read is considered for correcting. An advantage of this implementation is that once an erroneous base is corrected, the next *k*-mer that is considered will contain the corrected base. This allows RACER to correct more than one error in a read in one iteration.

2.7 Results

To test the performance of RACER for the publication we obtained fifteen data sets from the Sequence Read Archive (http://trace.ncbi.nlm.nih.gov/Traces/sra/). The information for each data set is listed in Table 2.1. We choose data sets that varied in read length, coverage, and genome size. There was also a large variation in the error rates, most notably the high error rate of D7.

Table 2.1: Data sets used for testing in the RACER publication.

Data Set	Organism	Genome Length	Read Length	Coverage	Estimated Per-Base Error
D1	*Lactococcus lactis*	2,600,000	36	61	0.52%
D2	*Treponema pallidum*	1,100,000	35	219	0.89%
D3	*Escherichia coli*	4,600,000	75	56	0.65%
D4	*Bacillus subtilis*	4,200,000	75	63	0.58%
D5	*Escherichia coli*	4,600,000	75	70	0.62%
D6	*Pseudomonas aeruginosa*	6,300,000	36	53	0.09%
D7	*Escherichia coli*	4,700,000	47	142	3.65%
D8	*Leptospira interrogans*	4,300,000	100	163	0.26%
D9	*Leptospira interrogans*	4,300,000	100	167	0.21%
D10	*Escherichia coli*	4,700,000	36	157	0.46%
D11	*Haemophilus influenzae*	1,800,000	42	549	0.39%
D12	*Staphylococcus aureus*	2,900,000	76	669	1.75%
D13	*Saccharomyces cerevisiae*	12,400,000	76	319	0.72%
D14	*Caenorhabditis elegans*	102,300,000	100	66	0.35%
D15	*Drosophila melanogaster*	120,200,000	45,75,95	57	1.12%

Many of the publications from the competing software include correction of mapped data sets. Mapping a data set requires the reference genome of the species from which the reads were obtained. Each read is aligned to the reference genome with a certain number of mismatches allowed per read. The reads that were able to align to the reference are kept, and the reads that did not align are removed. This procedure filters out reads with many errors, which improves the performance of the error correction software. This is not usually done in practice, but for completeness we corrected both the unmapped and mapped data sets. We used a program called Burrows-Wheeler Aligner [20] to map the reads to their reference genomes using the default parameters.

2.7.1 Evaluation

RACER was compared to the top performing error correction software at that time. This included Coral [32], HiTEC [13], Quake [17], Reptile [43], and SHREC [35]. All programs were tested on the raw and mapped data sets. The competing programs were run according to the specifications in their respective manuals, websites, and readme files. All tests were run on the Shared Hierarchical Academic Research Computing Network (SHARCNET), with a HP 24 core 2.1 GHz AMD Opteron with 98GB RAM running Linux Red Hat, CentOS 5.6.

The data sets are measured in time, space, and accuracy. The time is measured in seconds, and space is the peak space reported by SHARCNET. The accuracy is calculated using the number of reads corrected (TP - true positives), the number of correct reads made incorrect (FP - false positives), and the number of reads with errors that were not corrected (FN - false negatives). The formula used to calculate the accuracy is listed in Equation 2.2, and the accuracy has been multiplied by 100 to represent it as a percentage.

$$\frac{TP-FP}{TP+FN} \tag{2.2}$$

Both HiTEC and Reptile could only run in serial mode, so we have tested all programs in serial to get a fair comparison of time and space used. We have also provided the results of the programs run in parallel to show how each performs using multiple processors.

2.7.2 Results of the unmapped data sets

The accuracy results of the unmapped data sets is listed in Table 2.2. The best performing programs are highlighted in dark green and the worst performing programs are highlighted in white. The previous error correction software with the highest accuracy was HiTEC. The results of our testing clearly shows that RACER outperforms all other programs in accuracy in most cases. HiTEC corrects more reads for some of the unmapped data sets, but it is by less than 1%. RACER had much higher accuracy results than HiTEC for the data sets D12 and D13 because HiTEC stopped after one iteration of corrections, whereas this version of RACER used eight iterations of corrections.

Table 2.2: Accuracy percentage of the unmapped data sets.

Data Set	Serial						Parallel			
	Coral	HiTEC	Quake	Reptile	SHREC	RACER	Coral	Quake	SHREC	RACER
D1	65.54	80.61	71.65	60.27	-	80.49	65.50	71.40	-	80.49
D2	38.55	84.45	59.46	2.65	61.79	85.75	38.54	59.94	61.77	85.75
D3	26.04	82.72	1.50	21.82	72.61	83.58	26.04	1.54	72.59	83.58
D4	59.76	80.59	53.59	64.25	41.19	82.12	59.76	53.60	41.12	82.12
D5	9.80	76.38	2.51	54.55	38.58	76.32	9.80	1.48	37.73	76.32
D6	79.78	78.68	7.08	68.44	63.40	85.32	79.75	13.44	63.40	85.32
D7	0.00	19.35	8.53	0.00	-	56.50	0.00	8.48	-	56.50
D8	48.25	60.23	49.75	35.55	55.99	59.87	48.25	49.76	55.90	59.87
D9	44.16	54.26	44.97	38.46	48.09	53.91	44.16	44.94	48.08	53.91
D10	58.02	85.89	81.38	0.06	77.49	86.32	58.02	81.46	77.46	86.32
D11	28.39	73.33	60.52	10.64	53.45	78.35	28.39	61.11	53.45	78.35
D12	0.02	0.03	15.36	0.03	-	25.96	0.02	15.38	-	25.96
D13	2.85	0.23	6.81	11.38	-	12.25	2.85	6.81	-	12.25
D14	-	-	38.88	0.21	-	56.54	-	38.86	-	56.54
D15	-	-	35.35	0.56	-	42.95	-	35.41	-	42.95
AVERAGE	43.64	75.17	40.09	32.94	56.96	76.84	43.64	40.81	56.84	76.84

HiTEC was not able to correct D15 due to the varying read sizes. Quake was not able to correct D11 and D12 due to a failed cut off value. SHREC was not able to correct D1 and D7 because of an error while reading the input. The rest of the missing results were due to the programs running out of space with 98GB of RAM.

2.7.3 Results of the mapped data sets

The accuracy results of the mapped data sets is listed in Table 2.3. The results for the mapped data sets is similar to the unmapped data sets. The difference is that the programs run faster, use less space, and have much better accuracy results. This is because the mapped data sets have less reads, with a minimal amount of errors per read. Reptile was the only program that had a lower accuracy with the mapped data sets compared to the unmapped data sets.

Table 2.3: Accuracy percentage of the mapped data sets.

Data Set	Serial						Parallel			
	Coral	HiTEC	Quake	Reptile	SHREC	RACER	Coral	Quake	SHREC	RACER
D1	75.22	92.16	81.67	0.11	84.87	92.02	75.21	81.76	84.87	92.02
D2	50.81	91.72	68.77	0.77	70.72	92.35	50.81	69.55	70.72	92.35
D3	26.49	83.08	1.51	0.07	73.05	83.91	26.50	1.51	73.05	83.91
D4	73.93	92.64	63.90	32.25	47.93	93.55	73.93	63.91	47.93	93.55
D5	11.65	78.22	1.42	0.07	40.35	77.80	11.65	0.83	40.35	77.80
D6	84.28	82.92	60.54	3.26	66.74	89.51	84.26	7.71	66.74	89.51
D7	3.78	77.00	45.79	0.02	56.44	82.67	3.78	45.79	56.44	82.67
D8	75.18	91.58	76.07	0.12	85.44	90.81	75.17	76.08	85.44	90.81
D9	75.35	90.05	75.33	1.61	80.31	89.27	75.35	75.20	80.31	89.27
D10	67.74	90.74	90.73	0.07	86.65	90.80	67.74	90.88	86.65	90.80
D11	48.65	80.04	69.60	8.32	60.66	84.33	48.41	69.02	60.66	84.33
D12	0.25	0.43	32.75	0.07	40.49	47.00	0.25	32.75	-	47.00
D13	5.97	0.30	8.92	0.30	11.94	14.68	5.98	8.92	11.94	14.68
D14	27.53	-	47.90	0.26	-	65.96	27.49	47.89	-	65.96
D15	40.12	-	47.01	0.00	-	57.04	40.11	47.01	-	57.04
AVERAGE	57.12	86.78	56.43	5.17	67.98	88.04	57.09	50.52	67.98	88.04

2.7.4 Time and space

The run time and space usage for the testing is listed in Table 2.4 for the unmapped data sets, and in Table 2.5 for mapped data sets. The time and space is listed in seconds per mega base pair (Mbp) to give an indication of which programs perform best in both time and space usage. The best programs at time and space usage are highlighted in dark green and the worst performing programs are highlighted in white. It is clear from these tables that RACER is the best performing program at time and space usage.

RACER was the fastest program in both serial and parallel modes for both the unmapped and mapped data sets. Quake was the second fastest in serial mode, but RACER completed the task in half the time in serial mode. RACER was one order of magnitude faster than all programs in parallel mode. Coral was much faster between serial and parallel modes, but also required 12 times the amount of space. RACER was significantly faster in parallel, but the increase in space was minimal. RACER used the least amount of space overall. Quake was the next best program at space usage, but it still used >50% more space than RACER in parallel mode.

Table 2.4: Run time and space for unmapped data sets in seconds/Mbp.

	Serial						Parallel			
Data Set	Coral	HiTEC	Quake	Reptile	SHREC	RACER	Coral	Quake	SHREC	RACER
D1	11.60	5.68	9.64	4.15	-	1.16	2.75	4.90	-	0.39
D2	30.76	11.07	14.18	9.52	22.57	3.28	5.55	6.01	3.66	0.89
D3	25.62	12.44	2.91	9.10	27.01	5.53	4.02	3.27	5.27	1.53
D4	24.60	12.37	4.37	7.90	25.11	4.24	4.37	3.27	5.22	0.73
D5	27.14	12.89	2.81	10.65	27.39	4.26	5.42	2.71	6.40	0.88
D6	11.46	2.87	4.58	14.85	17.69	1.28	2.25	3.60	3.68	0.33
D7	6.13	12.17	32.19	6.98	-	4.32	1.67	7.41	-	1.05
D8	95.48	11.63	2.66	4.95	26.88	3.37	13.05	2.05	5.22	0.60
D9	101.94	12.99	3.26	4.84	27.55	2.99	15.20	2.41	5.42	0.62
D10	26.77	13.45	13.31	6.01	19.14	2.12	4.53	6.08	3.78	0.84
D11	88.37	14.15	6.05	10.75	19.54	2.22	10.66	3.53	3.48	0.51
D12	76.80	2.03	7.84	15.93	-	4.60	9.32	2.68	-	1.13
D13	95.17	2.14	2.32	7.73	-	3.64	15.26	1.70	-	1.26
D14	-	-	3.11	16.13	-	5.30	-	1.77	-	0.87
D15	-	-	8.61	19.59	-	7.06	-	2.80	-	1.91
AVERAGE	48.02	11.54	6.01	8.73	23.65	3.25	7.23	3.66	4.68	0.77

Table 2.5: Run time and space for mapped data sets in seconds/Mbp.

	Serial						Parallel			
Data Set	Coral	HiTEC	Quake	Reptile	SHREC	RACER	Coral	Quake	SHREC	RACER
D1	11.11	5.49	8.40	11.73	17.50	1.05	2.10	5.49	3.17	0.31
D2	30.94	7.01	9.11	10.77	16.65	1.94	6.02	4.08	18.29	0.55
D3	24.36	12.42	2.87	9.59	26.08	3.85	3.88	2.57	5.23	0.76
D4	24.78	11.62	4.37	7.27	20.94	3.50	3.98	2.79	4.12	0.73
D5	27.48	12.97	2.79	8.59	29.09	3.82	4.09	2.98	6.35	1.14
D6	11.18	2.71	3.56	13.11	17.92	1.56	2.14	3.60	3.28	0.30
D7	2.64	3.68	14.02	4.96	8.10	1.27	0.71	5.46	1.85	0.26
D8	91.58	10.51	2.28	6.50	25.05	2.47	15.55	1.98	4.79	0.53
D9	96.53	11.86	2.61	4.42	24.21	1.94	15.21	2.41	4.93	0.48
D10	26.15	7.15	6.57	8.41	18.01	1.32	5.01	3.46	3.57	0.28
D11	28.03	7.72	4.77	9.38	18.18	1.32	14.70	5.59	3.50	0.34
D12	4.98	2.12	3.01	5.14	16.99	2.65	11.32	1.56	-	0.43
D13	94.75	1.87	2.07	9.14	22.36	3.15	14.39	1.61	4.56	0.57
D14	40.72	-	2.46	16.13	-	5.13	7.31	1.56	-	0.85
D15	23.18	-	7.02	22.50	-	4.57	4.48	3.20	-	0.94
AVERAGE	40.12	9.33	4.33	8.67	21.79	2.41	7.84	3.27	6.01	0.57

2.8 Conclusions

RACER was capable of correcting many errors in sequencing when data sets were derived from small and medium sized genomes, but it was not able to correct large genomes due its implementation. The main problems with RACER was the implementation of the hash table, and the automatic parameter selection for large genomes. We have reimplemented the hash table so that it can quickly and efficiently manage the information needed to make error corrections for any sized genome and coverage level. We have also performed extensive testing to determine the proper automatic parameter selection for all genome sizes and coverage levels. Our testing showed that RACER performed as good or better than the state-of-the-art for all genome sizes, coverage levels, and sequencing technologies from Illumina.

Chapter 3

Evaluation of Error Correcting Software

3.1 Introduction

Professionals that work with NGS data must often correct the errors in their data sets before running any applications that use the data. The problem is that there are many error correction programs that are capable of correcting errors in NGS data. Determining the best software to use for a particular genome or sequencing machine is difficult since there is no standard way of measuring the accuracy of the error correction. A survey was published that evaluated the performance of current technologies [44], but the results were biased by the methods used for evaluation, and more accurate programs have been released since its publication. There was a need for a better evaluation of error correction software that was not biased, more comprehensive, and one that evaluated the current state-of-the-art software.

In this chapter I first present the problems with the previous methods for evaluating error correction programs, and the goals for the survey. Next, I outline some notation and details of the methodology developed to standardize the evaluation of error correction programs in a thorough and unbiased manner. Then I explain the implantation of two programs developed to perform the evaluation of error correction programs. The first program uses a suffix array to find reads that align to a reference genome, and the second program uses a hash table to find *k*-

mers in the reads that align to a reference genome. Next, I show the results of our comparisons using a variety of data sets from both the Illumina HiSeq and MiSeq machines, including three whole human data sets from the Illumina HiSeq machine. Finally, I explain recommendations for professionals that would be in need of correcting data from NGS machines, and outline the conclusions from this chapter.

3.2 Problems with existing approaches

One of the previous methods used for evaluating the accuracy of error correction software was to use alignment software to map reads to a reference genome, and then mark the bases that were erroneous. The corrected data was then mapped to the reference, and the differences were used to determine the accuracy of the programs [10, 23, 32]. The problem with this method is that many reads map to multiple regions in a genome, and many cannot be mapped at all. These reads would be removed from the data sets before being corrected in order to evaluate the performance of the correcting software. This approach makes it much easier for error correcting software to correct the data sets, and it does not reflect what happens in practice. Also, the mapping of the reads is biased by the alignment software, and can differ significantly depending on the parameters used for the alignments.

Another method used for evaluating the performance of error correction software was to use genome assembly software before and after correcting the data set, and the N50 or NGA50 would be used to evaluate the performance of the error correction [10, 23, 32]. The problem with this method is that many genome assembly programs have their own error correction modules, and it is difficult to determine how correcting the data before will affect the performance. Also, genome assembly programs may have many parameters that can be changed, and trying all combinations of parameters to find the best is impractical. For these reasons it is unacceptable to evaluate error correction software based on the performance of genome assembly programs.

3.3 Goals for the survey

The lack of standardization for measuring the performance of error correcting software and the limited resources for evaluation were the main motivating factors for us to conduct this survey. We set a goal of standardizing the evaluation of error correction software, and to provide the tools for comparing current and future error correction programs.

3.4 Coverage depth and breadth

Previous methods of evaluating error correction performance relied on a measure of the *gain* in corrected base pairs in a data set [44]. This was considered an acceptable measure of performance, but the methods used to calculate the gain were biased by the approaches mentioned previously. To provide a thorough and unbiased evaluation we introduced the concepts of gain in coverage depth and breadth for both whole reads and k-mers. Most applications that use NGS data will either use the entire reads in a data set, or k-mers from each read in a data set, so it was important to evaluate both whole reads and k-mers for our evaluation.

Consider a reference genome G with length L. We will denote the ith deoxyribonucleotide of G using $G[i]$, and the subsequence starting at position i, and ending at position j in the genome using $G[i...j]$. Therefore, the entire sequence of the genome would be denoted $G = G[1...L]$.

Consider a data set D, with R representing the reads in D, and the total number of reads in D is equal to N, then $D = \{R_i | 1 \leq i \leq N\}$. The length of a read R is denoted $|R|$, and it is the total number of deoxyribonucleotides in R; e.g. $|ACGTA| = 5$. We will distinguish a read R from its sequence of deoxyribonucleotides $seq(R)$, since different reads can have the same sequence. We can then denote the sequence of reads in D as $seq(D) = \{seq(R) | R \in D\}$.

To distinguish between k-mers and their sequences we will introduce the concept of a "positional k-mer," which is the k-mer starting at position j in read R_i, $R_i[j...j+k-1]$, and denoted (k, i, j). The set of positional k-mers is denoted pos-k-mers $= \{(k, i, j) | 1 \leq i \leq N, 1 \leq j \leq$

$|R_i| - k + 1\}$. The set of k-mers occurring in D or G is denoted k-mer(D) and k-mer(G) respectively. For example, if we consider a data set $D = \{ACCT, ACCT, GGGG\}$, then it contains three reads, two sequences $seq(D) = \{ACCT, GGGG\}$, nine pos-2-mers, and four 2-mers 2-mer(D) = $\{AC, CC, CT, GG\}$.

3.4.1 Coverage

We have introduced four new measures of gain with the goal of providing a thorough and unbiased evaluation of the performance of error correction software. Correcting a data set can improve either the *depth of coverage*, which is the average number of times each base pair in the genome is covered, as shown in Equation 3.1, or the *breadth of coverage*, which is the proportion of the genome that is covered by the reads or k-mers.

$$\frac{1}{L}\sum_{i=1}^{N}|R_i| \tag{3.1}$$

In order to understand the breadth of coverage more clearly it is important to understand what it means to have a position in the genome covered by a read or k-mer. We consider all reads or k-mers starting at every position in the genome. For reads of length l, or in the case of k-mers we can replace l with k, we define the breadth of coverage as the ratio of the l-mers in the reference genome that appear in the data set, which is shown in equation 3.2.

$$\frac{|l - mer(G) \cap l - mer(D)|}{|l - mer(G)|} \tag{3.2}$$

Since the reads of a data set can be used as whole reads or broken into k-mers, we chose to evaluate the error correction performance with respect to each type of coverage. The four measures of evaluation we have introduced are: *ReadDepthGain*, *KmerDepthGain*, *ReadBreadthGain*, and *KmerDepthGain*.

The evaluation approaches we have introduced consider both the correction of whole reads and k-mers, but it does not evaluate the point correction of single errors. Although, counting point corrections of single errors is not necessarily relevant in practice, and this can be shown by considering two scenarios of error correction. In the first scenario, assume there are five errors that are corrected in a read that only has five errors. In the second scenario, assume there are five errors corrected in a read that has ten errors. While both scenarios correct a total of five errors each, the read in the first scenario becomes error free after correction. This is an important aspect of error correction because error free reads are much more useful in downstream applications than reads with errors. This is why we chose to evaluate error correction performance based on whole reads and k-mers instead of point corrections.

3.4.2 Gain in depth of coverage

In this section we will consider both the *ReadDepthGain* and *KmerDepthGain* measures of performance. A read R is considered to be correct if it is found in the reference genome: $seq(R) = seq(G)[i...i + |R| - 1]$ for some $1 \leq i \leq L - |R| + 1$. Otherwise, R is considered to be erroneous. Based on this definition, we can define a binary classifier on any data set; TP is the number of reads that are erroneous before correction but correct after correction, TN is the number of reads that are correct both before and after correction, FP is the number of reads that are correct before correction but erroneous after correction, and FN is the number of reads that are erroneous both before and after correction. The gain in depth of coverage for whole reads can then be defined by Equation 3.3. This represents the total number of correct whole reads gained, which is $TP - FP$, as a fraction of the total number of erroneous reads before correction, which is $P = TP + FN$.

$$ReadDepthGain = \frac{TP - FP}{P} \tag{3.3}$$

The *ReadDepthGain* measures the gain in depth of coverage given by the correct reads, but the quality of reads in the corrected data set is inversely proportional to the quality of the reads in the uncorrected data set. The proportion of the new correct reads out of the total number of reads is shown in Equation 3.4.

$$\frac{TP - FP}{P + N} = \frac{P}{P + N} ReadDepthGain \tag{3.4}$$

To properly evaluate correction we introduce Equation 3.5, which is the ratio between the number of correct reads in the original data set, and the total number of reads.

$$OrigReadDepth = \frac{N}{P + N} \tag{3.5}$$

We also introduce Equation 3.6, which is the ratio between the number of correct reads in the corrected data set, and the total number of reads. The relation between the original and corrected read depth, and the read depth gain is shown in Equation 3.7, which can be represented as shown in Equation 3.8.

$$CorrReadDepth = \frac{TP + TN}{P + N} \tag{3.6}$$

$$CorrReadDepth = OrigReadDepth + \frac{P}{P + N} ReadDepthGain \tag{3.7}$$

This works if all the reads in the data set have the same length, but to deal with data sets that have varying read lengths we have modified the above definitions. The modified version counts the contribution of each read R toward each of the values TP, TN, FP, FN as $|R|$ instead of 1. This produces the same values as Equation 3.8 when all the reads have the same length.

$$CorrReadDepth = OrigReadDepth + ReadDepthGain(1 - OrigReadDepth) \tag{3.8}$$

Correcting whole reads is important, but it is also difficult, which is why the percentage of the errors corrected becomes important. However, the ratio of errors corrected is not acceptable by itself to determine the quality of correction. Consider two reads with ten errors each, and both have five of the errors corrected. If we only consider error correction as a percentage, then both reads will have the same percentage of errors corrected. Assume that the first read has the five leftmost errors corrected, and the second read has every other error corrected. It is most likely that the first read will have a longer error free subsequence than the second read, which will make the first read much more useful in downstream applications.

The main point is that a corrected error becomes more useful when it creates a sufficiently long error free subsequence. To accommodate this situation we consider correction of k-mers, where the value of k is chosen to be the smallest possible value that guarantees, with a high probability, that there is a unique position in the reference genomes for each k-mer. The total number and length of the repeat regions in a genome varies significantly for each genome, so it is not possible to find a value of k that will allow for all of the k-mers in every genome to be unique. For the evaluation in our survey we chose $k = 20$, but the evaluation program that we have provided allows for the evaluation of any value of k between 5 and 32.

$$KmerDepthGain = \frac{TP - FP}{P} \tag{3.9}$$

To evaluate *KmerDepthGain* we use a similar approach as *ReadDepthGain*, but we must define it in terms of positional k-mers instead of whole reads. A positional k-mer (k, i, j) is called correct if it can be found in the reference genome: $seq(R_i[j...j + k - 1]) = seq(G[l...i + k - 1])$, for some $1 \leq l \leq L - k + 1$. Otherwise, the positional k-mer (k, i, j) is called erroneous. We can now define a binary classifier on pos-k-mers(D), similar to the one for whole reads. The depth of coverage for positional k-mer gain is defined as shown in Equation 3.9. The *KmerDepthGain* quantifies the gain in depth of coverage as given by correct positional k-mers. As we did for whole reads, to evaluate correction on the quality of the k-mers, we introduce Equation 3.10 and Equation 3.11 that satisfy Equation 3.8 with the appropriate modifications as shown in Equation 3.12.

$$OrigKmerDepth = \frac{N}{P + N} \tag{3.10}$$

$$CorrKmerDepth = \frac{TP + TN}{P + N} \tag{3.11}$$

$$CorrKmerDepth = OrigKmerDepth + KmerDepthGain(1 - OrigKmerDepth) \tag{3.12}$$

3.4.3 Gain in breadth of coverage

The depth of coverage is a property of the data set, whereas the breadth of coverage is a property of the genome. To illustrate this point we will consider two programs that correct two different unique reads R_1 and R_2. Assume that R_1 has twenty copies in the data set, ten of which are erroneous, and R_2 has ten copies in the data set, five of which are erroneous. Assume that the first program P_1 corrects all copies of R_1, but destroys all copies of R_2. Assume that the second program P_2 corrects two copies of R_1, and one copy of R_2, without destroying anything. Then the *ReadDepthGain* of P_1 is 0.33 and the *ReadDepthGain* of P_2 is 0.20. Therefore, based on *ReadDepthGain* alone, P_1 would be judged as the better of the two programs for correction. However, P_2 preserves copies of both reads, which results in a better breadth of coverage of the genome. Therefore, based on the breadth of coverage, P_2 would be judged as the better of the two programs for correction. This example demonstrates the need for both measures of performance in order to have a thorough evaluation of error correction.

We will introduce two measures for the gain in breadth coverage that complements the gain in read coverage. First, we need to define a binary classification on k-mer(D), similar to the whole read classifier. A k-mer K of the reference genome is considered "covered" if $K \in k$-mer(G). Otherwise, the k-mer is considered to be "not covered." The classifier TP becomes the number of k-mers that are not covered before correction but covered afterward, etc. The new measure is called *KmerBreadthGain* and is shown in Equation 3.13. The impact on the breadth of coverage is assessed using *OrigKmerBreadth* and *CorrKmerBreadth* defined as before, and satisfying Equation 3.12.

$$KmerBreadthGain = \frac{TP - FP}{P} \tag{3.13}$$

For whole reads, if all reads have the same length l, then $seq(D) = l\text{–mer}(D)$, and the definition of *ReadBreadthGain* is similar to *KmerBreadthGain* but with k replaced by l. If the reads are not the same length, then TP becomes the number of elements of $seq(D)$ that do not occur in G before, but do after correction, FP represents the opposite of TP, and TN represents those that appeared both before and after correction. For FN, we use $|l\text{–mer}(G)|$, where l is the weighted average read length. As above, we also introduce *OrigReadBreadth* and $CorrReadBreadth$ satisfying Equation 3.8.

Defining the breadth of coverage in this way, *KmerBreadthGain* measures the gain in the breadth of coverage as given by correct k-mers, and *ReadBreadthGain* measures the gain in breadth of coverage as given by correct whole read sequences.

3.5 Evaluation tools

We have provided two programs to calculate the four measures of performance for users that would like to evaluate error correction performance based on our methodology. The program that calculates the gain in depth and breadth of coverage for whole reads is called $readSearch$. This program that calculates the gain in k-mer depth and breadth of coverage is called *kmerSearch*.

3.5.1 The readSearch algorithm

The readSearch algorithm uses a *suffix array* and *longest common prefix* (LCP) array to store and search the reference genome. A suffix array is an alternative implementation of a suffix tree that stores the locations of the suffixes of a string in an array instead of using pointers. Consider the DNA string AGAAGAT, which can be stored in a character array with the first letter stored in location 0 of the array, and the last letter stored in location 6 of the array. A special character is added to the end of the string to mark the end point of the string. The special character is considered to have the largest lexicographical ordering so that it is always at the end of the suffix array for a string. A suffix array stores the starting locations of the lexicographically sorted suffixes of the string in the suffix array. If the special character for the above example is $, then the lexicographical ordering of the suffixes in the above example would be: AAGAT$, AGAAGAT$, AGAT$, AT$, GAAGAT$, GAT$, T$, and $. Therefore, the suffix array for this example would contain: 2, 0, 3, 5, 1, 4, 6, 7.

The LCP array is used to improve the search time for finding strings in the suffix array, and the algorithm was introduced by Udi Manber and Gene Myers [24]. This array stores the longest common prefix between an element in the suffix array, and the next element in the suffix array. For example, the longest common prefix between the first element in the above suffix array AAGAT$, and the second element in the suffix array AGAAGAT$ is 1. Therefore, the value in location 1 of the LCP array would be 1, and it should be noted that location 0 of the LCP array would contain 0 since there is no previous element to compare against. The full LCP array for the above example would contain: 0, 1, 3, 1, 0, 2, 0, 0.

The algorithm for readSearch is listed in Algorithm 2. The implementation of readSearch requires that the reads in the original and corrected data sets are in the same order. The implementation of readSearch also requires that none of the reads have been removed, but it does allow for reads to be trimmed. If a read has been trimmed then it is considered to not be found in the reference genome.

Two integer arrays are used to store the lengths of each read in the original and corrected data sets to determine if any of the reads have been trimmed. Two boolean arrays are used to store which reads are found in the original and corrected data sets. These arrays are set to false initially, and if the read is found in the reference genome then the corresponding array element for that read is set to true. Two more boolean arrays that are set to the length of the reference genome are used to mark the locations in the genome that are covered by the reads for both the original and corrected data sets. The positions are all set to false initially and set to true if the position in the genome is covered by a read. All of these arrays are used to calculate the gain results for the depth and breadth of coverage for whole reads.

Algorithm 2 readSearch: Calculate whole read depth and breadth gain.

```
Input: Reference genome $RG$, original data set $O$, corrected data set $C$.
Output: Whole read depth and breadth gain results.
$NR \leftarrow$ Number of reads in $O$.
$G \leftarrow Array(RG)$                          ▷ Store the reference genome in an array.
$GSA \leftarrow SuffixArray(G)$                   ▷ Build the suffix array.
$LCP \leftarrow LongestCommonPrefix(G, GSA)$      ▷ Compute the LCP array.
for $1 \leq i \leq NR$ do
    $FO[i]$= false     ▷ Array to store which reads from original data set are found in $G$.
    $FC[i]$ = false    ▷ Array to store which reads from corrected data set are found in $G$.
    $GO[i]$ = false    ▷ Array to store positions in $G$ covered by original reads.
    $GC[i]$ = false    ▷ Array to store positions in $G$ covered by corrected reads.
$TP_D = TN_D = FP_D = FN_D = 0$                   ▷ Initialize depth gain binary classifiers.
$TP_B = TN_B = FP_B = FN_B = 0$                   ▷ Initialize breadth gain binary classifiers.
for each read $r_i \in O$ do
    if $Occurs(r_i, RG, GSA, LCP, pos)$ then
        $FO[i]$ = true
        $GO[pos]$ = true   ▷ Marks each position in the genome that is covered by the read.
for each read $r_i \in C$ do
    if $Occurs(r_i, RG, GSA, LCP, pos)$ then
        $FC[i]$ = true
        $GC[pos]$ = true   ▷ Marks each position in the genome that is covered by the read.
for $1 \leq i \leq NR$ do
    if $FO[i]$ = false and $FC[i]$ = true then
        $TP_D + +$
    else if $FO[i]$ = true and $FC[i]$ = false then
        $FP_D + +$
    else if $FO[i]$ = true and $FC[i]$ = true then
        $TN_D + +$
    else
        $FN_D + +$
for $1 \leq i \leq NR$ do
    if $GO[i]$ = false and $GC[i]$ = true then
        $TP_B + +$
    else if $GO[i]$ = true and $GC[i]$ = false then
        $FP_B + +$
    else if $GO[i]$ = true and $GC[i]$ = true then
        $TN_B + +$
$FN_B = (genomeLength - averageReadLength) - (TP_B + FP_B + TN_B)$
$ReadDepthGain = (TP_D - FP_D)/(TP_D + FN_D)$
$ReadBreadthGain = (TP_B - FP_B)/(TP_B + FN_B)$
return $ReadDepthGain, ReadBreadthGain$
```

3.5.2 The kmerSearch algorithm

The kmerSearch algorithm uses a hash table to store the *k*-mers that are found in the reference genome. The implementation of the hash table is similar to the one used in RACER, and described in Section 2.4. The kmerSearch algorithm uses the hash table to quickly determine if the *k*-mers in the original and corrected data sets are in the reference genome. The algorithm for kmerSearch is listed in Algorithm 3.

The kmerSearch algorithm can search a data set for *k*-mers with a length of *k* between 5 and 32. The implementation requires that none of the reads have been removed or trimmed by the correcting program. The implementation also requires the reads to be in the same order in the original and corrected data sets for purposes of calculating the binary classifiers for the depth and breadth of coverage. The *k*-mers in the reference genome that contain any ambiguous bases are not stored in the hash table. The *k*-mers in the original or corrected data sets that contain any ambiguous bases are considered to not be found in the reference genome.

Two boolean arrays are used to calculate the breadth of coverage gain results. The arrays are set to the length of the hash table and the values are initially set to false. When a *k*-mer is found in the genome for the original or corrected data set then the corresponding location in the boolean arrays is set to true. When kmerSearch is finished processing the original and corrected data sets the values in the boolean arrays are used to calculate the breadth classifiers TP, TN, and FP. To calculate the FN value for the breadth classifier we subtract the total values for TP, TN, and FP from the total number of unique *k*-mers in the hash table for the reference genome.

3.6 Illumina HiSeq and MiSeq machines

The data sets we have selected are all real data sets that have not been altered in any way. At the time of the publication the main platforms from Illumina were the HiSeq 2000, HiSeq 2500, and MiSeq machines. HiSeq machines have large amounts of output and low error rates. MiSeq

Algorithm 3 kmerSearch: Find all k-mers that align to a reference genome.

```
Input: Reference genome $RG$, original data set $O$, corrected data set $C$, $k$-mer length $k$.
Output: $k$-mer depth and breadth gain.
$NR \leftarrow$ Number of reads in $O$.
$totalUniqueKmers = 0$        ▷ Variable to count the total unique $k$-mers in $G$.
$G \leftarrow Array(RG)$        ▷ Store the reference genome in an array.
for each $k$-mer $K \in G$ do
    $HT \leftarrow Hash(K)$        ▷ Store each $k$-mer in $G$ in the hash table $HT$.
    if first occurrence found of $K$ in $HT$ then
        $totalUniqueKmers++$
$TP_D = TN_D = FP_D = FN_D = 0$        ▷ Initialize depth gain binary classifiers.
$TP_B = TN_B = FP_B = FN_B = 0$        ▷ Initialize breadth gain binary classifiers.
for $1 \leq i \leq NR$ do
    $FO[i] =$ false        ▷ Array for breadth gain binary classifiers of the original file.
    $FC[i] =$ false        ▷ Array for breadth gain binary classifiers of the corrected file.
for $1 \leq i \leq NR$ do        ▷ Searches $k$-mers in $O$ and $C$ simultaneously.
    for each $k$-mer $K$ in read $r_i$ do
        foundOriginal = false
        foundCorrected = false
        if $Occurs(O, r_i, K, HT, pos)$ then
            foundOriginal = true
            $FO[pos] =$ true
        if $Occurs(C, r_i, K, HT, pos)$ then
            foundCorrected = true
            $FC[pos] =$ true
        if foundOriginal = false and foundCorrected = true then
            $TP_D++$
        else if foundOriginal = true and foundCorrected = false then
            $FP_D++$
        else if foundOriginal = true and foundCorrected = true then
            $TN_D++$
        else
            $FN_D++$
for $1 \leq i \leq NR$ do
    if $FO[i] =$ false and $FC[i] =$ true then
        $TP_B++$
    else if $FO[i] =$ true and $FC[i] =$ false then
        $FP_B++$
    else if $FO[i] =$ true and $FC[i] =$ true then
        $TN_B++$
$FN_B = totalUniqueKmers - (TP_B + FP_B + TN_B)$
$KmerDepthGain = (TP_D - FP_D)/(TP_D + FN_D)$
$KmerBreadthGain = (TP_B - FP_B)/(TP_B + FN_B)$
return $KmerDepthGain, KmerBreadthGain$
```

machines are bench top computers that are cheaper, have much lower amounts of output, and significantly higher error rates. Correcting errors in data sets produced by these three machines poses different challenges. This is the first work to make such a comparison.

3.7 Data sets used for evaluation

We used a variety of data sets from reference genomes ranging from bacteria to humans. They provide a good assessment of the actual performance of the correcting programs in practice, and the testing provides a clear indication of the current state-of-the-art in correcting Illumina data. We did not include simulated data sets because correction is much easier for simulated data sets, which gives the false impression that a high percentage of errors, often more than 99%, can be corrected.

Table 3.1: HiSeq data sets used for evaluation.

Data Set	Organism	Genome Length	Read Length	Coverage	Estimated Per-Base Error
H1	*Mycobacterium tuberculosis*	4,400,000	151	72	0.15%
H2	*Salmonella enterica*	4,900,000	100	67	0.20%
H3	*Saccharomyces cerevisiae*	12,400,000	100	40	0.20%
H4	*Legionella pneumophila*	3,400,000	100	260	0.40%
H5	*Escherichia coli*	4,600,000	101	255	0.73%
H6	*Escherichia coli*	4,600,000	100	465	0.68%
H7	*Caenorhabditis elegans*	102,300,000	100	32	0.38%
H8	*Caenorhabditis elegans*	102,300,000	101	58	0.36%
H9	*Drosophila melanogaster*	120,200,000	100	52	0.77%
H10	*Drosophila melanogaster*	120,200,000	101	64	0.90%
H11	*Homo sapiens*	3,210,000,000	100-102	43	0.73%
H12	*Homo sapiens*	3,210,000,000	100-102	52	0.70%
H13	*Homo sapiens*	3,210,000,000	101	54	0.23%

We have included 13 HiSeq data sets, and 9 MiSeq data sets, from a wide variety of reference genomes and with varying coverage levels. The details of the HiSeq data sets are listed in Table 3.1, and the details of the MiSeq data sets are listed in Table 3.2. We have included three whole human data sets, H11 - H13. The data sets H1, H4, H5 and H11 are from HiSeq 2500

Table 3.2: MiSeq data sets used for evaluation.

Data Set	Organism	Genome Length	Read Length	Coverage	Estimated Per-Base Error
M1	*Escherichia coli*	4,600,000	251	43	1.67%
M2	*Mycobacterium tuberculosis*	4,400,000	50-250	79	0.18%
M3	*Salmonella enterica*	4,900,000	35-250	89	0.60%
M4	*Salmonella enterica*	4,900,000	35-251	97	0.27%
M5	*Listeria monocytogenes*	3,000,000	35-251	171	1.60%
M6	*Pseudomonas syringae*	6,100,000	35-251	105	0.87%
M7	*Bifidobacterium dentium*	2,600,000	35-251	373	0.15%
M8	*Escherichia coli*	4,600,000	251	605	1.43%
M9	*Orientia tsutsugamushi*	2,100,000	301	1,460	1.92%

machines, and the rest are from HiSeq 2000 machines.

Table 3.3: Depth and breadth coverage for the HiSeq data sets.

Data Set	Read Depth %	20-mer Depth %	Read Breadth %	20-mer Breadth %
H1	80.15	93.80	30.36	98.54
H2	89.65	92.99	39.68	93.90
H3	82.01	90.76	26.46	98.26
H4	66.73	88.41	80.38	98.94
H5	47.83	71.53	50.59	84.08
H6	50.56	86.16	79.67	99.96
H7	68.13	90.98	18.08	96.38
H8	69.70	78.96	31.34	99.45
H9	46.23	77.33	19.28	93.19
H10	40.07	75.36	19.97	95.41
H11	47.51	94.23	19.79	98.86
H12	49.09	94.21	23.23	98.96
H13	78.90	95.08	35.05	98.54

The quality of the original data sets is evaluated with respect to each of the four measures, as given by *OrigReadDepth*, *OrigKmerDepth*, *OrigReadBreadth*, and *OrigKmerBreadth*. The quality of the original HiSeq data sets is listed in Table 3.3, and the quality of the original MiSeq data sets is listed in Table 3.4. A significant ratio of reads and *k*-mers contain errors, which makes correction an essential step to improving the usefulness of the data sets. HiSeq data sets have fewer errors, yet up to 60% of the reads can contain one or more positions with an error. This percentage can be as high as 99% for MiSeq data sets. The percentage of erroneous 20-mers is lower, but it can still reach 30% for HiSeq data sets, and 60% for MiSeq data sets. For M1, M5, M8 and M9, almost all of the reads are erroneous.

Table 3.4: Depth and breadth coverage for the MiSeq data sets.

Data Set	Read Depth %	20-mer Depth %	Read Breadth %	20-mer Breadth %
M1	1.46	50.57	0.25	100.00
M2	64.92	93.21	18.27	98.60
M3	23.21	72.79	5.78	83.16
M4	52.60	84.77	14.15	97.51
M5	2.35	41.01	0.87	50.16
M6	11.40	67.48	3.62	78.50
M7	69.35	94.36	58.28	99.99
M8	2.71	62.11	5.92	100.00
M9	0.29	42.90	1.01	74.95

The breadth of coverage depends on both the quality of the data set, and the depth of coverage. The *OrigReadBreadth* is not expected to be very high, but the *OrigKmerBreadth* is expected to be high. This is true for the HiSeq data sets, where most have an *OrigKmerBreadth* that is more than 90%, but MiSeq data sets have a wide range of values from 50% to 100%.

3.8 Results

We compared seven programs that have performed the best in recent studies; BLESS, Coral, HiTEC, Musket, RACER, SGA and SHREC. All tests were run on the Shared Hierarchical Academic Research Computing Network (SHARCNET), with a DELL PowerEdge R820 32 core Intel Xeon at 2.2 GHz with 1 TB of RAM, running Linux Red Hat CentOS 6.3. All programs have been tested in parallel, except BLESS and HiTEC, which do not have parallel modes. All programs were run with default parameters as indicated in their manuals, since this is typically what would happen in practice. It should be noted that not all data sets could be run by some of the programs for a variety of reasons.

Table 3.5: HiSeq *ReadDepthGain*

Data Set	BLESS	Coral	HiTEC	Musket	RACER	SGA	SHREC
H1	61.72	51.85	58.41	58.30	59.05	51.16	51.28
H2	44.30	41.16	43.37	42.77	42.72	40.14	42.64
H3	34.07	28.41	33.42	33.09	34.13	32.73	31.56
H4	91.64	55.66	92.01	86.56	92.27	86.94	88.90
H5	13.52	12.32	13.70	13.08	13.35	12.35	-
H6	86.16	4.37	84.46	79.08	82.84	55.60	65.96
H7	51.65	46.49	-	41.56	52.43	52.89	-
H8	25.14	24.21	-	23.68	26.77	25.97	-7.12
H9	18.14	10.96	-	16.79	17.91	17.03	14.80
H10	26.13	24.10	-	26.04	25.92	25.63	24.44
H11	-	-	-	53.28	57.49	61.66	-
H12	-	-	-	55.67	61.30	65.46	-
H13	-	-	-	21.25	26.53	27.86	-
AVG. H1-10	45.25	29.95	54.23	42.10	44.74	40.04	39.06
AVG. ALL	45.25	29.95	54.23	42.40	45.59	42.72	39.06

Table 3.6: HiSeq *KmerDepthGain*

Data Set	BLESS	Coral	HiTEC	Musket	RACER	SGA	SHREC
H1	51.14	34.08	51.00	51.28	51.11	34.20	41.84
H2	31.36	26.92	33.03	30.03	31.18	26.29	32.45
H3	17.86	14.45	18.22	18.55	19.26	15.96	16.79
H4	83.20	46.92	85.83	80.72	84.82	75.42	82.91
H5	8.79	7.13	9.27	8.41	8.74	7.14	-
H6	79.52	2.92	82.83	72.04	78.45	41.68	67.72
H7	42.21	36.79	-	38.92	47.35	42.46	-
H8	9.92	8.51	-	10.72	11.92	8.95	2.01
H9	14.43	7.81	-	15.40	16.19	12.90	14.24
H10	18.24	16.60	-	19.64	19.55	17.55	18.62
H11	-	-	-	36.73	38.11	31.79	-
H12	-	-	-	37.73	42.42	35.93	-
H13	-	-	-	31.38	35.49	28.26	-
AVG. H1-10	35.67	20.21	46.70	34.57	36.86	28.25	34.57
AVG. ALL	35.67	20.21	46.70	34.73	37.28	29.12	34.57

For readability purposes, all gain measures have been multiplied by 100 to represent the gain as a percentage. Also, a heat map has been used to easily determine the better performing programs; on each row the darker the green the better the performance. The error correction comparison on HiSeq data sets is presented in Table 3.5 for the *ReadDepthGain* and Table 3.6 for the *KmerDepthGain* measures. Table 3.7 shows the results for the *ReadBreadthGain*, and Table 3.8 shows the results for the *KmerBreadthGain* measures for the HiSeq data sets.

Table 3.7: HiSeq *ReadBredthGain*

Data Set	BLESS	Coral	HiTEC	Musket	RACER	SGA	SHREC
H1	5.66	4.42	5.00	4.99	5.04	4.36	4.37
H2	4.54	4.15	4.19	4.35	4.32	4.06	4.11
H3	2.28	1.88	2.16	2.13	2.23	2.13	2.03
H4	49.49	31.05	44.96	47.05	49.15	47.10	43.63
H5	4.63	4.09	4.35	4.37	4.43	4.08	-
H6	82.87	1.32	80.77	77.59	79.37	59.38	68.67
H7	4.69	4.10	-	3.63	4.62	4.63	-
H8	4.06	3.63	-	3.43	4.06	3.96	-2.96
H9	4.50	2.49	-	3.71	4.13	3.95	3.26
H10	7.98	7.12	-	7.60	7.82	7.77	7.16
H11	-	-	-	11.17	12.05	12.92	-
H12	-	-	-	13.76	15.10	16.15	-
H13	-	-	-	2.42	3.02	3.18	-
AVG. H1-10	17.07	6.43	23.57	15.89	16.52	14.14	16.28
AVG. ALL	17.07	6.43	23.57	14.32	15.03	13.36	16.28

Only Musket, RACER and SGA were able to run the human data sets. To provide a fair comparison of the HiSeq data sets, we show the average gain for all the data sets and for data sets H1 - H10. The results are very close for all four measures. For *ReadDepthGain*, BLESS is first for H1 - H10, and RACER is the best overall, with HiTEC, Musket, SGA, and SHREC close behind. SGA was first for the human data sets, RACER was second, and Musket was third.

For *KmerDepthGain* HiTEC was first, but RACER, BLESS, Musket and SHREC were all very close behind HiTEC. RACER was first for the human data sets, then Musket, and finally

Table 3.8: HiSeq *KmerBreadthGain*

Data Set	BLESS	Coral	HiTEC	Musket	RACER	SGA	SHREC
H1	-3.08	-1.74	-6.22	-4.02	-3.00	-2.09	-4.16
H2	-1.91	-1.92	-1.90	-1.85	-1.89	-1.73	-1.89
H3	-1.07	-5.09	-1.26	-0.93	-1.12	-0.88	-1.02
H4	-0.36	-1.65	-1.72	-1.72	-1.72	-1.73	-1.72
H5	-11.60	-9.98	-11.75	-11.36	-11.69	-9.83	-
H6	-2.20	0.00	-4.92	-10.63	-2.88	-0.21	-5.71
H7	-5.10	-2.68	-	-13.46	-7.69	-1.99	-
H8	-38.17	-10.96	-	-180.53	-86.13	-2.84	-285.54
H9	-30.27	-16.51	-	-33.87	-17.15	-5.70	-33.27
H10	-24.62	-39.15	-	-30.01	-19.22	-6.49	-33.34
H11	-	-	-	-6.30	-6.06	-4.89	-
H12	-	-	-	-7.15	-7.15	-5.63	-
H13	-	-	-	-5.00	-5.60	-4.32	-
AVG. H1-10	-11.84	-8.97	-4.63	-28.84	-15.25	-3.35	-45.83
AVG. ALL	-11.84	-8.97	-4.63	-23.60	-13.18	-3.72	-45.83

SGA. For *ReadBreathGain* BLESS was first, but RACER, HiTEC, SHREC, Musket, and SGA were not far behind BLESS. SGA was first for the human data sets, RACER was second, and Musket was third. For *KmerBreadthGain* SGA was first, followed by Coral, BLESS, RACER and HiTEC. For the human data sets SGA was first, Musket second, and RACER was third. As mentioned previously, all of the programs tested decreased the *k*-mer breadth of coverage. The programs that decreased the *k*-mer breadth of coverage the least were BLESS, RACER, and SGA.

The error correction comparison on MiSeq data is presented in Table 3.9 for the *ReadDepthGain* and Table 3.10 for the *KmerDepthGain* measures. Table 3.11 shows the results for the *ReadBreadthGain*, and Table 3.12 shows the results of the *KmerBreadthGain* measures for the MiSeq data sets.

BLESS and HiTEC could not run six of the nine data sets because they contain reads with different lengths. Musket did not do well on the MiSeq data sets, and the TP values of Musket were several orders of magnitude lower than those of the best programs, and they were equal to zero for four of the MiSeq data sets. This means that Musket basically did not correct the

Table 3.9: MiSeq *ReadDepthGain*

Data Set	BLESS	Coral	HiTEC	Musket	RACER	SGA	SHREC
M1	22.04	0.06	25.27	0.00	26.10	26.30	13.38
M2	-	30.65	-	-0.02	58.06	55.10	53.43
M3	-	10.23	-	0.06	13.09	11.56	-12.36
M4	-	61.32	-	0.70	74.96	66.16	-39.56
M5	-	0.42	-	0.06	0.88	0.64	-1.27
M6	-	13.03	-	0.00	16.46	14.60	-1.31
M7	-	77.71	-	0.00	86.11	67.74	-33.41
M8	33.61	0.01	37.78	0.00	20.19	22.72	19.95
M9	0.37	0.00	0.39	0.00	0.39	0.18	0.33
AVERAGE	18.67	21.49	21.15	0.09	32.92	29.44	-0.09

Table 3.10: MiSeq *KmerDepthGain*

Data Set	BLESS	Coral	HiTEC	Musket	RACER	SGA	SHREC
M1	21.22	0.04	36.29	0.00	37.42	9.50	23.81
M2	-	22.32	-	-0.16	49.48	41.22	42.80
M3	-	11.78	-	0.07	19.05	12.20	8.93
M4	-	34.36	-	0.46	58.72	33.33	22.66
M5	-	2.81	-	0.12	4.00	2.81	-1.72
M6	-	11.40	-	0.01	21.09	11.28	16.42
M7	-	48.40	-	0.00	65.59	38.27	59.46
M8	32.74	0.91	47.65	0.00	30.67	11.82	35.55
M9	4.28	0.61	6.71	0.00	6.19	1.61	5.74
AVERAGE	19.42	14.74	30.22	0.06	32.47	18.00	23.74

MiSeq data sets at all.

Table 3.11: MiSeq *ReadBreadthGain*

Data Set	BLESS	Coral	HiTEC	Musket	RACER	SGA	SHREC
M1	6.75	0.01	4.22	0.00	4.36	4.39	2.26
M2	-	3.38	-	-0.02	6.31	5.99	5.81
M3	-	1.50	-	0.00	1.94	1.70	1.70
M4	-	6.29	-	0.02	7.73	6.81	6.67
M5	-	0.07	-	0.01	0.10	0.09	0.09
M6	-	2.76	-	0.00	3.47	3.14	2.57
M7	-	25.48	-	0.00	28.41	22.60	26.72
M8	67.96	0.01	54.46	0.00	33.65	37.29	33.82
M9	0.97	0.00	0.47	0.00	0.46	0.28	0.42
AVERAGE	25.22	4.39	19.72	0.00	9.60	9.14	8.90

Table 3.12: MiSeq *KmerBreadthGain*

Data Set	BLESS	Coral	HiTEC	Musket	RACER	SGA	SHREC
M1	-187.97	0.00	-1206.02	0.00	-12.78	0.00	-110.53
M2	-	-1.57	-	0.01	-6.60	-3.98	-6.41
M3	-	-2.61	-	-0.01	-3.76	-2.91	-3.61
M4	-	-0.35	-	0.00	-0.60	-0.36	-0.91
M5	-	-7.49	-	-0.95	-12.43	-6.56	-11.71
M6	-	-3.06	-	0.00	-5.12	-3.33	-4.77
M7	-	-0.51	-	0.00	-6.15	-1.54	-6.67
M8	-6.67	0.00	5.00	0.00	-5.00	-0.83	-2.50
M9	-39.71	-1.70	-40.69	0.00	-42.11	-15.28	-37.07
AVERAGE	-78.12	-1.92	-413.90	-0.11	-10.51	-3.87	-20.46

For *ReadDepthGain* RACER was first, and was followed by SGA. BLESS and HiTEC could only correct three data sets, but they performed well on what they could run. For *KmerDepthGain* RACER was first, and no other program close to it. Although, HiTEC was first on two out of three data sets that it was able to run.

For *ReadBreadthGain* RACER was first, but SGA and SHREC were not far behind. However, BLESS was the best for all three data sets it could run, and HiTEC was second for two of the three data sets it could run. For *KmerBreadthGain* Coral was the best with SGA in second.

Our testing shows that there is no single program that is best at correcting all types of data sets. BLESS, Musket, RACER and SGA performed the best overall, with each having its own advantages and disadvantages for different data sets. Despite the amount of research in this area there is still significant room for improvement with respect to gain in both depth and breadth of coverage. In particular, all of the programs we tested actually decreased the breadth of coverage for the *k*-mers, which resulted in negative values for the *KmerBreadthGain*.

Table 3.13: Run time in seconds for all of the data sets tested.

Data Set	BLESS	Coral	HiTEC	Musket	RACER	SGA	SHREC
H1	2,716	453	1,711	100	47	187	570
H2	2,518	455	1,244	82	99	187	344
H3	1,971	400	2,391	95	70	249	416
H4	7,469	2,139	6,013	258	325	670	1,244
H5	9,517	3,446	7,981	251	350	732	-
H6	17,843	4,671	14,673	1,099	705	1,602	3,924
H7	39,179	5,654	-	923	949	2,343	-
H8	74,957	8,267	-	1,576	1,612	4,322	12,397
H9	77,361	11,320	-	2,891	2,284	4,888	12,811
H10	111,554	12,254	-	2,556	2,466	6,040	15,600
H11	-	-	-	39,777	109,169	148,642	-
H12	-	-	-	52,066	82,326	175,667	-
H13	-	-	-	46,054	85,068	177,643	-
M1	2,495	208	1,123	17	171	265	684
M2	-	605	-	63	61	202	257
M3	-	6,563	-	45	145	313	597
M4	-	986	-	45	175	373	714
M5	-	1,540	-	63	61	202	676
M6	-	5,407	-	56	227	521	982
M7	-	9,787	-	71	292	699	1,038
M8	47,437	5,692	17,831	218	1,268	3,539	14,375
M9	44,701	5,707	21,947	222	1,062	3,746	7,959

The run time for each of the data sets is listed in Table 3.13, and the peak memory usage for each program is listed in Table 3.14. As mentioned before, BLESS and HiTEC cannot run in parallel mode, so their time and memory reported is for running in serial mode.

For the data sets H1 - H10 RACER was the fastest, followed closely by Musket, and finally SGA. Musket was the fastest when considering all the data sets, and RACER was second.

Table 3.14: Memory usage in MB for all of the data sets tested.

Data Set	BLESS	Coral	HiTEC	Musket	RACER	SGA	SHREC
H1	13	71,712	5,987	3,077	3,094	2,863	1,002,668
H2	13	71,406	6,205	3,049	3,155	2,863	1,002,733
H3	26	73,152	9,533	3,057	3,317	2,888	1,002,668
H4	20	79,922	9,377	3,057	3,871	3,052	1,002,668
H5	13	80,763	9,080	3,054	3,662	3,119	-
H6	47	97,732	10,792	3,081	4,633	3,469	1,002,732
H7	163	105,409	-	3,122	8,791	3,780	-
H8	190	139,211	-	4,713	14,602	4,747	1,002,690
H9	213	145,579	-	4,723	17,778	4,937	1,002,795
H10	228	164,131	-	6,377	18,219	5,370	1,002,923
H11	-	-	-	60,915	231,930	49,107	-
H12	-	-	-	67,059	390,971	58,183	-
H13	-	-	-	60,915	237,682	58,857	-
M1	24	74,338	4,085	1,402	6,159	2,878	1,002,668
M2	-	71,906	-	3,115	2,761	2,854	1,002,668
M3	-	70,226	-	2,681	3,663	2,881	1,002,668
M4	-	75,552	-	2,681	3,610	2,915	1,002,668
M5	-	74,310	-	3,115	3,369	2,978	1,002,668
M6	-	77,771	-	3,121	3,912	2,872	1,002,668
M7	-	79,262	-	3,122	3,127	2,975	1,002,668
M8	177	136,933	11,694	1,585	7,112	3,816	1,002,732
M9	73	105,178	10,158	1,610	4,268	3,572	1,002,668

BLESS is by far the slowest because it only runs in serial mode, and it required a significant amount of time to read and write files in order to reduce memory usage. BLESS did not finish correcting the human data set H13 after running for 7 days. Although, the peak memory of BLESS is by far the lowest and it was two orders of magnitude lower than second best which was SGA. Coral, SHREC, and HiTEC cannot run large data sets due to memory usage.

For MiSeq data sets the space is usually not an issue since they are usually much smaller than HiSeq data sets. The order of performance for memory usage is similar to that for the HiSeq data sets with BLESS two orders of magnitude ahead, then Musket, SGA, and RACER. The fastest programs for the MiSeq data sets were Musket, RACER and then SGA.

3.9 Recommendations for biologists

This survey was a thorough comparison of the current state-of-the-art error correction programs, and it is clear that there is no single winner. For HiSeq data sets BLESS, Musket, RACER and SGA generally resulted in the greatest increase in the metrics. BLESS has the lowest memory requirements, but it has the longest running time and cannot run data sets with different read lengths. Musket did not finish on top very often, but it was not usually far from the best, and it runs fast with low memory usage. RACER often has the highest gain and lowest run time, but it uses a large amount of memory for human data sets. SGA usually has one of the top gain results and low space usage, but it is much slower than RACER and Musket. Using any of the four programs will provide good results, but only Musket, RACER and SGA can run on human data sets.

For MiSeq data sets RACER was the best in gain for depth and breadth of read coverage, and *k*-mer coverage depth. For *k*-mer coverage breadth SGA provides the smallest decrease. BLESS and HiTEC produced good results for the few data sets they could run.

Error correction can significantly increase the number of correct reads in a data set as shown in the corrected unmapped data sets in Table 2.2 and the corrected mapped data sets in Table 2.3. The overall improvement of the corrected data sets is very important, in spite of the slight decrease in *k*-mer coverage breadth. There is room for improvement in all aspects of coverage. The read coverage depth can be more than doubled for HiSeq data sets, and can be increased by over five times for MiSeq data sets. The *k*-mer coverage depth can be increased three times for HiSeq data sets, and six times for MiSeq data sets. The increase in coverage breadth depends both on the quality of the correction, and on the coverage depth, so it is more difficult to determine how much it can be improved.

It may be too difficult to design a program that improves the current state-of-the-art with respect to all four measures. If it is not possible then future programs may need to become more specialized, and target improvements for only some aspects of the original data sets.

An important problem that requires further investigation is that of the biological significance of the correction of data sets. There is important information in uncorrected data sets that may be removed by correction. Also, we have shown that all programs reduce the *k*-mer breadth of coverage, and further investigation is necessary to determine if the information lost was important and can be prevented.

3.10 Conclusions

We have provided the scientific community with a comprehensive assessment of the current state-of-the-art error correction software available for the Illumina technologies. We have also introduced a methodology of standardizing the assessment of the quality of the corrections that is thorough and unbiased. We have determined that there is no single program that performed best overall. This information is extremely important for any researcher that uses DNA sequencing technologies.

Chapter 4

Genome Assembly: SAGE2

4.1 Introduction

The information outlined in this chapter is the continuation of our work on a previously published *de novo* genome assembly program called SAGE [14]. SAGE uses the OLC method for building an overlap graph using overlaps in reads from NGS machines. SAGE uses information about the flow through the overlap graph to estimate the number of times each edge in the graph should be used. This flow information, along with the paired-end information from the reads, is used to extend the contiguity of the edges in the overlap graph. Our results from [14] showed that SAGE was an improvement over the state-of-the-art *de novo* genome assemblers for small and medium sized genomes. SAGE had not been tested on large genomes, and many changes were necessary for it to assemble large genomes accurately and efficiently. We have reimplemented and improved SAGE so that it can assemble large genomes as good or better than the current state-of-the-art; the new program is called SAGE2.

In this chapter I first outline the goals for SAGE2, and provide an overview of the SAGE2 algorithm. Next, I explain the methods used to store the reads, and build the hash table of the prefixes and suffixes of the reads. I then outline the new algorithm for building the overlap graph in parallel, and the rest of the overlap graph construction and cleaning methods. Next, I describe the new implementation of the edge merging process used in SAGE2, and the new implementation of the scaffold building process. Finally, I show the results of SAGE2 compared to leading *de novo* genome assembly programs.

4.2 Goals for SAGE2

There were three main problems with SAGE that needed to be addressed in SAGE2, and this required a complete reimplementation of the program. The first problem was that SAGE can only run in serial, and to be able to assemble large genomes with high coverage would require parallelization, so we have developed a new algorithm and implementation of the overlap graph construction that works in parallel, and we have parallelized most of the pipeline in SAGE2. The second problem was that the assembly process in SAGE was very aggressive, which resulted in longer contigs and scaffolds, but also caused many mis-assemblies. SAGE2 required a new algorithm for assembling genomes with less mis-assemblies, while maintain the contiguity of the contigs and scaffolds. The final problem with SAGE was that it can only assemble data sets that have reads with the same length. SAGE2 has been implemented so that it can assemble reads with any length and from multiple data sets, assuming all the data sets are from the same sampled genome.

The bottleneck in run time for SAGE is the building of the overlap graph. The new parallel overlap graph construction used in SAGE2 significantly reduces the run time for large genomes with high coverage. To reduce the run time even more we have reimplemented many of the functions in SAGE2 so that they can run in parallel, most notably the merging of contigs and scaffolds. The parallel implementation used in SAGE2 runs nearly as fast as the current state-of-the-art *de novo* genome assemblers for all genomes sizes and coverage levels.

The algorithm of SAGE resulted in many mis-assemblies for all of the genomes that were tested in [14]. Large genomes, such as the human genome, are more complex and have many more repeat regions, which makes them much more difficult to assemble. To assemble large genomes accurately would require new algorithmic approaches to reduce mis-assemblies and increase the contiguity of the assemblies. The new algorithms introduced in SAGE2 have significantly reduced mis-assemblies, while also significantly increasing the contiguity of the assemblies compared to SAGE. We have performed comprehensive testing that shows these improvements have made SAGE2 perform as good or better than the current state-of-the-art for all genomes sizes and coverage levels.

4.3 SAGE2 algorithm

An overview of the SAGE2 algorithm is listed in Algorithm 4. A general description of the algorithm will be given here, and the following sections will give detailed explanations of the algorithm. Some of the differences in the algorithms of SAGE and SAGE2 will be explained in the respective sections.

Algorithm 4 Overview of the SAGE2 algorithm.

```
Read the error corrected data set.
Build the hash table of read prefixes and suffixes.
Build the overlap graph in parallel.
Build the remaining overlap graph in serial.
Simplify the overlap graph.
Estimate the genome size.
Compute the minimum cost flow through the overlap graph.
Map paired-end reads to the edges in the overlap graph.
Estimate the mean and standard deviation of paired-end insert size.
Merge contigs using flow and paired-end read support.
Merge contigs into scaffolds using paired-end read support.
```

Before inputting a data set for *de novo* genome assembly SAGE2 requires that the data set have the errors corrected by an external program. SAGE2 can assemble an uncorrected data set, but the results will be significantly better if the data set is corrected first. We suggest that RACER be used to correct the data set, but any error correction program can be used.

SAGE2 begins by reading the data set and storing all of the unique reads from the data set. To build the overlap graph from the reads requires a search of all the reads for overlaps. If there are n unique reads in the data set then it would require in the worst case $O(n^2)$ comparisons. For large data sets this is an unreasonable amount of time to spend searching for overlaps. To reduce the search time for finding overlaps in the reads SAGE2 uses a hash table to store the prefixes and suffixes of the unique reads. The hash table is used to quickly find reads that overlap each other by a minimum amount of bases in order to build the overlap graph.

The parallel overlap graph construction finds all reads that have no ambiguous extensions, which we call "contained" reads. This parallel overlap graph procedure typically finds 70% to 80% of the reads in a data set to be contained reads. The remaining reads are inserted into the overlap graph using the serial implementation from SAGE that reduces transitive edges while the graph is being built.

Once the overlap graph has been built it must be simplified to reduce its size and remove erroneous edges. There are three components to the simplification process; contracting composite paths, removing dead-ends, and removing bubbles. These three procedures are performed repeatedly until there are no more simplifications made.

Once the graph has been simplified we can estimate the size of the genome based on the length of the edges in the overlap graph, and the number of reads on each of the edges. The estimation of the genome size is needed for the next step in the SAGE2 algorithm, which is finding the minimum cost flow through the overlap graph. The flow through the edges of the overlap graph gives us a highly accurate estimation of the number of times each edge should appear in the sequenced genome. This information is used in the merging process, along with the paired-end support, to accurately merge edges in the overlap graph.

In order to merge edges in the overlap graph we need the flow and the paired-end support. To find the paired-end support we first need to map each of the paired-end reads to the edges in the overlap graph. Once this is finished we can then use paired reads that map uniquely to the same edge to get an estimate of the insert size, mean, and standard deviation of the paired-end reads for the data set. This improves the accuracy of the merging process by verifying the distance between paired-end reads for support, and helps set limits on the distance searched along edges for paired-end support.

Multiple rounds of merging are performed to build the final contigs from the overlap graph. After the final contigs have been built, the paired-end support is used to merge contigs into scaffolds. The estimated insert size and standard deviation of the paired-end reads are used to assist the merging process and to fill the gaps with the proper length between contigs. The following sections will give a detailed explanation of the each of these steps in the SAGE2 algorithm.

4.4 Error correction

Uncorrected data sets usually contain many errors, and these errors cause problems when building an overlap graph. Genome assemblers that use the OLC method usually require that the reads are corrected before assembly because they use the entire read for building and modifying the graph. The more errors there are in the reads, the less likely there will be overlaps between the reads, which means there will be fewer edges that are inserted into the overlap graph. SAGE2 expects that the reads have been corrected before the assembly process, but it will still work even if the reads have not been corrected.

The DBG genome assemblers usually input the uncorrected data sets because they only require the length of the k-mer to match, and any errors before or after the k-mer can be corrected using the topology of the DBG graph. This is a problem if the reads have many errors spread evenly throughout the reads, but the Illumina technology mainly introduces errors at the ends of the reads or at positions that can be idiosyncratic to the particular run.

We use RACER to correct the reads before assembly because it is fast, memory efficient, and it is one of the top performing error correction programs available [27]. SAGE2 will work with reads corrected by any error correction software, and even uncorrected data sets will still work, but the assembly will not be as good as using a corrected data set.

4.5 Inputing the reads

SAGE2 accepts reads that are either in FASTA or FASTQ format. The reads can either be in one file with the paired-end reads interlaced, or a text file can be provided with the paired-end read files listed one after the other. SAGE2 also accepts interlaced files to be incorporated in the list of files. If the reads are in FASTQ format, the quality values are ignored since they are not used in the SAGE2 algorithm. Reads that are shorter than the minimum overlap length, and reads that contain any bases that are not A, C, G, or T are not used in SAGE2.

To save memory the reads are stored in 8-bit arrays that encode the bases in 2-bits per base. This reduces the amount of space required to store a read by as much as four times compared to storing the reads in a character array. The deoxyribonucleotide A is stored as 00, and its complement deoxyribonucleotide T is stored as 11. The deoxyribonucleotide G is stored as 10, and its complement deoxyribonucleotide C is stored as 01. This encoding allows SAGE2 to quickly find the reverse complement of a sequence by flipping the bits of each deoxyribonucleotide in the reverse order of the forward sequence. This is much faster than reading each character and then replacing it with the appropriate complement deoxyribonucleotide using a character array.

Since a read and its reverse complement are considered the same read, only the lexicographically smaller of the two is kept initially in order to organize the reads properly. After all the reads have been processed, they are then sorted lexicographically so that we can remove duplicate reads. SAGE2 only stores one copy of each unique read in a data set, and a counter is used to keep track of the number of times each unique read is present in the data set, which is needed in the downstream functions. After the reads have been stored and sorted the next step is to build a hash table of the read prefixes and suffixes.

4.6 Building the hash table

To build the overlap graph we must find all of the reads that overlap with each other by a minimum overlap length. As mentioned previously, comparing all reads to each other for overlaps is impractical, so to reduce the number of reads searched we use a hash table to quickly find reads that overlap with each other.

The hash table stores the prefixes and suffixes of each read for both the forward and reverse orientations of each read. The length of the prefix and suffix used is the minimum overlap length, or 64 bases if the minimum overlap length is greater than 64. The limit is set to 64 because we store the prefixes and suffixes in two 64-bit words to reduce space and search time. Using a longer hash element for the hash table would increase the complexity of the hash function, increase space usage, and increase the time needed to search the hash table. Also, reads that overlap by at least 64 base pairs is specific enough to find all of the reads that overlap quickly and efficiently.

Large genomes typically have many repetitive regions that are less than or equal to our maximum hash element length of 64 bases. These repetitive regions can significantly increase the search time needed to find overlaps between reads, and they typically cause the assemblies to be less optimal. To minimize the impact of highly repetitive hash elements we have set a limit on the number of times a hash element is found. If a hash element is found more than the set threshold then it is marked and not used during the overlap graph construction. The default threshold to mark a hash element in SAGE2 has been set to 100. This threshold was experimentally determined to increases run time, reduce mis-assemblies, and increase the contiguity of the assemblies.

4.7 Parallel overlap graph construction

Building the overlap graph in serial is a bottleneck in the run time to assemble larger genomes with high coverage in SAGE. We have resolved this issue by implementing a new algorithm for building an overlap graph in parallel for SAGE2. The information in this section outlines the implementation for parallel overlap graph construction used in SAGE2.

The parallel overlap graph in SAGE2 is built using OpenMP, and the set of reads is divided into equal sized chunks for each thread. Each thread searches the reads in their chunk for unambiguous extensions on both sides of the read. A read has unambiguous extensions if at any position in the read there is no more than one left overlapping read or one right overlapping read that overlaps the current read at that position. SAGE2 also requires that each of the reads that overlap in previous positions also overlaps with any reads found at subsequent positions. If there are no ambiguous extensions at any position along a read then it is considered to be a contained read.

Figure 4.1 shows a simplified example of a contained read. In this example the current read is being searched for overlapping reads. The first left overlapping read that would be found that overlaps the current read would be r_1, and the first right overlapping read that would be found to overlap the current read is r_3. The overlapping regions with the current read are highlighted in blue.

```
Read $r_1$        : AGCTAAGCAACGATAGCCGATAGCTAAATTAC
Read $r_2$        :    TAAGCAACGATAGCCGATAGCTAAATTACGTT
Current Read   :       GCAACGATAGCCGATAGCTAAATTACGTTATA
Read $r_3$        :        CAACGATAGCCGATAGCTAAATTACGTTATACTC
Read $r_4$        :          ACGATAGCCGATAGCTAAATTACGTTATACTCATC
```

Figure 4.1: The current read is searched for extensions by reads that overlap both the left and right side of the current read. A hash table with the prefixes and suffixes of each read is used to find overlapping reads. If the extending reads overlap the current read (blue), and the extending reads also overlap with each other (green and red), then the current read is considered to be a contained read. Contained reads are added to the overlap graph with edges connected to the extending reads with the longest overlap to the left and right of the current read (r_1 and r_3).

The next left overlapping read that would be found is r_2, and SAGE2 requires that it also overlaps with the previously found left overlapping read r_1, which is highlighted in green. The next right overlapping read that would be found is r_4, and SAGE2 requires that it also overlaps with the previously found right overlapping read r_3, which is highlighted in red. This process is continued until all of the overlapping reads at each position in the current read have been found. The algorithm for building the overlap graph in parallel to find the contained reads is listed in Algorithm 5.

SAGE2 uses two arrays to store the information about the overlapping reads for each of the unique reads in the data set. One array stores the information for the left overlapping read with the largest overlap for each unique read, and the other array stores the information for the right overlapping read with the largest overlap for each unique read. For data sets with varying read lengths there can be more than one read that overlaps at each position in the read being searching. SAGE2 only stores the longest read at any particular position, as long as it also overlaps with the shorter reads found at that position. In Figure 4.1 read r_2 would be the final read stored in the left overlapping array for the current read, and read r_3 would be the final read stored in the right overlapping array for the current read.

SAGE2 searches each read starting at the first position in the read, and finds all the reads that overlap at that position using the hash table. If there is only one read overlapping the left or right then we save it in the respective tables. We then move to the next position in the read and check that there is only one read that overlaps in either direction, and that read must also overlap with the previous read that was found, as shown in the green and red highlighted areas in Figure 4.1. This process is continued until we have searched for the overlapping reads at each position of the current read.

Highly connected reads cause a great deal of branching in the overlap graph and it is difficult to resolve these highly branching nodes. Removing these reads from the graph breaks the branching, which allows for the edges that are connected to these highly connected reads to be resolved easier, with less chance of making mis-assemblies. If a read has more than 300 reads that overlap it, then we mark the read as *highly connected* and it is not used in the overlap graph. This threshold has been experimentally determined.

After searching the reads for overlaps the two arrays will contain the incoming and outgoing reads with the longest overlaps for each read. If the read had any ambiguous extensions then

Algorithm 5 Parallel overlap graph construction.

```
Input: Set of unique reads $R$, hash table $H$ of read prefixes and suffixes.
Output: Overlap graph $G$.
$L \leftarrow 0$    ▷ Initialize array to store left overlapping read with maximum overlap.
$R \leftarrow 0$    ▷ Initialize array to store right overlapping read with maximum overlap.
for each read $r \in R$ do    ▷ Parallel for loop using OpenMP.
    $totalOverlappingReads \leftarrow 0$    ▷ Variable to count the number of overlapping reads.
    for each position $i \in r$ do
        for each read $s \in H$ that overlaps $r$ starting at position $i$ do
            if $s$ is left overlapping then
                if $s$ is the first left overlapping read found then
                    $L \leftarrow s$
                    $totalOverlappingReads + +$
                else
                    if $s$ overlaps with the previous left overlapping read $L_s$ then
                        $L \leftarrow s$
                        $totalOverlappingReads + +$
                    else
                        Mark $r$ as ambiguous
            if $s$ is right overlapping then
                if $s$ is the first right overlapping read found then
                    $R \leftarrow s$
                    $totalOverlappingReads + +$
                else
                    if $s$ overlaps with the previous right overlapping read $R_s$ then
                        $R \leftarrow s$
                        $totalOverlappingReads + +$
                    else
                        Mark $r$ as ambiguous.
    if $totalOverlappingReads \geq 300$ then
        Mark $r$ as highly connected.
for each read $r \in R$ do
    if $r$ is not marked as highly connected or ambiguous then
        if $r$ has reciprocal maximum extensions then
            $G \leftarrow edge(L_s, r)$    ▷ Add longest left overlapping read to the overlap graph.
            $G \leftarrow edge(r, R_s)$    ▷ Add longest right overlapping read to the overlap graph.
            Mark $r$ as contained.
return $G$
```

it is marked so that it is not considered for the following step of the parallel overlap graph construction.

Next SAGE2 checks each read to see if it has *reciprocal maximum extensions* for both the incoming and outgoing reads. A read has a reciprocal maximum extension if the read that extends the current read also has the current read as its maximum extension in the respective array location. In Figure 4.1 we would require that r_2 and r_3 have the current read in their respective arrays locations for it to be a reciprocal maximum extension. If a read has reciprocal maximum extensions for both the incoming and outgoing reads then we add the respective edges in the overlap graph, and then mark the current read as contained so that it is not used in the serial overlap graph construction step. The remaining reads that do not have reciprocal maximum extensions, or have ambiguous extensions, are added to the overlap graph using the serial procedure from SAGE.

Experimental results show that typically between 70% to 80% of the reads in a data set are contained reads for Illumina data sets. The remaining reads that are not contained are searched for overlaps in serial mode using the same procedure that is implemented in SAGE. The parallel construction of the overlap graph has the biggest impact on large genomes with high coverage. Table 4.1 shows the time need to build the overlap graph for both SAGE and SAGE2 on three human genome data sets with varying coverage levels. H1 has a coverage of 41x, H2 has a coverage of 50x, and H3 has a coverage of 54x. This table shows that the parallel overlap construction can significantly reduce the run time for building the overlap graph for large genomes with high coverage. This is important because the newest sequencing technologies from Illumina have much higher coverage levels and longer read lengths.

4.8 Serial overlap graph construction

After adding all of the contained reads to the overlap graph the remaining reads that were not marked as contained, or highly connected, are used in the serial overlap graph construction

Table 4.1: Comparison of overlap graph construction time in hours between SAGE and SAGE2 for three human data sets.

Data	SAGE	SAGE2			
Set	Serial	Parallel	Serial	Total	Difference
H1	12.6	5.2	3.5	8.7	3.9
H2	29.6	5.8	3.3	9.1	20.5
H3	33.4	7.7	3.6	11.3	22.1

with transitive edge reduction. This algorithm has not been modified from SAGE so only the important points of the algorithm will be mentioned here. The reader is referred to [9, 14] for a more detailed explanation of the serial overlap graph construction algorithm used in SAGE and SAGE2.

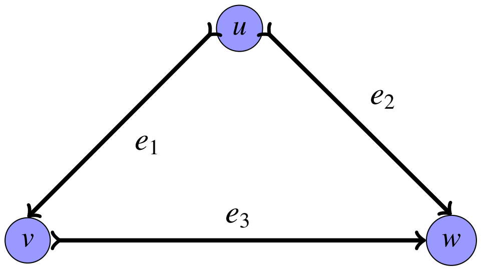


Figure 4.2: An example of an overlap graph before transitive edge reduction. The string spelled by the edge e_2 and the string spelled by the edges e_1 and e_3 is the same. We call the edge e_2 a transitive edge since it can be removed without losing information about the string.

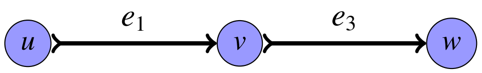


Figure 4.3: An example of the overlap graph in Figure 4.2 after transitive edge reduction. The edge e_2 has been removed from Figure 4.2 resulting in an overlap graph without a transitive edge.

Assume there are three overlapping reads u, v, and w with edges $e_1 = (u, v)$, $e_2 = (u, w)$, and $e_3 = (v, w)$. Assume the string spelled by the edge e_2 and the string spelled by the edges e_1 and e_3 is the same. We call the edge e_2 a transitive edge because it is redundant, and it can be removed from the graph without losing information about the string. Figure 4.2 shows this graph before transitive edge reduction, and Figure 4.3 shows the graph after transitive edge reduction. Removing transitive edges from the overlap graph is important because it significantly reduces the amount of space used to store the overlap graph.

SAGE builds the overlap graph using a modified version of an algorithm proposed by Eugene Myers [28]. The problem with the algorithm proposed by Myers is that it requires the entire overlap graph to be built before removing the transitive edges. For large genomes this is impractical since it would require a large amount of memory to build the entire overlap graph. The algorithm in SAGE significantly reduces the memory used to build the overlap graph by removing the transitive edges for a node before adding the edges for another node. Once the serial overlap graph construction step has finished then the complete overlap graph has been built. The next step in the SAGE2 algorithm is to simplify the overlap graph to reduce its size and remove erroneous edges that were inserted during the overlap graph construction.

4.9 Graph simplification

After the overlap graph has been built it must be simplified to reduce its size and remove erroneous edges. SAGE2 goes through multiple iterations of the simplification steps to simplify the overlap graph. In each iteration of simplification we *contract composite paths* first, then we remove *dead-ends*, and finally we *pop bubbles* in the overlap graph. The details of each procedure will be explained in the following subsections.

The graph simplification process has significantly changed from SAGE to SAGE2. Some of the simplification steps used in SAGE have been eliminated from SAGE2 because it was found that they caused many mis-assemblies, while only extending the contiguity of the edges in the overlap graph by a small amount or not at all. Two of the three simplification functions that were kept from SAGE have been modified to provide better assemblies in SAGE2.

4.9.1 Contracting composite paths

Many of the nodes in the transitively reduced overlap graph only have one incoming edge and one outgoing edge. Since there is no ambiguity in the path through these types of nodes we can condense it without losing any information about the genome. We call this type of path a composite path because we are taking two edges in the overlap graph and combining them into one edge. The information about the node along the path is stored on the new edge.

For each node v that only has one incoming edge $e_1 = (u, v)$, and one outgoing edge $e_2 = (v, w)$, we remove the node v along with edges e_1 and e_2, and insert an edge $e_3 = (u, w)$ in the overlap graph. The edge e_3 stores the information about node v and the edges e_1 and e_2.

An example of an overlap graph before contracting composite paths is shown in Figure 4.4. The nodes r_2, r_4, and r_6 all have one incoming edge and one outgoing edge, so this function will contract them and put their information in the appropriate edges. The resulting graph after contracting composite paths is shown in Figure 4.5.

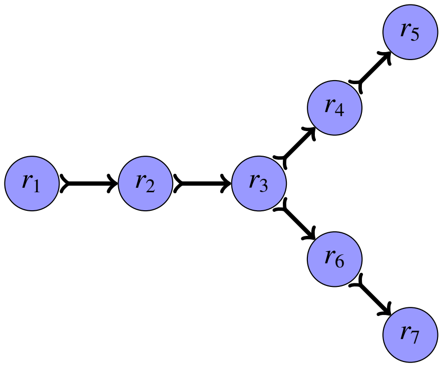


Figure 4.4: An example of an overlap graph before contracting composite paths. Nodes with only one incoming edge and one outgoing edge are considered to have a composite path. The edges are combined into one edge and the information in the node is added to the new edge.

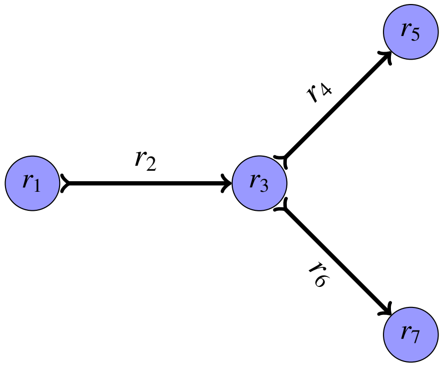


Figure 4.5: The overlap graph from Figure 4.4 after contracting composite paths. The resulting graph only contains nodes with more than two edges connected to them, or only one edge connected to them.

4.9.2 Removing dead-ends

Even though we correct the reads before building the overlap graph there are still reads that contain errors. These erroneous reads may still overlap with another read but their nodes will likely only have a single edge connected to them. This creates a dead-end in the overlap graph, as shown by r_4 in Figure 4.6, which can be removed from the graph to reduce the branching from erroneous connections. Some dead-ends in the graph may not be errors, but instead are low coverage regions that extend for more than one read. We remove these from the graph because they maintain splits in the graph that can prevent merging of many of the edges with higher coverage.

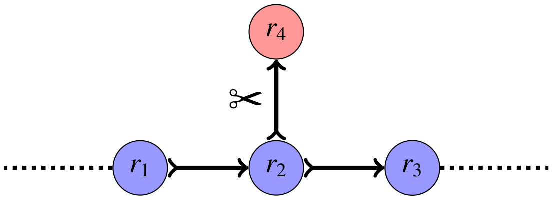


Figure 4.6: A example of a dead end in the overlap graph caused by an erroneous read. Reads with only one edge connected to them are considered dead ends and removed from the overlap graph.

Dead-ends with many reads on the edge are likely not errors, but instead are regions of the sampled genome that do not connect to another portion of the graph. This could either be a region of the genome that has low coverage or the end of a chromosome. We use a threshold for the number of reads on an edge that can be deleted to distinguish these types of dead-ends from those with errors.

The threshold for the number of reads on an edge used in SAGE is 5 for all iterations of the dead-end removal function. After extensive testing of SAGE2 on multiple genomes we found that we can reduce errors in the assembly by starting with a low threshold and then increasing it after each iteration. SAGE2 starts with a threshold of 0 reads on an edge, which is most likely an error, and we increase the threshold by one after every iteration of simplification, until the maximum threshold of 4 is reached.

4.9.3 Popping bubbles

Bubbles in an overlap graph can be created either by erroneous overlaps in the graph, or by heterozygous regions in a diploid genome. Erroneous overlaps will create a bubble with a path that has many reads along one path, and very few reads along the other path. In both SAGE and SAGE2, if the two paths have very similar strings and one of the paths has less than half the reads as the other path, then we remove the path with less reads. An example of bubble popping is shown in Figures 4.7 and 4.8.

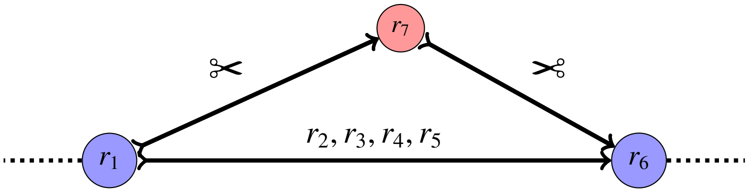


Figure 4.7: An example of a bubble in the overlap graph. There are two paths from r_1 to r_6 that form the bubble. The path with the least amount of reads is removed from the graph.

SAGE only pops bubbles that have lengths on the edges that are different by ≤ 50 bp. After extensive testing of SAGE2 on multiple genomes it was determined that popping bubbles with more similar lengths in the first iteration, and then less similar lengths in subsequent iterations, reduces the number of mis-assemblies. SAGE2 starts with a threshold of similarity between

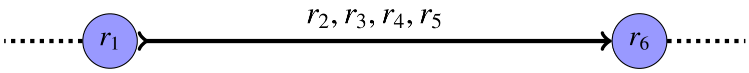


Figure 4.8: The overlap graph in Figure 4.7 after removing the bubble. The node r_3 is removed if there are no more edges connected to it. If there are still edges connect to the nodes in the path that was removed then they remain in the overlap graph.

the length of the edges that is ≤ 10 bp, and increases the threshold by 10 bp after each iteration, until the maximum threshold of 50 bp is reached.

4.10 Genome size estimation

In *de novo* genome assembly the size of the sampled genome is unknown, but the length of the genome is needed to find the assembly that most likely represents the sampled genome. In previous testing of SAGE the genome size estimation was always very close to that of the sampled genome, so we have not changed the algorithm used for genome size estimation in SAGE2, and the reader is referred to [9, 14] for a detailed explanation of the algorithm and it is briefly explained below.

SAGE and SAGE2 use an algorithm to estimate the size of the sampled genome that was proposed by Eugene Myers [28]. Assume that L is the length of the sampled genome and there are n reads in the data set, and the average length of the reads is l. If we assume the reads were sampled randomly and uniformly, then each position of the genome should have been sampled on average $\frac{nl}{L}$ times. Similarly, we can take an edge from the overlap graph with length m that has k reads on the edge, and assuming it is a unique sequence in the genome, then each position in the sequence will have been sampled on average $\frac{kl}{m}$ times. If m is long enough then it is most likely a unique sequence in the genome, and the two estimations will be very similar.

For the first estimation of the length of the sampled genome we find the sum of the lengths of all edges L in the overlap graph that are longer than 500 bp. We also count the total number of reads R that appeared on those edges. Assuming the edges over 500 bp appear only once in the genome, and the total number of reads in the data set is N, we can estimate the length of the sampled genome to be approximately $\frac{N}{RL}$.

This estimation will be incorrect if some of the edges considered in the estimation appear multiple times in the genome. To find a more accurate estimate of the genome length we only want to include long edges that are likely to be unique in the genome. To determine if a sequence is unique in the sampled genome we compute the ratio of the probability that the sequence represented by an edge in the graph appears only once in the genome, and the probability that it appears twice in the genome.

We iteratively estimate the genome size until the previous estimate is the same as the current estimate, or until we have tried 10 iterations of estimation. Our testing shows that SAGE2 typically only requires 3 or 4 iterations until the prior and posterior probabilities converge.

4.11 Estimating insert size distribution

In order to properly use the paired-end reads in the following steps we need to have an approximation of the distribution of the insert size for the data set. If we know the mean μ and the standard deviation σ of the distribution, then we can set limits for how far we search in the overlap graph to find paths between paired-end reads. The calculations and thresholds for finding the insert size are the same in both SAGE and SAGE2, and the reader is referred to [9, 14] for a detailed explanation of the algorithm and the approach is outlined below.

We find the initial estimates of μ and σ by finding paired-end reads for which both of the reads map uniquely to the same edge in the overlap graph. For each of those paired-end reads we find the distance between them on that edge, and add it to the set of all of the distances we find. We use this set of distances to calculate the first approximation for the values of μ and σ.

The initial calculation of μ and σ can be skewed a little by any outliers, and we would like to have an estimation that is closer to the majority of the insert sizes. To account for outliers we recalculate the values of μ and σ again, but we only consider insert sizes that are $\leq 4\mu$. We repeat this process for a maximum of 10 iterations, or until the value of μ is $> 1\%$ different than the previous estimation. It typically takes 3 iterations of the above procedure for the prior and posterior estimates to converge.

The minimum and maximum lengths we set for searching for paired-end reads on a edge are $\mu - 3\sigma$ and $\mu + 3\sigma$ respectively. These thresholds are used because the distribution of insert sizes will most likely follow a normal curve, and more than 99% of the insert sizes will fall within this range.

4.12 Minimum cost flow

To assemble a genome from a set of edges in the overlap graph that best represents the actual genome sampled it is important to know the number of times each edge appears in the genome. The number of times that an edge in the overlap graph appears in the sampled genome is called its copy count. It was shown in [9, 14] that SAGE performs very well at estimating the copy counts, so we use the same approach in SAGE2, and the reader is referred to [9, 14] for a detailed explanation of the algorithm that is explained briefly below.

To obtain accurate estimates of the copy counts of the edges in the overlap graph we convert the overlap graph into a flow network, and then we solve the minimum cost flow problem on the flow network. We use an external program called CS2 [7] for solving the flow problem. We use CS2 because it has been shown to be one of the top performing programs for solving minimum flow cost problems [7]. The total amount of flow through an edge in the overlap graph approximates the number of times that the reads on that edge appear in the sampled genome. We estimate copy counts of the edges by setting a cost in the edges that approximates the number of times the reads are present. Our initial estimates are based on the number of times each read appears in the data set.

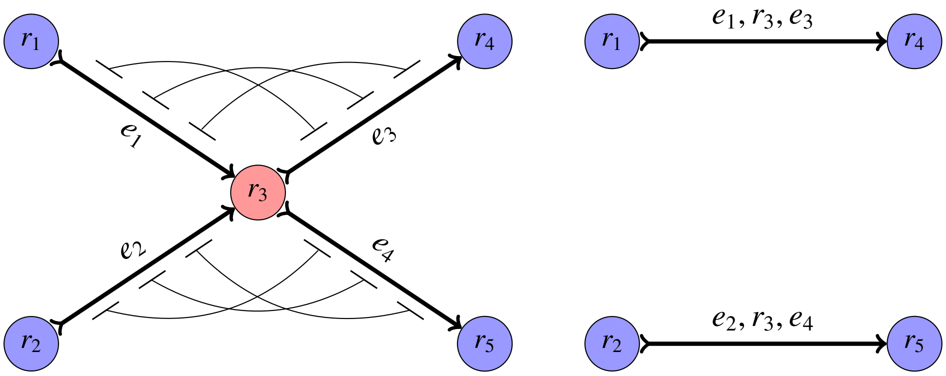
CS2 returns an estimate of the number of times each edge should be represented in the actual genome. We use these estimates when merging edges in the overlap graph to increase the accuracy and contiguity of the edges in the overlap graph.

4.13 Resolving ambiguous nodes

4.13.1 Introduction

After simplifying the overlap graph there are still many ambiguous nodes, where the edges could be merged in more than one way. To accurately resolve an ambiguous node we use paired-end reads that span across the ambiguous node as support for which way to merge the edges.

Consider the ambiguous node r_3 in Figure 4.9a, for which we will assume has a flow of 1 on all four edges connected to r_3. If we only consider the flow in the edges then we will not be able to find the correct merging through r_3, since all of the edges have the same amount of flow. If there are enough paired-end reads that uniquely support a pair of edges, such as e_1 with e_3, then we merge those edges. The same applies for the edges e_2 with e_4, assuming they also have enough paired-end read support for merging.

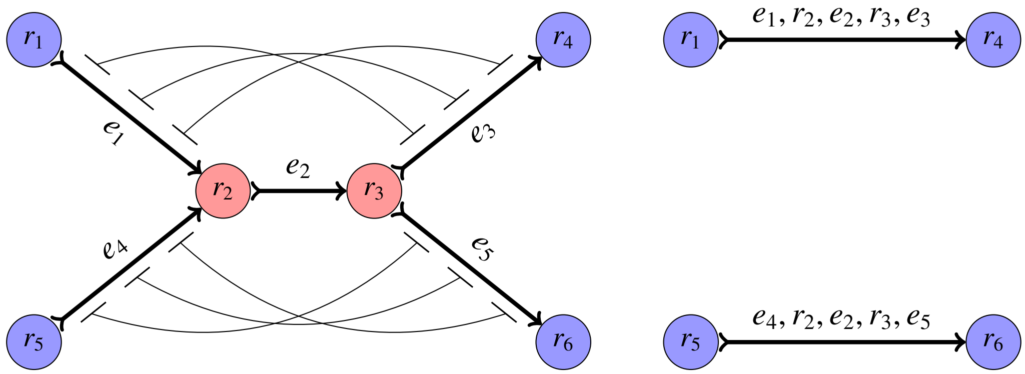


(a) Ambiguous node r_3. (b) Node r_3 resolved..

Figure 4.9: Ambiguous node r_3 in Figure 4.9a resolved using paired-end information, assuming the flow on each of the edges is 1. There are paired-end reads supporting the merge between edges e_1 and e_3, and between edges e_2 and e_4. The resulting graph after merging the edges is shown in Figure 4.9b

Assume that e_1 and e_3 are uniquely supported by paired-end reads above the threshold for merging, and assume that e_2 and e_4 are uniquely supported by paired-end reads above the threshold for merging. We can then merge the edge e_1 with e_3 and e_2 with e_4. The resulting graph after merging the edges is shown in Figure 4.9b, with the ambiguous node r_3 now located on both of the merged edges.

This procedure only works when the paired-end reads span a single ambiguous node, but many paired-end reads will span multiple nodes when the edges are short and have very few reads on them. These types of ambiguous nodes require a path search through the overlap graph to find the proper edges to merge.



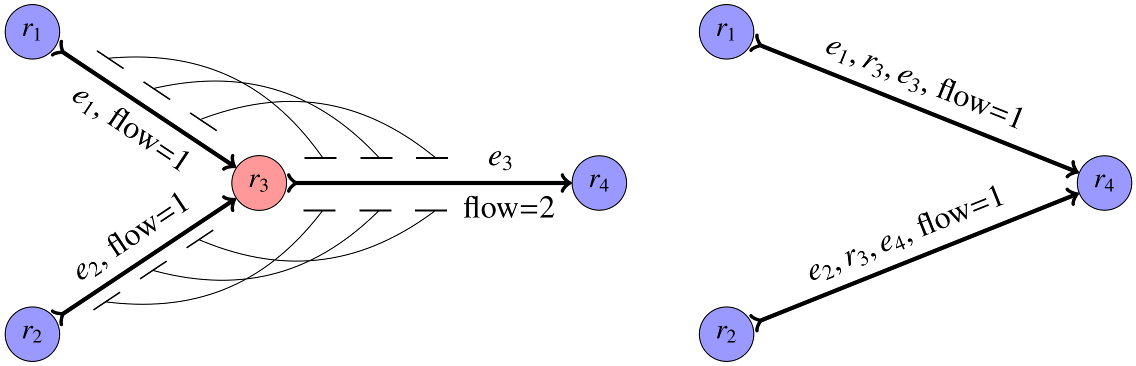
(a) Paired-end support using path search.

(b) Edges merged using paired-end support.

Figure 4.10: Edges merged using paired-end support, assuming the flow on edge e_2 is > 1. There are paired-end reads supporting a merge between edges e_1 and e_3, and between edges e_4 and e_5. The resulting graph is shown in Figure 4.10b

Consider the example shown in Figure 4.10. Nodes r_2 and r_3 are ambiguous nodes, and assume that the short edge e_2 between them has no support to merge with any other edges. For this example we will also assume that the flow on edge e_2 is > 1. Using a path search through the overlap graph shows that there is paired-end support to merge edge e_1 with e_3 and edges e_4 with e_6 as shown in Figure 4.10a. The resulting graph from resolving the ambiguous nodes r_2 and r_3 is shown in Figure 4.10b. Note that the edge e_2 and the nodes r_2 and r_3 appear in both of the new edges in Figure 4.10b.

In SAGE2 we use the flow on the edges to assist the merging process. The flow tells us how many times an edge is represented in the sampled genome. Consider the example shown in Figure 4.11a in which we are trying to resolve the ambiguous node r_3. Assume that there is paired-end support to merge edge e_1 with e_3 and e_2 with e_3. If we only consider paired-end



(a) Ambiguous node r_3.

(b) Node r_3 is resolved.

Figure 4.11: An example of an ambiguous node resolved using flow and paired-end support. The edges must have unambiguous support by the paired-end reads, and the flow must balance properly for the edges to be merged.

support then we would not be able to merge either of the edge pairs because there would be more than one way to merge edge e_3. We can merge the two pairs of edges if we know that the flow on edge e_3 is greater than 1 and the flow on edges e_1 and e_2 is at least 1. The paired-end support and the flow allows us to resolve the ambiguous node as shown in Figure 4.11b.

4.13.2 Overview

One of the biggest differences between SAGE and SAGE2 is how ambiguous nodes are resolved. The algorithmic changes to the resolution of ambiguous nodes significantly reduces the mis-assemblies and increases the contiguity of the edges in the overlap graph in SAGE2. SAGE2 relies on paired-end support and the flow through the overlap graph to resolve ambiguous nodes. Many of the functions in SAGE merged edges based on the topology of the overlap graph, but did not consider paired-end information or the flow. This is what caused many of the mis-assemblies in the contigs and scaffolds produced by SAGE.

For the remaining merging functions from SAGE we have changed the algorithm to deal with certain aspects of the overlap graph in a different order. Most notably, SAGE2 merges edges based on paired-end support through single nodes before merging edges based on a path search. This was changed in order to reduce mis-assemblies by merging the edges that were connected by a single node first, and then merge the remaining edges based on a path search through the graph. This merges edges that are connected by one or more nodes in the overlap graph based on paired-end support.

A new merging function has been implemented in SAGE2 that merges edges that contain a short overlap at the ends of the contigs that is less than the minimum overlap length. These types of edges would not be merged in the previous steps because there would not have been a connection in the graph between them. This is because the overlap length between the terminal nodes is less than the minimum overlap length. All of these changes together have resulted in a better overall assembly of all the data sets that were tested.

The merging process in SAGE2 first tries to merge all ambiguous nodes that have paired-end support on the edges that only span across a single ambiguous node. After all of the single ambiguous nodes have been resolved with paired-end support, SAGE2 then tries to resolve ambiguous nodes using a search through the overlap graph. After all the ambiguous nodes have been resolved with a path search using paired-end support, SAGE2 then tries to merge edges that overlap with each other by 10 bp or more. The details of all the merging functions will be outlined in each of the following subsections.

4.13.3 Resolving single ambiguous nodes with paired-end reads

The first step in the contig merging process for SAGE2 is to resolve single ambiguous nodes. We first calculate the support of all the paired-end reads in edges that are connected to the same node in the overlap graph. For each of these edges we limit the search between the two reads of each pair to a distance along the edges between $\mu - 3\sigma$ and $\mu + 3\sigma$, where μ is the mean of the insert size, and σ is the standard deviation of the insert size that we estimated previously. After finding the support for all of the edges connected to a single node in the overlap graph, we merge the edges that have support above the threshold. Thresholds are needed to determine if there is enough support to merge two edges, and edges are not merged if there is multiple pairs of edges that support a merge.

The algorithm for resolving single ambiguous nodes by paired-end support is described in Algorithm 6. A queue is used to store the support totals for all of the edge pairs, and it is sorted from highest to lowest after all the support values have been calculated. A hash table is used to store the information about which edges are connected to each of the edges in the overlap graph. This allows us to quickly determine if there are any other edges with support over the threshold for any edge in the graph.

Algorithm 6 Resolve single ambiguous nodes by paired-end support.

```
Input: Overlap graph $G$, threshold $t$
$Q \leftarrow \emptyset$                                  ▷ Queue to store the support for edge pairs.
for each node $v \in G$ do
    for each incoming edge $e_i$ connected to $v$ do
        for each outgoing edge $e_o$ connected to $v$ do
            $Q \leftarrow$ support for each edge pair $(e_i, e_o)$ .
Sort $Q$ from highest to lowest
for each edge pair $(e_i, e_o) \in Q$ with support $\geq t$ do
    if no other edges with support $\geq t$ connected to $e_i$ or $e_o$ then
        Merge$(e_i, e_o)$
    else if flow on $e_i$ or $e_o > 1$ then
        Merge$(e_i, e_o)$
return Number of edges merged
```

In SAGE the threshold used for all iterations of merging single ambiguous nodes was set to 5. In SAGE2 we have performed extensive testing to determine thresholds that are adjusted for different coverage levels, genome lengths, and read lengths. For the threshold formulas throughout the rest of this chapter assume that the coverage level is c and the average read length is r.

For merging single ambiguous nodes the threshold value for the first three iterations is $20\frac{c}{r}$, the next three iterations use a threshold of $10\frac{c}{r}$ if the estimated genome length is greater than one billion, and the threshold for the remaining iterations is reduced to $5\frac{c}{r}$ if the estimated genome length is greater than one billion. If the genome size is less than one billion the threshold stays at $20\frac{c}{r}$. We use a genome length threshold of one billion because testing on human data sets showed better results with a lower threshold, but the lower threshold resulted in many mis-assemblies for the small and medium sized genomes. The iterations are repeated until the number of resolved nodes in an iteration is 0.

4.13.4 Resolving ambiguous nodes by path search

In the previous step SAGE2 merged all pairs of edges with paired-end support that span a single ambiguous node. What is left in the overlap graph are ambiguous nodes that span multiple nodes, single ambiguous nodes that do not have enough paired-end support for merging any edges that span them, and single ambiguous nodes that do not have unique paired-end support for merging edges that span them. The next step is to find the paired-end support for edges that span multiple nodes. The algorithm for resolving ambiguous nodes by path search is described in Algorithm 7.

To find support between all pairs of edges in the overlap graph we need to search all of the paths in the overlap graph for all paired-end reads that map to different edges. For a node v in the overlap graph we find all paths from v of length $\leq \mu + 3\sigma$ and store them in a list. For each read r in the edges of v, we check if the paired read s of r is on any of the paths in the list. If there is a path in the list then we mark that path. We then check if there are any other paths between the reads in the list of paths, and we only mark the pair of edges that are adjacent on all of the paths found for v. For each pair of supported edges (e_1, e_2) we store the pair of edges in a hash table and set the support of the pair of edges to 1. If the pair of edges is already in the hash table then we increment the support of the pair of edges in the hash table by 1. We do this for all of the paired-end reads that are on the edges incident to v. This is repeated for all of the nodes in the overlap graph.

We have set threshold limits based on coverage, genome length, and average read length similar to the previous merging step. The first three iterations of the merging process have a threshold value of $20\frac{c}{r}$, the next three iterations use a threshold of $10\frac{c}{r}$ if the estimated genome length is greater than one billion, and the threshold for the remaining iterations is reduced to $5\frac{c}{r}$ if the estimated genome length is greater than one billion. If the genome size is less than

Algorithm 7 Resolve ambiguous nodes by path search.

```
**Input:** Overlap graph $G$, threshold $t$
$Q \leftarrow \emptyset$                                   ▹ Queue to store the support for edge pairs.
**for** each node $v \in G$ **do**
    **for** each incoming edge $e_i$ connected to $v$ **do**
        **for** each read $r \in e_i$ **do**
            **for** each outgoing edge $e_o$ connected to $v$ **do**
                **if** path found for the paired read $s$ of $r$ **then**
                    $Q \leftarrow$ support for each edge pair $(e_i, e_o)$ .
**for** each edge pair $(e_i, e_o) \in Q$ with support $\geq t$ **do**
    **if** no other edges with support $\geq t$ connected to $e_i$ or $e_o$ **then**
        Merge$(e_i, e_o)$
    **else if** flow on $e_i$ or $e_o > 1$  **then**
        Merge$(e_i, e_o)$
```

one billion the threshold stays at $20\frac{c}{r}$. The iterations are repeated until the number of resolved nodes is 0, or less than 100 if the estimated genome length is greater than one billion. The iterations stop at less than 100 for large genomes because testing showed that merging after this resulted in little extension in the larger edges, but a significant increase in mis-assemblies.

4.13.5 Merging short overlapping contigs

Some of the remaining edges in the overlap graph may have support to merge, but they will not be connected in the overlap graph if the terminal reads of the edges do not overlap more than the minimum overlap length. The previous two merging functions would not consider these edges for merging since they are not connected in the overlap graph. To resolve these types of edges we search for overlaps between the ends of the edges in the overlap graph. If there is a overlap that is greater than 10 bp between two edges then we check to see if there is enough paired-end support to merge the edges. Similar to the previous two functions, we limit the search for paired-end reads to a distance along the edges of $\mu + 3\sigma$. The algorithm for merging short overlapping contigs is listed in Algorithm 8.

After calculating the support for all such pairs of edges, we merge each pair of edges if there are no other edges that have support above the threshold. The threshold for merging short

Algorithm 8 Merge short overlapping contigs.

```
Input: Overlap graph $G$, threshold $t$
$Q \leftarrow \emptyset$                                  ▷ Queue to store the support for edge pairs.
for each edge $e \in G$ do
    for each read $r \in e$ do
        if edge found for the paired read $s$ of $r$ then
            $Q \leftarrow$ support for each edge pair $(e_1, e_2)$ .
Sort $Q$ from highest to lowest
for each edge pair $(e_1, e_2) \in Q$ with support $\geq t$ do
    if no other edges with support $\geq t$ connected to $e_1$ or $e_2$ then
        Merge($e_1, e_2$)
```

overlapping edges is set to $5\frac{c}{r}$ for all iterations, and SAGE2 stops when there are no more edges merged.

4.14 Scaffolding

After SAGE2 has finished merging all of the edges in the overlap graph that represent the contigs, the next step is to connect and orient the contigs into scaffolds. Since we have already merged all of the connected edges as much as possible, we must consider edges in the graph that are not connected in the overlap graph. To do this we consider a new graph based on the overlap graph, and this new graph is called a *scaffold graph*.

The procedure in SAGE2 for merging contigs into scaffolds considers the contigs to be nodes in the scaffold graph, and the paired-end reads that link contigs are considered to be edges in the scaffold graph. The weight of the edges in the scaffolds graph is the number of paired-end reads that support a link between contigs. Since the contigs in the scaffold graph are not connected there will be a gap between the contigs. To determine the length of the gap between contigs we use the estimation of the insert size of the paired-end reads connecting contigs to estimate the distance between contigs. We fill the gaps between contigs with the letter “N” to indicate that the information in that gap is unknown.

The scaffolding procedure has been reimplemented in SAGE2 in order to reduce mis-assemblies and extend the contiguity of the scaffolds. In SAGE, the contigs are merged only by their paired-end support. The contigs are merged in the order of the highest supported edges to the lowest supported edges that are above the support threshold. In SAGE2 we not only consider the paired-end support, but we also consider the topology of the scaffold graph when merging contigs. This has resulted in a significantly better scaffolds assembly compared to SAGE.

SAGE2 goes through multiple stages of merging to build the scaffolds from the scaffold graph. For each stage of merging SAGE2 sets varying thresholds for the number of paired-end reads needed to consider two contigs for merging. The first four stages of the merging process in SAGE2 only considers contigs that are longer than 200 bp. The final stage of merging only considers contigs that are shorter than 500 bp.

4.14.1 Merging contained contigs

For the first stage of merging contigs into scaffolds, SAGE2 only considers contigs that have one incoming edge and one outgoing edge. This procedure is similar to contracting composite paths in the overlap graph, except we are contracting composite edges in the scaffold graph instead. For this stage of merging the paired-end support threshold is set to to $10\frac{c}{r}$ for the first five iterations, and then it is reduced to $5\frac{c}{r}$ for the remaining iterations until no more contigs are merged into scaffolds.

Algorithm 9 Merging contained contigs.

```
Input: Scaffold graph $G$, threshold $t$
$Q \leftarrow \emptyset$                    ▷ Queue to store the support for contig pairs.
for each contig $c \in G$ do
    for each contig $c_i$ with support to merge with $c$ do
        $Q \leftarrow$ support for each contig pair $(c, c_i)$ .
Sort $Q$ from highest to lowest
for each contig pair $(c_1, c_2) \in Q$ with support $\geq t$ do
    if no more than one other contig with support $\geq t$ connected to $c_1$ or $c_2$ then
        Merge$(c_1, c_2)$
```

The algorithm for this function is listed in Algorithm 9. The scaffold graph does not need to be built since we can use the modified overlap graph from the previous steps to infer the scaffold graph. The flow and paired-end information from the overlap graph are also used in the scaffold graph for the entire scaffolding procedure. The contigs are merged from those with the highest support to the lowest support.

4.14.2 Merging contained contigs with multiple support

After merging all of the contained contigs what is left are contigs in the scaffold graph that have multiple edges of support for merging. For this next stage of merging SAGE2 only considers contigs that have two or less incoming edges and two or less outgoing edges. The contigs are merged from those with the highest support to the lowest support. The algorithm for this procedure is listed in Algorithm 10.

For this stage of merging the support threshold is set to to $10\frac{c}{r}$ for the first five iterations, and then it is reduced to $5\frac{c}{r}$ for the remaining iterations until no more contig pairs are merged into scaffolds.

Algorithm 10 Merging contained contigs with multiple support

```
Input: Scaffold graph $G$, threshold $t$
$Q \leftarrow \emptyset$                          ▷ Queue to store the support for contig pairs.
for each contig $c \in G$ do
    for each contig $c_i$ with support to merge with $c$ do
        $Q \leftarrow$ support for each contig pair $(c, c_i)$ .
Sort $Q$ from highest to lowest
for each contig pair $(c_1, c_2) \in Q$ with support $\geq t$ do
    if no more than two other contigs with support $\geq t$ connected to $c_1$ or $c_2$ then
        if flow on $c_1$ and $c_2 \geq t$ then
            Merge$(c_1, c_2)$
```

4.14.3 Merging contained contigs with low support

For the third stage of merging the support threshold is set to to 2 for all iterations of merging until no contigs are merged. For this stage of merging only contigs that have one incoming edge and one outgoing edge will be merged. This is the same procedure as Section 4.14.1, but with a lower threshold. SAGE2 then does another stage of merging contained contigs with a threshold of 1. Extensive testing on genomes of all sizes showed that lowering the thresholds to these low levels at this point increases the contiguity of the scaffolds with a negligible increase in mis-assemblies.

4.14.4 Merging short contigs

The final stage of contig merging only considers contigs that are less than 500 bp long. This final stage of merging attempts to build larger scaffolds from the remaining short contigs. For this stage of merging only contigs that have two or less incoming edges and two or less outgoing edges will be merged. The support threshold for this stage of merging is set to 1. After this stage of merging the final scaffolds are completed.

4.15 Results

The main goals for SAGE2 were to reduce the number of mis-assemblies and increase the length of the contigs and scaffolds compared to SAGE. This is difficult because there is usually a trade-off between the number of mis-assemblies and the length of the contigs and scaffolds. Reducing mis-assemblies usually results in shorter contigs and scaffolds, and an increase in the the length of the contigs and scaffolds typically results in an increase in the mis-assemblies. We were able to significantly reduce the mis-assemblies in SAGE2 compared to SAGE, while also increasing the length of the contigs and scaffolds. Our results show that SAGE2 performs as good or better than the current state-of-the-art.

We have tested SAGE2 against three of the top performing *de novo* genome assembly programs; ABySS [38], SGA [37], and SOAPdenovo2 [22]. We have also included the results of SAGE for the data sets that we were able to obtain results. We have done comparisons on two medium sized genomes (M1 and M2) and six human data sets (H1-H6). The details of each data set are listed in Table 4.2.

Table 4.2: Data sets used for genome assembly results.

Data Set	Organism	Accession Number	Read Length	Coverage	Estimated Per-Base Error
M1	*Caenorhabditis elegans*	SRX218989	100	32	0.38%
M2	*Drosophila melanogaster*	SRR823377	100	52	0.77%
H1	*Homo sapiens*	SRR1302280	101	41	0.23%
H2	*Homo sapiens*	ERR194147	101	50	0.24%
H3	*Homo sapiens*	ERX069505	101	54	0.23%
H4	*Homo sapiens*	SRR5279717	150	36	0.24%
H5	*Homo sapiens*	SRR5282272	150	38	0.24%
H6	*Homo sapiens*	GIAB	35-250	60	0.32%

C.elegans (M1) and *D.melanogaster* (M2) are the two medium sized genomes with genome lengths of approximately 100 million bp and 120 million bp respectively. These genomes were chosen because they are highly studied genomes, and both have very reliable reference genomes. The human genome is also highly studied, and the reference genome currently available is very reliable. The human genome is considered a large genome and is approximately 3.2 billion bp in length.

Data sets M1, M2, H2, and H3 were sequenced using the Illumina HiSeq 2000 machine. Data sets H1 and H6 were sequenced using the Illumina HiSeq 2500 machine. Data sets H4 and H5 were sequenced using the Illumina HiSeq X Ten machine. Data sets M1, M2, and H1 to H5 were downloaded from the Sequence Read Archive (`https://www.ncbi.nlm.nih.gov/sra`) website, and H6 was downloaded from the Genome in a Bottle website (`http://jimb.stanford.edu/giab/`).

4.15.1 Medium sized genome results

All of the alignment results for the contigs and scaffolds have been computed using the alignment software called LASER [18]. The results of the mis-assemblies (Mis) and NGA50 for each data set are listed in the following sections, and the complete LASER results are listed in Appendix A.

The results of the contigs and scaffolds for the two medium sized genomes are listed in

Table 4.3: Alignment results for the medium sized genomes.

		M1		M2	
		Mis	NGA50	Mis	NGA50
Contigs	ABySS	128	7,457	596	10,983
	SAGE	356	9,315	640	5,715
	SGA	477	8,862	1,333	7,630
	SOAPdenovo2	138	7,578	1,543	5,295
	SAGE2	208	9,844	676	13,772
Scaffolds	ABySS	438	11,495	1,114	38,540
	SAGE	973	18,191	2,696	40,249
	SGA	526	12,767	1,805	20,706
	SOAPdenovo2	296	17,607	1,980	23,552
	SAGE2	435	17,584	1,489	48,671

Table 4.3. For the contigs of the M1 data set ABySS and SOAPdenovo2 had the fewest mis-assemblies, but SAGE and SAGE2 both had a significantly longer NGA50. Although, SAGE2 had approximately 40% less mis-assemblies and a slightly longer NGA50 than SAGE. SAGE2 had a slightly lower NGA50 for the scaffolds than SAGE, but SAGE also had more than 50% more mis-assemblies than SAGE2. SOAPdenovo2 had the second best NGA50 for the M1 scaffolds but only slightly lower than the best, yet it had by far the least amount of mis-assemblies.

For the contigs of the M2 data set SAGE2 had the third fewest mis-assemblies, but it had by far the largest NGA50. SAGE2 had less than half the mis-assemblies of both SGA and SOAPdenovo2, but it also had an NGA50 that was more than double that of SAGE and SOAPdenovo2, and nearly double that of SGA. For the scaffolds of the M2 data set SAGE2 had the second fewest mis-assemblies, and a significantly larger NGA50 than all other programs.

Figure 4.12 shows a plot that compares the NGA50 to the mis-assemblies of the contigs for data sets M1 and M2. The best performing programs will appear in the upper left of the figure, and the worst performing programs will appear in the bottom right of the figure. This figure shows that SAGE2 is the best performing program since it is in the upper left of the chart, followed by ABySS. SOAPdenovo2 is the worst performing program because it is in the bottom right of the figure, followed by SAGE and SGA.

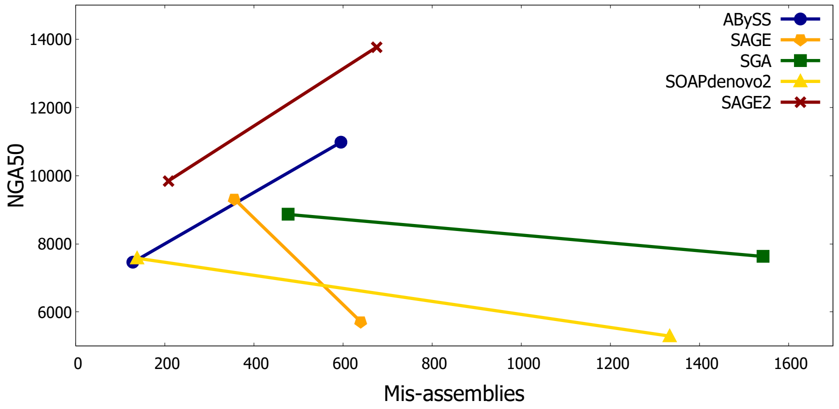


Figure 4.12: Plot comparing NGA50 to mis-assemblies of the contigs for the medium sized genome data sets. The best performing programs are in the top left, and the worst performing programs are in the bottom right.

Figure 4.13 shows a plot that compares the NGA50 to the mis-assemblies of the scaffolds for data sets M1 and M2. This chart shows that SAGE2 is the best performing program since it is in the upper left of the figure, followed by ABySS. SGA is the worst performing program because it is in the bottom right of the figure, followed by SOAPdenovo2 and SAGE.

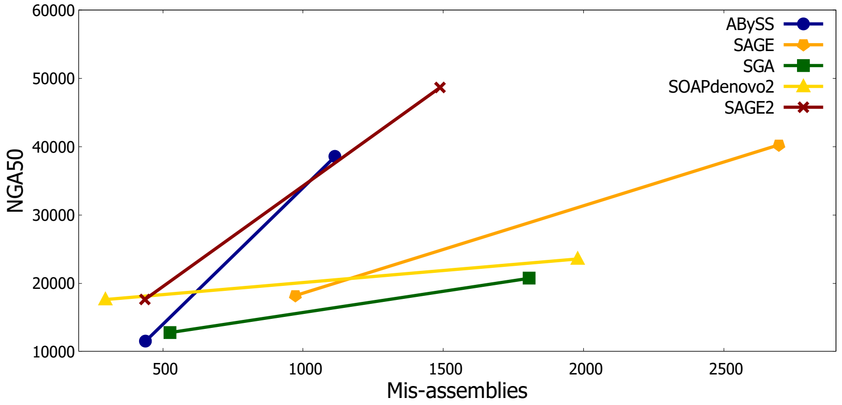


Figure 4.13: Plot comparing NGA50 to mis-assemblies of the scaffolds for the medium sized genome data sets. The best performing programs are in the top left, and the worst performing programs are in the bottom right.

4.15.2 Human genome results

Table 4.4 shows the mis-assemblies and NGA50 results for the human data sets with a read length of 101 bp, and Table 4.5 shows the mis-assemblies and NGA50 results for the human data sets with a read length of 150 bp. For the contig results SAGE2 had the largest NGA50 results for four of the five human data sets, and SGA had a higher NGA50 than SAGE2 for H1 but SGA had nearly twice the mis-assemblies. SAGE2 also had the lowest, or was close to the lowest, in mis-assemblies for H1 to H5.

Table 4.4: Alignment results for 101 bp read length human data sets.

		H1		H2		H3	
		Mis	NGA50	Mis	NGA50	Mis	NGA50
Contigs	ABySS	1,196	3,285	1,677	4,713	1,646	4,532
	SGA	1,908	4,006	2,108	4,405	2,028	4,399
	SOAPdenovo2	1,012	2,594	1,650	3,224	1,862	3,260
	SAGE2	959	3,458	1,616	5,632	1,675	5,804
Scaffolds	ABySS	6,315	4,531	17,656	38,012	17,897	40,955
	SGA	3,433	5,364	12,152	15,379	13,488	15,440
	SOAPdenovo2	26,342	9,826	105,209	38,774	104,593	37,070
	SAGE2	3,399	8,138	21,370	41,082	24,629	43,230

Table 4.5: Alignment results for 150 bp read length human data sets.

		H4		H5	
		Mis	NGA50	Mis	NGA50
Contigs	ABySS	2,126	9,932	2,032	9,258
	SGA	2,653	6,629	2,689	6,568
	SOAPdenovo2	2,328	4,964	2,353	4,815
	SAGE2	2,125	13,025	2,215	12,756
Scaffolds	ABySS	11,957	42,858	13,609	47,667
	SGA	9,994	24,249	3,188	12,745
	SOAPdenovo2	94,817	51,178	100,925	53,583
	SAGE2	45,534	39,803	46,187	47,051

For the scaffolding results SAGE2 had the largest NGA50 for H1 - H3, and the lowest mis-assemblies for the H1 data set. For H2 and H3 the mis-assemblies for SAGE2 were comparable to ABySS, but SAGE2 had a much larger NGA50. ABySS had the best scaffolding results for H4 and H5. SGA had the lowest mis-assemblies for H2 - H5, but a very low NGA50. SOAPdenovo2 had relatively good NGA50 results but made between 5 to 8 times the number of mis-assemblies compared to ABySS and SAGE2.

Figure 4.14 shows a plot that compares the NGA50 to the mis-assemblies of the contigs for data sets H1 to H5. This chart shows that SAGE2 is the best performing program since it is in the upper left of the figure, followed by ABySS. SOAPdenovo2 is the worst performing program, followed closely by SGA.

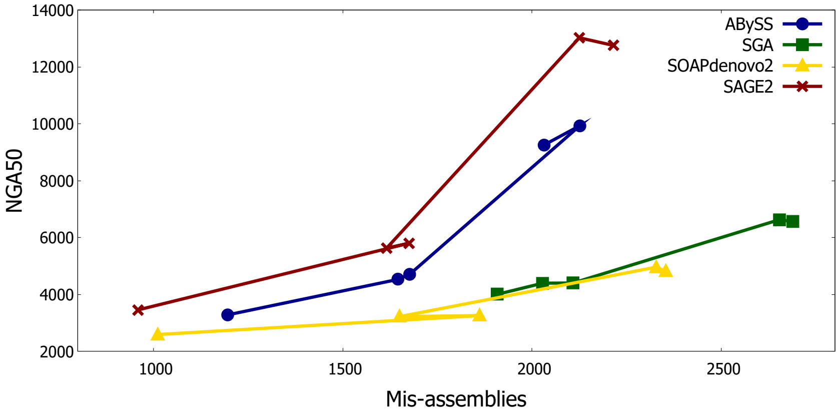


Figure 4.14: Plot comparing NGA50 to mis-assemblies of the contigs for the human genome data sets. The best performing programs are in the top left, and the worst performing programs are in the bottom right.

Figure 4.15 shows a plot that compares the NGA50 to the mis-assemblies of the scaffolds for data sets H1 to H5. This chart shows that there is mixed results for ABySS, SGA, and SAGE2 for the scaffolds of H1 to H5, but it is clear that SOAPdenovo2 is very poor at scaffolding for the human genome data sets due to the large number of mis-assemblies.

Data set H6 is from an Illumina HiSeq 2500 machine run in “Rapid Run Mode” which outputs reads with a length of 250 bp. We tested H6 with ABySS, SGA, SOAPdenovo2, SAGE2, and a program called DISCOVAR [42] from the Broad Institute, which is a program

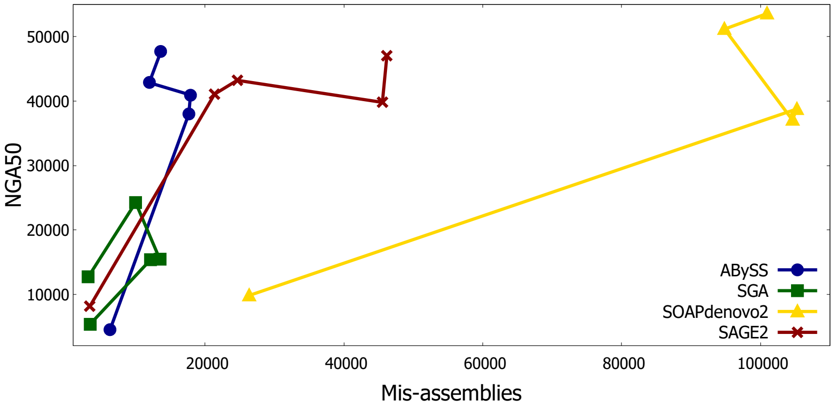


Figure 4.15: Plot comparing NGA50 to mis-assemblies of the scaffolds for the human genome data sets. The best performing programs are in the top left, and the worst performing programs are in the bottom right.

that is designed specifically for data sets with read lengths of 250 bp. SOAPdenovo2 and DISCOVAR were not able to assemble this data set because they ran out of space when using 1TB of RAM. SGA was able to assemble the data set, but LASER was not able to obtain the alignment results with 1 TB of RAM. Only ABySS and SAGE2 were capable of assembling H6 well enough to obtain the alignment results of the contigs.

For the H6 contigs ABySS had the longest NGA50 and the lowest mis-assemblies compared to SAGE2. The NGA50 for ABySS was 9,913bp and 5,844bp for SAGE2. ABySS made 4,181 mis-assemblies compared to 5,159 for SAGE2. It was not possible to obtain the results of the scaffolds for either program because LASER ran out of available memory with 1 TB of RAM.

4.15.3 Time and space usage

Table 4.6 shows the time in hours for each program to assemble all of the data sets tested. All testing was completed using the SHARCNet (www.sharcnet.ca) cluster Goblin with a 32 core Intel Xeon processor at 2.2 GHz and 1 TB of RAM. SOAPdenovo2 was not able to assemble H6 within the memory limits. The time for DISCOVAR is not listed since it was not able to assemble the one data set it was designed for.

Table 4.6: Run time in hours for all data sets tested.

Data Set	ABySS	SGA	SOAPdenovo2	SAGE2
M1	1.20	2.04	0.47	0.64
M2	2.43	7.89	0.73	1.40
H1	53.96	123.66	19.53	31.07
H2	54.30	183.87	13.19	36.08
H3	71.02	205.59	15.46	43.15
H4	58.46	201.09	17.97	38.06
H5	47.21	184.17	16.52	38.38
H6	39.07	449.76	-	58.04

Table 4.7: Space used in MB for all data sets tested.

Data Set	ABySS	SGA	SOAPdenovo2	SAGE2
M1	16,943	4,143	31,593	11,115
M2	21,579	5,966	52,493	18,488
H1	205,415	64,762	344,193	350,668
H2	222,751	80,750	344,270	448,624
H3	169,044	92,092	344,302	462,999
H4	375,848	56,808	464,869	238,363
H5	347,756	53,232	423,641	237,811
H6	866,506	76,443	-	282,262

SOAPdenovo2 was by far the quickest at assembling the human genome data sets, and was able to assemble H1 to H5 in less than one day each. SAGE2 was the next fastest program and was able to assemble H1 to H5 in less than two days each, and less than three days for H6. ABySS was able to assemble H5 and H6 in just under two days, and H1 to H4 in less than three days. SGA was by far the slowest program to assemble the human data sets and took over one week to assemble data sets H2 to H5, and took over three weeks to assemble H6.

Table 4.7 shows the space used in MB for each program to assemble all of the data sets tested. SGA used by far the least amount of memory, but still required between 56GB to 92GB of RAM to assemble the human data sets. ABySS was the next best at memory usage for H1 to H3, but SAGE2 used less memory than ABySS for H4 to H6. This is because H4 to H6 have longer read lengths which requires less memory for SAGE2 to store the information about the reads on each of the edges in the overlap graph. This is clearly an issue for ABySS in the H6 data set, which ABySS needed 866GB of RAM to assemble compared to 282GB of RAM for SAGE2.

4.16 Conclusions

The main problems with SAGE have been addressed in our new implementation called SAGE2. The largest bottleneck in run time was building the overlap graph in serial, and we have implemented a parallel approach to building the overlap graph that is both time and space efficient. This allows SAGE2 to build the overlap graph from any sized genome and any coverage level nearly as fast as the DBG assemblers. We have also parallelized many of the functions in SAGE2 so that it can assemble large genomes with high coverage levels in a similar amount of time as the current state-of-the-art de novo*de novo* genome assemblers.

The changes made in SAGE2 has significantly reduced the mis-assemblies, while also extending the length of the contigs and scaffolds compared to SAGE. The biggest difference is in the quality of the assemblies for large genomes. SAGE2 is able to assemble the human genome as good or better than the current state-of-the-art for all of the sequencing technologies from Illumina, except the Illumina HiSeq 2500 machine run in Rapid Run Mode. SAGE2 is also capable of assembling data sets with varying read lengths so that multiple data sets with different read length can be used in a single assembly, or one data set with varying read lengths.

Chapter 5

Conclusions and Future Research

5.1 Conclusions

We have reimplemented the hash table in RACER so that it can quickly and efficiently perform error corrections for any sized genome and coverage level. We have also determined appropriate automatic parameter selection for a wide range of genome sizes and coverage levels. RACER now performs as good or better than the state-of-the-art for all genome sizes, coverage levels, and sequencing technologies from Illumina.

We have performed a comprehensive assessment of the current state-of-the-art error correction software available for the Illumina technologies, and we have introduced a methodology of standardizing the assessment of the quality of the corrections that is thorough and unbiased. The information we have provided is extremely important for any researcher that uses DNA sequencing technologies.

The main problems with SAGE have been addressed in SAGE2. The new parallel implementation of overlap graph construction is both time and space efficient. The new algorithmic changes to the merging process in SAGE2 has significantly reduced the mis-assemblies, while also extending the length of the contigs and scaffolds compared to SAGE. SAGE2 is also capable of assembling data sets with varying read lengths so that multiple data sets with different read length can be used in a single assembly, or one data set with varying read lengths.

5.2 Future research

5.2.1 RACER

RACER has been designed to correct substitution errors from the Illumina technologies. The Illumina technologies produce short reads between 100 bp to 300 bp, and the errors tend to be at the 3′ ends of the reads. The newest sequencing technologies produce very long reads, from the tens of thousands to hundreds of thousands of bases per read. Although, the error rate is much higher than the Illumina technologies. RACER can still correct this type of data, but it will require extensive testing to determine the proper automatic parameter selection in order to correct this type of data well.

RACER assumes that the data that is provided is from a single genome, but the Illumina technologies can also produce data sets for metagenomic data. Metagenomic data contains reads from many different species, and the coverage levels from each species can vary significantly. This type of data may have many similar *k*-mers since many of the species may be closely related. RACER may assume that many of the low coverage *k*-mers are errors, but they are actually low coverage genomes. This will cause RACER to change *k*-mers from the low coverage genomes to *k*-mers from the higher coverage genomes, which will cause problems for metagenomic assemblers. RACER will need to be modified to correct metagenomic data properly.

5.2.2 Error correction evaluation

Both readSearch and kmerSearch were designed for the Illumina technologies. The newest sequencing technologies produce reads that are much longer, and it would be extremely difficult to correct any of the whole reads from these technologies. Since readSearch requires whole reads to be correct, it is not suitable for error correction evaluation from these technologies. The kmerSearch software only requires short segments of length *k* to be correct, and therefore it can still be used to asses the performance of correction from the newest sequencing technologies. Testing of kmerSearch on the newest technologies would be required to check for any unforeseen problems with assessing the performance of correction from these technologies.

5.2.3 SAGE2

Extensive testing has shown that SAGE2 produces long contigs with few mis-assemblies compared to the state-of-the-art. One area of the assemblies that can be improved upon for the contigs produced by SAGE2 is the percent of the genome that is covered in the longer contigs. This may require either a modification to the current functions that resolve ambiguous nodes, or it may require a new implementation that tries to merge the smaller contigs together, or into the longer contigs. Care will be needed to merge them correctly so that the genome fraction covered will increase, but the number of mis-assemblies stays low.

Another problem that needs to be addressed is the number of mis-assemblies of the scaffolds produced by SAGE2. The current implementation only checks the relation of contig pairs, and the other contigs that they have support with. SAGE2 does not search any paths along the scaffold graph to check the topology of a larger region before merging. SAGE2 will require a new implementation to the scaffold graph that can search paths along the scaffold graph, or to incorporate long read sequence data to scaffold the contigs. This could significantly decrease the number of mis-assemblies in the scaffolds produced by SAGE2, and proper merging could also increase the length of the scaffolds as well.

Finally, the newest sequencing technologies produce reads that are very long and have many errors, even after correcting the errors. SAGE2 is not capable of assembling genomes from these technologies because it requires that there be exact overlaps between whole reads for there to be an edge in the overlap graph between them. To assemble genomes from these sequencing technologies would require that SAGE2 be able to handle overlaps between reads that are not exact. This is a difficult problem and would require a new implementation to be able to build an overlap graph from reads that do not overlap exactly.

Bibliography

[1] Serafim Batzoglou, David B Jaffe, Ken Stanley, Jonathan Butler, Sante Gnerre, Evan Mauceli, Bonnie Berger, Jill P Mesirov, and Eric S Lander. Arachne: a whole-genome shotgun assembler. *Genome research*, 12(1):177–189, 2002.

[2] Jonathan Butler, Iain MacCallum, Michael Kleber, Ilya A Shlyakhter, Matthew K Belmonte, Eric S Lander, Chad Nusbaum, and David B Jaffe. Allpaths: de novo assembly of whole-genome shotgun microreads. *Genome research*, 18(5):810–820, 2008.

[3] German Cancer Research Center. The illumina hiseq 2000 sequencing technology, 2015. URL `https://www.dkfz.de/gpcf/hiseq_technology.html`.

[4] Nicolaas Govert De Bruijn. A combinatorial problem. 1946.

[5] Robert D Fleischmann, Mark D Adams, Owen White, Rebecca A Clayton, et al. Whole-genome random sequencing and assembly of haemophilus influenzae rd. *Science*, 269 (5223):496, 1995.

[6] André Goffeau, Bart G Barrell, Howard Bussey, RW Davis, et al. Life with 6000 genes. *Science*, 274(5287):546, 1996.

[7] Andrew V Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. *Journal of algorithms*, 22(1):1–29, 1997.

[8] Alexey Gurevich, Vladislav Saveliev, Nikolay Vyahhi, and Glenn Tesler. Quast: quality assessment tool for genome assemblies. *Bioinformatics*, 29(8):1072–1075, 2013.

[9] Md Bahlul Haider. *A new algorithm for de novo genome assembly*. PhD thesis, The University of Western Ontario, 2012.

[10] Yun Heo, Xiao-Long Wu, Deming Chen, Jian Ma, and Wen-Mei Hwu. Bless: bloom filter-based error correction solution for high-throughput sequencing reads. *Bioinformatics*, page btu030, 2014.

[11] David Hernandez, Patrice François, Laurent Farinelli, Magne Østerås, and Jacques Schrenzel. De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer. *Genome research*, 18(5):802–809, 2008.

[12] Lucian Ilie and Michael Molnar. Racer: Rapid and accurate correction of errors in reads. *Bioinformatics*, page btt407, 2013.

[13] Lucian Ilie, Farideh Fazayeli, and Silvana Ilie. Hitec: accurate error correction in high-throughput sequencing data. *Bioinformatics*, 27(3):295–302, 2011.

[14] Lucian Ilie, Bahlul Haider, Michael Molnar, and Roberto Solis-Oba. Sage: string-overlap assembly of genomes. *BMC bioinformatics*, 15(1):302, 2014.

[15] Marc TJ Johnson, Eric J Carpenter, Zhijian Tian, Richard Bruskiewich, Jason N Burris, Charlotte T Carrigan, Mark W Chase, Neil D Clarke, Sarah Covshoff, Patrick P Edger, et al. Evaluating methods for isolating total rna and predicting the success of sequencing phylogenetically diverse plant transcriptomes. *PloS one*, 7(11):e50226, 2012.

[16] John D Kececioglu and Eugene W Myers. Combinatorial algorithms for dna sequence assembly. *Algorithmica*, 13(1-2):7, 1995.

[17] David R Kelley, Michael C Schatz, and Steven L Salzberg. Quake: quality-aware detection and correction of sequencing errors. *Genome biology*, 11(11):R116, 2010.

[18] Nilesh Khiste and Lucian Ilie. Laser: Large genome assembly evaluator. *BMC research notes*, 8(1):709, 2015.

[19] Eric S Lander, Lauren M Linton, Bruce Birren, Chad Nusbaum, Michael C Zody, Jennifer Baldwin, Keri Devon, Ken Dewar, Michael Doyle, William FitzHugh, et al. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001.

[20] Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[21] Ruiqiang Li, Yingrui Li, Karsten Kristiansen, and Jun Wang. Soap: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713–714, 2008.

[22] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

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

[24] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[25] Michael L Metzker. Sequencing technologiesthe next generation. *Nature reviews genetics*, 11(1):31–46, 2010.

[26] Jason R Miller, Arthur L Delcher, Sergey Koren, Eli Venter, Brian P Walenz, Anushka Brownley, Justin Johnson, Kelvin Li, Clark Mobarry, and Granger Sutton. Aggressive assembly of pyrosequencing reads with mates. *Bioinformatics*, 24(24):2818–2824, 2008.

[27] Michael Molnar and Lucian Ilie. Correcting illumina data. *Briefings in bioinformatics*, 16(4):588–599, 2015.

[28] Eugene W Myers. The fragment assembly string graph. *Bioinformatics*, 21(suppl 2): ii79–ii85, 2005.

[29] Eugene W Myers, Granger G Sutton, Art L Delcher, Ian M Dew, Dan P Fasulo, Michael J Flanigan, Saul A Kravitz, Clark M Mobarry, Knut HJ Reinert, Karin A Remington, et al. A whole-genome assembly of drosophila. *Science*, 287(5461):2196–2204, 2000.

[30] Pavel A Pevzner, Haixu Tang, and Michael S Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

[31] Leslie Pray. Discovery of dna structure and function: Watson and crick. *Nature Education*, 1(1):100, 2008.

[32] Leena Salmela and Jan Schröder. Correcting errors in short reads by multiple alignments. *Bioinformatics*, 27(11):1455–1461, 2011.

[33] Frederick Sanger, Steven Nicklen, and Alan R Coulson. Dna sequencing with chain-terminating inhibitors. *Proceedings of the national academy of sciences*, 74(12):5463–5467, 1977.

[34] Michael C Schatz, Arthur L Delcher, and Steven L Salzberg. Assembly of large genomes using second-generation sequencing. *Genome research*, 20(9):1165–1173, 2010.

[35] Jan Schröder, Heiko Schröder, Simon J Puglisi, Ranjan Sinha, and Bertil Schmidt. Shrec: a short-read error correction method. *Bioinformatics*, 25(17):2157–2163, 2009.

[36] Let's Talk Science's Education Services. Sanger sequencing, 2012. URL `https://explorecuriocity.org/Explore/ArticleId/2027/sanger-sequencing-2027.aspx`.

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

[38] Jared T Simpson, Kim Wong, Shaun D Jackman, Jacqueline E Schein, Steven JM Jones,

and Inanç Birol. Abyss: a parallel assembler for short read sequence data. *Genome research*, 19(6):1117–1123, 2009.

[39] Lloyd M Smith, Jane Z Sanders, Robert J Kaiser, Peter Hughes, Chris Dodd, Charles R Connell, Cheryl Heiner, Stephen BH Kent, and Leroy E Hood. Fluorescence detection in automated dna sequence analysis. 1986.

[40] J Craig Venter, Mark D Adams, Eugene W Myers, Peter W Li, Richard J Mural, Granger G Sutton, Hamilton O Smith, Mark Yandell, Cheryl A Evans, Robert A Holt, et al. The sequence of the human genome. *science*, 291(5507):1304–1351, 2001.

[41] I VIIEW. Genome sequence of the nematode c. elegans: a platform for investigating biology. 1998.

[42] Neil I Weisenfeld, Shuangye Yin, Ted Sharpe, Bayo Lau, Ryan Hegarty, Laurie Holmes, Brian Sogoloff, Diana Tabbaa, Louise Williams, Carsten Russ, et al. Comprehensive variation discovery in single human genomes. *Nature genetics*, 46(12):1350, 2014.

[43] Xiao Yang, Karin S Dorman, and Srinivas Aluru. Reptile: representative tiling for short read error correction. *Bioinformatics*, 26(20):2526–2533, 2010.

[44] Xiao Yang, Sriram P Chockalingam, and Srinivas Aluru. A survey of error-correction methods for next-generation sequencing. *Briefings in bioinformatics*, 14(1):56–66, 2013.

[45] Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome research*, 18(5):821–829, 2008.

Appendix A

Complete LASER Alignment Results For Genome Assemblies

Table A.1: LASER results for M1 contigs.

Program:	**ABySS**	**SGA**	**SOAPdenovo2**	**SAGE2**
Overlap/*k*-mer Length:	**53**	**74**	**57**	**58**
# contigs (>= 0 bp)	105,099	172,597	197,613	38,356
# contigs (>= 1000 bp)	15,248	14,395	15,258	13,305
Total length (>= 0 bp)	98,514,187	111,083,146	106,535,818	94,721,723
Total length (>= 1000 bp)	83,706,934	87,209,296	84,651,899	86,532,900
# contigs	21,743	20,110	21,765	18,498
Largest contig	103,434	111,003	84,339	113,941
Total length	88,376,057	91,270,378	89,300,027	90,229,438
Reference length	100,286,070	100,286,070	100,286,070	100,286,070
GC (%)	35.46	35.53	35.47	35.47
Reference GC (%)	35.44	35.44	35.44	35.44
N50	9,556	10,671	9,505	12,307
NG50	7,614	9,112	7,760	10,148
N75	3,593	4,270	3,701	4,706
NG75	2,065	2,893	2,241	2,976
L50	2,281	2,166	2,381	1,892
LG50	2,977	2,624	3,020	2,343
L75	6,069	5,566	6,153	4,870
LG75	9,351	7,490	9,024	6,876
# misassemblies	128	477	138	208
# misassembled contigs	120	445	129	188
Misassembled contigs length	1,076,875	2,485,018	1,074,403	1,744,887
# local misassemblies	759	931	798	924
# unaligned contigs	974+586part	509+771part	936+696part	719+802part
Unaligned length	1,920,483	1,998,366	1,981,410	2,031,497
Genome fraction (%)	85.95	88.44	86.72	87.39
Duplication ratio	1.00	1.01	1.01	1.01
# N's per 100 kbp	0.00	0.00	0.00	0.00
# mismatches per 100 kbp	80.22	83.61	79.75	83.41
# indels per 100 kbp	28.08	29.07	28.34	29.36
Largest alignment	103,384	111,003	84,338	113,941
NA50	9,425	10,439	9,373	12,007
NGA50	7,457	8,862	7,578	9,844
NA75	3,402	3,956	3,476	4,412
NGA75	1,874	2,588	2,037	2,661
LA50	2,304	2,196	2,407	1,925
LGA50	3,011	2,665	3,058	2,387
LA75	6,222	5,760	6,321	5,038
LGA75	9,766	7,868	9,431	7,241

Table A.2: LASER results for M1 scaffolds.

Program:	**ABySS**	**SGA**	**SOAPdenovo2**	**SAGE2**
Overlap/*k*-mer Length:	**53**	**74**	**57**	**58**
# contigs (>= 0 bp)	76,775	26,823	48,098	29,008
# contigs (>= 1000 bp)	12,555	11,568	9,799	9,694
Total length (>= 0 bp)	98,348,326	94,832,897	98,367,265	95,571,402
Total length (>= 1000 bp)	88,415,420	88,458,304	89,959,162	89,545,218
# contigs	16,767	15,833	13,682	13,315
Largest contig	137,054	137,087	214,293	197,631
Total length	91,453,545	91,499,103	92,707,527	92,118,625
Reference length	100,286,070	100,286,070	100,286,070	100,286,070
GC (%)	35.52	35.52	35.46	35.49
Reference GC (%)	35.44	35.44	35.44	35.44
N50	13,905	15,327	21,007	21,022
NG50	11,920	13,269	18,620	18,357
N75	5,371	6,079	7,953	8,021
NG75	3,644	4,065	5,530	5,507
L50	1,646	1,527	1,097	1,111
LG50	1,991	1,835	1,289	1,320
L75	4,310	3,888	2,880	2,865
LG75	5,807	5,206	3,735	3,783
# misassemblies	438	526	296	435
# misassembled contigs	389	450	251	357
Misassembled contigs length	3,446,069	3,920,434	3,864,799	5,716,638
# local misassemblies	1,184	1,364	4,013	1,456
# unaligned contigs	624+615part	374+623part	1607+1382part	460+587part
Unaligned length	2,020,022	1,955,963	4,888,210	2,000,931
Genome fraction (%)	88.39	88.86	86.15	88.97
Duplication ratio	1.01	1.01	1.02	1.01
# N's per 100 kbp	34.93	25.56	1,482.59	0.00
# mismatches per 100 kbp	92.46	84.83	78.92	87.62
# indels per 100 kbp	31.36	30.04	39.82	32.10
Largest alignment	137,054	137,087	213,479	197,397
NA50	13,469	14,876	19,883	20,207
NGA50	11,495	12,767	17,607	17,584
NA75	4,949	5,572	6,676	7,269
NGA75	3,221	3,580	4,237	4,792
LA50	1,683	1,562	1,143	1,153
LGA50	2,039	1,881	1,346	1,371
LA75	4,490	4,074	3,118	3,033
LGA75	6,143	5,552	4,182	4,074

Table A.3: LASER results for M2 contigs.

Program:	**ABySS**	**SGA**	**SOAPdenovo2**	**SAGE2**
Overlap/*k*-mer Length:	**46**	**80**	**67**	**60**
# contigs (>= 0 bp)	192,784	404,378	354,153	77,409
# contigs (>= 1000 bp)	15,407	20,587	24,779	13,447
Total length (>= 0 bp)	131,724,039	166,674,311	152,449,213	129,584,414
Total length (>= 1000 bp)	112,697,396	111,924,801	106,863,222	114,429,019
# contigs	19,143	27,895	35,598	18,352
Largest contig	147,335	104,592	92,987	145,646
Total length	115,355,075	117,086,387	114,617,818	117,876,112
Reference length	120,381,546	120,381,546	120,381,546	120,381,546
GC (%)	42.51	42.43	42.48	42.42
Reference GC (%)	42.41	42.41	42.41	42.41
N50	12,347	8,517	6,085	15,129
NG50	11,636	8,157	5,625	14,725
N75	5,734	3,695	2,599	6,921
NG75	5,039	3,409	2,230	6,468
L50	2,438	3,443	4,709	2,044
LG50	2,648	3,640	5,202	2,129
L75	5,877	8,696	12,002	4,908
LG75	6,579	9,392	13,797	5,189
# misassemblies	596	1,543	1,333	676
# misassembled contigs	566	1,437	1,274	628
Misassembled contigs length	9,607,977	11,168,381	7,846,455	11,783,345
# local misassemblies	1,164	1,236	1,169	1,282
# unaligned contigs	2525+987part	2731+1801part	3009+1983part	3093+1129part
Unaligned length	4,255,261	6,584,207	5,778,387	6,169,296
Genome fraction (%)	92.05	91.12	89.84	92.04
Duplication ratio	1.00	1.01	1.01	1.01
# N's per 100 kbp	0.00	0.00	0.00	0.00
# mismatches per 100 kbp	536.55	529.96	516.71	539.91
# indels per 100 kbp	122.50	117.04	116.50	127.14
Largest alignment	147,335	104,537	82,615	133,959
NA50	11,661	7,959	5,716	14,171
NGA50	10,983	7,630	5,295	13,772
NA75	5,314	3,233	2,316	6,175
NGA75	4,631	2,914	1,937	5,745
LA50	2,583	3,662	4,967	2,186
LGA50	2,805	3,874	5,490	2,276
LA75	6,236	9,436	12,909	5,305
LGA75	6,998	10,240	14,950	5,620

Table A.4: LASER results for M2 scaffolds.

Program: **Overlap/*k*-mer Length:**	**ABySS** **46**	**SGA** **80**	**SOAPdenovo2** **67**	**SAGE2** **60**
# contigs (>= 0 bp)	148,586	35,839	106,133	62,942
# contigs (>= 1000 bp)	6,537	9,788	9,039	5,349
Total length (>= 0 bp)	133,003,932	126,032,492	136,929,581	130,894,591
Total length (>= 1000 bp)	120,403,787	117,603,937	120,064,923	117,886,146
# contigs	7,725	12,484	11,924	8,867
Largest contig	419,589	182,862	391,195	518,070
Total length	121,254,060	119,456,571	122,072,236	120,338,872
Reference length	120,381,546	120,381,546	120,381,546	120,381,546
GC (%)	42.46	42.37	42.38	42.38
Reference GC (%)	42.41	42.41	42.41	42.41
N50	45,012	22,988	29,265	66,932
NG50	45,538	22,789	29,696	66,932
N75	20,158	10,738	12,894	27,813
NG75	20,549	10,476	13,450	27,790
L50	740	1,383	1,093	493
LG50	730	1,403	1,064	493
L75	1,734	3,258	2,664	1,176
LG75	1,702	3,323	2,568	1,177
# misassemblies	1,114	1,805	1,980	1,489
# misassembled contigs	873	1,452	1,348	852
Misassembled contigs length	36,550,041	27,443,425	33,876,149	50,469,631
# local misassemblies	1,865	3,529	4,265	2,421
# unaligned contigs	1998+604part	1688+1424part	2485+2719part	2366+789part
Unaligned length	5,772,573	6,987,597	15,058,979	6,983,406
Genome fraction (%)	93.74	92.60	85.23	93.18
Duplication ratio	1.02	1.01	1.04	1.01
# N's per 100 kbp	148.07	334.64	1,688.81	9.53
# mismatches per 100 kbp	547.92	542.80	499.29	539.70
# indels per 100 kbp	131.74	129.06	143.29	131.04
Largest alignment	320,267	175,542	217,716	462,403
NA50	38,194	20,946	23,022	48,671
NGA50	38,540	20,706	23,552	48,671
NA75	17,021	9,280	6,908	21,502
NGA75	17,422	9,078	7,541	21,455
LA50	891	1,548	1,336	683
LGA50	879	1,571	1,300	683
LA75	2,072	3,652	3,641	1,597
LGA75	2,034	3,728	3,466	1,599

Table A.5: LASER results for H1 contigs.

Program:	**ABySS**	**SGA**	**SOAPdenovo2**	**SAGE2**
Overlap/*k*-mer Length:	**71**	**78**	**65**	**68**
# contigs (>= 0 bp)	3,807,863	7,475,987	9,232,099	2,929,342
# contigs (>= 1000 bp)	654,426	595,224	720,840	601,226
Total length (>= 0 bp)	2,948,618,719	3,426,420,918	3,352,005,758	2,966,355,413
Total length (>= 1000 bp)	2,379,169,532	2,443,974,781	2,281,485,489	2,334,034,418
# contigs	895,450	780,169	1,029,543	924,627
Largest contig	64,639	79,480	54,595	66,911
Total length	2,553,429,133	2,577,142,475	2,504,557,120	2,565,458,628
Reference length	3,209,286,105	3,209,286,105	3,209,286,105	3,209,286,105
GC (%)	40.69	40.66	40.51	40.57
Reference GC (%)	40.99	40.99	40.99	40.99
N50	4,545	5,391	3,654	4,772
NG50	3,288	4,014	2,598	3,463
N75	2,301	2,788	1,908	2,348
NG75	922	1,135	721	835
L50	157,734	137,359	195,364	150,465
LG50	242,708	205,366	309,866	229,699
L75	355,457	303,281	432,696	341,367
LG75	683,369	560,513	867,869	680,971
# misassemblies	1,196	1,908	1,012	959
# misassembled contigs	1,190	1,894	1,005	913
Misassembled contigs length	3,905,048	7,943,176	2,507,040	3,385,961
# local misassemblies	2,454	2,710	2,243	5,474
# unaligned contigs	2538+3129part	3219+11042part	2479+17096part	2666+24838part
Unaligned length	2,435,127	3,520,374	2,550,340	3,021,762
Genome fraction (%)	79.07	79.78	77.46	77.50
Duplication ratio	1.01	1.01	1.01	1.03
# N's per 100 kbp	0.00	0.00	0.00	0.00
# mismatches per 100 kbp	89.81	89.21	87.66	90.72
# indels per 100 kbp	16.29	13.30	14.04	18.54
Largest alignment	64,639	79,480	54,593	64,677
NA50	4,542	5,382	3,650	4,765
NGA50	3,285	4,006	2,594	3,458
NA75	2,298	2,780	1,904	2,342
NGA75	918	1,126	714	829
LA50	157,840	137,554	195,512	150,621
LGA50	242,882	205,690	310,159	229,971
LA75	355,740	303,840	433,204	341,871
LGA75	684,382	562,277	870,475	683,306

Table A.6: LASER results for H1 scaffolds.

Program:	**ABySS**	**SGA**	**SOAPdenovo2**	**SAGE2**
Overlap/*k*-mer Length:	**71**	**78**	**65**	**68**
# contigs (>= 0 bp)	2,611,361	976,253	1,895,367	2,452,948
# contigs (>= 1000 bp)	567,371	496,167	339,466	345,357
Total length (>= 0 bp)	2,880,697,724	2,696,673,489	3,058,311,846	2,995,888,780
Total length (>= 1000 bp)	2,500,013,739	2,505,114,340	2,780,623,371	2,463,556,480
# contigs	735,696	618,811	422,048	576,878
Largest contig	75,147	105,368	193,249	126,183
Total length	2,622,166,536	2,593,289,067	2,839,672,455	2,627,513,351
Reference length	3,209,286,105	3,209,286,105	3,209,286,105	3,209,286,105
GC (%)	40.79	40.64	40.68	40.68
Reference GC (%)	40.99	40.99	40.99	40.99
N50	6,077	7,150	13,824	10,984
NG50	4,551	5,387	11,805	8,189
N75	2,997	3,701	6,864	5,146
NG75	1,350	1,523	4,471	1,488
L50	119,642	104,327	58,213	67,041
LG50	175,539	154,006	72,689	97,748
L75	273,490	230,019	130,817	153,820
LG75	487,616	417,512	180,496	298,491
# misassemblies	6,315	3,433	26,342	3,399
# misassembled contigs	5,603	3,317	15,557	2,627
Misassembled contigs length	23,052,332	18,810,546	149,392,224	29,528,307
# local misassemblies	18,014	10,059	1,435,344	49,817
# unaligned contigs	3026+3127part	3577+8657part	4978+8724part	3213+7697part
Unaligned length	3,241,866	4,198,603	10,562,533	3,650,560
Genome fraction (%)	81.17	80.50	81.10	79.21
Duplication ratio	1.01	1.00	1.09	1.03
# N's per 100 kbp	41.54	27.89	7,084.89	0.00
# mismatches per 100 kbp	93.60	92.56	91.65	93.99
# indels per 100 kbp	17.48	15.98	15.12	23.48
Largest alignment	75,147	105,368	189,474	126,151
NA50	6,051	7,121	11,767	10,915
NGA50	4,531	5,364	9,826	8,138
NA75	2,981	3,681	5,034	5,108
NGA75	1,334	1,505	2,552	1,457
LA50	120,078	104,671	65,735	67,463
LGA50	176,221	154,564	82,922	98,379
LA75	274,645	230,953	156,723	154,851
LGA75	490,485	419,870	232,183	301,310

Table A.7: LASER results for H2 contigs.

Program:	**ABySS**	**SGA**	**SOAPdenovo2**	**SAGE2**
Overlap/*k*-mer Length:	**77**	**82**	**75**	**78**
# contigs (>= 0 bp)	3,504,232	9,575,174	7,464,012	1,787,707
# contigs (>= 1000 bp)	556,078	568,900	674,014	493,543
Total length (>= 0 bp)	3,015,950,088	3,718,556,270	3,381,038,091	2,866,766,284
Total length (>= 1000 bp)	2,504,611,265	2,480,254,464	2,396,831,742	2,528,435,349
# contigs	724,989	743,040	936,513	655,647
Largest contig	80,842	78,375	65,608	117,620
Total length	2,626,277,625	2,605,274,187	2,586,140,270	2,644,017,639
Reference length	3,209,286,105	3,209,286,105	3,209,286,105	3,209,286,105
GC (%)	40.70	40.64	40.62	40.66
Reference GC (%)	40.99	40.99	40.99	40.99
N50	6,245	5,864	4,372	7,376
NG50	4,725	4,419	3,233	5,653
N75	3,114	2,989	2,216	3,661
NG75	1,388	1,288	973	1,611
L50	119,382	124,989	165,154	102,316
LG50	173,101	184,357	248,148	146,090
L75	268,087	280,288	373,175	229,059
LG75	473,641	504,524	684,289	398,740
# misassemblies	1,677	2,108	1,650	1,616
# misassembled contigs	1,646	2,041	1,625	1,523
Misassembled contigs length	7,007,899	9,485,460	5,001,004	7,941,798
# local misassemblies	3,178	3,161	2,841	6,153
# unaligned contigs	4259+3540part	5190+8219part	4325+13881part	6440+16543part
Unaligned length	6,321,919	7,469,597	6,048,074	8,580,397
Genome fraction (%)	81.20	80.55	79.85	81.38
Duplication ratio	1.01	1.01	1.01	1.01
# N's per 100 kbp	0.00	0.00	0.00	0.00
# mismatches per 100 kbp	91.22	88.98	86.90	95.00
# indels per 100 kbp	17.84	12.90	14.69	20.32
Largest alignment	80,842	78,375	65,608	117,620
NA50	6,232	5,848	4,363	7,356
NGA50	4,713	4,405	3,224	5,632
NA75	3,103	2,974	2,206	3,642
NGA75	1,372	1,268	959	1,585
LA50	119,574	125,287	165,399	102,562
LGA50	173,406	184,829	248,584	146,471
LA75	268,671	281,151	374,038	229,803
LGA75	475,590	507,433	687,708	401,134

Table A.8: LASER results for H2 scaffolds.

Program: **Overlap/*k*-mer Length:**	**ABySS** **77**	**SGA** **82**	**SOAPdenovo2** **75**	**SAGE2** **78**
# contigs (>= 0 bp)	1,405,048	596,920	1,976,491	1,007,596
# contigs (>= 1000 bp)	132,387	238,250	109,724	118,681
Total length (>= 0 bp)	2,981,605,214	2,753,368,704	3,012,050,220	2,898,086,669
Total length (>= 1000 bp)	2,807,622,942	2,638,523,252	2,713,555,250	2,707,799,231
# contigs	154,890	284,743	145,926	178,086
Largest contig	428,603	222,828	538,085	476,393
Total length	2,823,798,350	2,671,451,186	2,738,907,589	2,749,246,560
Reference length	3,209,286,105	3,209,286,105	3,209,286,105	3,209,286,105
GC (%)	40.85	40.67	40.72	40.77
Reference GC (%)	40.99	40.99	40.99	40.99
N50	46,715	20,290	57,014	51,908
NG50	39,683	15,726	46,576	42,419
N75	23,321	9,932	28,490	25,654
NG75	14,668	4,540	15,143	13,418
L50	17,618	37,091	13,914	15,183
LG50	22,096	52,161	18,482	20,086
L75	38,870	83,958	30,745	33,874
LG75	54,360	142,428	47,408	52,144
# misassemblies	17,656	12,152	105,209	21,370
# misassembled contigs	11,502	8,671	30,027	11,549
Misassembled contigs length	267,107,043	100,471,573	941,127,839	226,572,802
# local misassemblies	63,265	72,885	183,498	147,165
# unaligned contigs	5364+3396part	6410+4959part	7755+19228part	7230+5135part
Unaligned length	13,991,254	13,171,146	55,380,962	16,178,675
Genome fraction (%)	85.20	82.41	80.89	84.18
Duplication ratio	1.03	1.01	1.03	1.01
# N's per 100 kbp	338.57	492.22	2,157.12	98.97
# mismatches per 100 kbp	102.54	97.36	101.95	106.13
# indels per 100 kbp	28.84	25.87	63.55	31.29
Largest alignment	427,174	222,443	536,132	476,249
NA50	44,812	19,848	48,412	50,228
NGA50	38,012	15,379	38,774	41,082
NA75	22,203	9,638	22,363	24,628
NGA75	13,775	4,295	9,397	12,682
LA50	18,293	37,752	16,078	15,664
LGA50	22,966	53,160	21,504	20,730
LA75	40,533	85,806	36,676	35,027
LGA75	56,888	146,687	59,970	54,169

Table A.9: LASER results for H3 contigs.

Program:	**ABySS**	**SGA**	**SOAPdenovo2**	**SAGE2**
Overlap/*k*-mer Length:	**78**	**82**	**77**	**78**
# contigs (>= 0 bp)	3,433,810	10,083,067	7,143,688	1,831,159
# contigs (>= 1000 bp)	569,265	567,367	673,574	480,482
Total length (>= 0 bp)	3,010,873,530	3,774,427,245	3,376,035,748	2,877,999,967
Total length (>= 1000 bp)	2,498,815,487	2,477,272,963	2,406,225,742	2,525,918,443
# contigs	744,694	741,582	935,582	648,603
Largest contig	70,592	74,646	61,855	97,109
Total length	2,625,264,380	2,602,300,170	2,595,036,781	2,645,240,454
Reference length	3,209,286,105	3,209,286,105	3,209,286,105	3,209,286,105
GC (%)	40.67	40.62	40.62	40.65
Reference GC (%)	40.99	40.99	40.99	40.99
N50	6,015	5,883	4,399	7,615
NG50	4,543	4,413	3,268	5,825
N75	3,006	2,992	2,226	3,759
NG75	1,351	1,278	998	1,624
L50	123,913	124,444	164,174	98,556
LG50	179,825	184,070	245,295	140,906
L75	278,190	279,424	371,830	221,722
LG75	490,574	505,349	674,314	387,905
# misassemblies	1,646	2,028	1,862	1,675
# misassembled contigs	1,616	1,964	1,834	1,574
Misassembled contigs length	7,024,951	8,980,285	5,411,377	8,462,601
# local misassemblies	3,215	3,150	2,950	6,245
# unaligned contigs	4455+3832part	5299+7391part	4705+15059part	6737+17189part
Unaligned length	6,644,111	7,557,260	6,483,978	9,026,923
Genome fraction (%)	81.17	80.46	80.09	81.33
Duplication ratio	1.01	1.01	1.01	1.01
# N's per 100 kbp	0.00	0.00	0.00	0.00
# mismatches per 100 kbp	91.30	88.96	87.39	95.22
# indels per 100 kbp	17.24	12.71	14.90	20.62
Largest alignment	70,592	74,646	61,847	97,109
NA50	6,003	5,867	4,391	7,592
NGA50	4,532	4,399	3,260	5,804
NA75	2,994	2,977	2,216	3,737
NGA75	1,335	1,258	983	1,594
LA50	124,125	124,733	164,428	98,801
LGA50	180,161	184,536	245,740	141,288
LA75	278,827	280,280	372,722	222,471
LGA75	492,702	508,248	677,770	390,406

Table A.10: LASER results for H3 scaffolds.

Program:	**ABySS**	**SGA**	**SOAPdenovo2**	**SAGE2**
Overlap/*k*-mer Length:	**78**	**82**	**77**	**78**
# contigs (>= 0 bp)	1,316,103	588,861	2,041,534	1,046,111
# contigs (>= 1000 bp)	128,518	237,695	120,610	116,209
Total length (>= 0 bp)	2,983,450,141	2,753,129,326	3,041,613,982	2,913,957,318
Total length (>= 1000 bp)	2,817,601,458	2,640,226,727	2,723,536,763	2,709,056,387
# contigs	151,308	284,654	162,994	186,641
Largest contig	560,030	277,348	540,433	513,158
Total length	2,833,989,654	2,673,491,079	2,753,368,917	2,758,031,648
Reference length	3,209,286,105	3,209,286,105	3,209,286,105	3,209,286,105
GC (%)	40.82	40.65	40.70	40.76
Reference GC (%)	40.99	40.99	40.99	40.99
N50	50,197	20,438	54,262	54,990
NG50	42,755	15,828	44,247	44,869
N75	24,676	9,960	26,334	26,554
NG75	15,518	4,570	13,995	13,901
L50	16,299	36,695	14,423	14,279
LG50	20,344	51,606	19,080	18,828
L75	36,259	83,349	32,472	32,154
LG75	50,529	141,336	49,978	49,388
# misassemblies	17,897	13,488	104,593	24,629
# misassembled contigs	11,876	9,459	29,928	12,747
Misassembled contigs length	290,769,241	108,494,521	894,303,559	265,221,745
# local misassemblies	66,350	72,453	188,601	152,169
# unaligned contigs	5669+3571part	6689+5162part	8995+20294part	7751+5453part
Unaligned length	14,919,882	13,954,741	63,020,257	17,527,084
Genome fraction (%)	85.30	82.43	80.67	84.22
Duplication ratio	1.03	1.01	1.04	1.01
# N's per 100 kbp	356.61	497.15	2,473.54	100.37
# mismatches per 100 kbp	103.04	98.07	103.53	107.26
# indels per 100 kbp	29.85	25.94	59.68	31.75
Largest alignment	559,292	277,134	535,906	513,027
NA50	48,209	19,995	46,246	52,796
NGA50	40,955	15,440	37,070	43,230
NA75	23,410	9,641	20,450	25,408
NGA75	14,603	4,306	8,355	13,109
LA50	16,981	37,372	16,592	14,828
LGA50	21,207	52,640	22,098	19,556
LA75	37,893	85,293	38,670	33,453
LGA75	52,968	145,864	63,562	51,585

Table A.11: LASER results for H4 contigs.

Program:	**ABySS**	**SGA**	**SOAPdenovo2**	**SAGE2**
Overlap/*k*-mer Length:	**95**	**105**	**95**	**85**
# contigs (>= 0 bp)	2,090,332	5,947,202	5,295,504	1,000,590
# contigs (>= 1000 bp)	345,728	465,340	542,293	281,249
Total length (>= 0 bp)	2,985,812,932	3,743,802,253	3,389,518,869	2,878,436,768
Total length (>= 1000 bp)	2,648,848,342	2,620,519,797	2,529,980,097	2,662,902,138
# contigs	430,157	614,197	732,452	359,302
Largest contig	134,504	172,299	129,492	203,677
Total length	2,708,111,910	2,724,191,459	2,665,648,222	2,717,533,904
Reference length	3,209,286,105	3,209,286,105	3,209,286,105	3,209,286,105
GC (%)	40.82	40.78	40.77	40.79
Reference GC (%)	40.99	40.99	40.99	40.99
N50	12,473	8,353	6,575	16,332
NG50	9,965	6,652	4,978	13,091
N75	6,268	4,100	3,117	8,132
NG75	3,214	2,184	1,479	4,165
L50	62,286	89,328	107,523	47,141
LG50	84,772	121,872	155,107	63,965
L75	138,527	205,704	255,106	105,865
LG75	220,557	325,364	441,756	167,998
# misassemblies	2,126	2,653	2,328	2,125
# misassembled contigs	1,968	2,413	2,187	1,859
Misassembled contigs length	16,003,022	15,748,128	10,182,145	21,927,574
# local misassemblies	4,033	4,070	3,524	7,324
# unaligned contigs	5086+3360part	7443+4943part	5693+8789part	5086+4891part
Unaligned length	7,871,276	10,274,608	7,904,530	9,058,906
Genome fraction (%)	83.76	83.96	82.31	83.62
Duplication ratio	1.01	1.01	1.01	1.01
# N's per 100 kbp	0.00	0.00	0.00	0.00
# mismatches per 100 kbp	96.28	95.40	86.95	98.67
# indels per 100 kbp	21.79	15.94	17.70	24.68
Largest alignment	134,502	172,299	129,486	203,677
NA50	12,436	8,324	6,559	16,254
NGA50	9,932	6,629	4,964	13,025
NA75	6,238	4,078	3,102	8,084
NGA75	3,179	2,156	1,458	4,109
LA50	62,463	89,602	107,756	47,341
LGA50	85,019	122,261	155,463	64,249
LA75	138,979	206,481	255,817	106,388
LGA75	221,572	327,199	443,981	169,101

Table A.12: LASER results for H4 scaffolds.

Program:	**ABySS**	**SGA**	**SOAPdenovo2**	**SAGE2**
Overlap/*k*-mer Length:	**95**	**105**	**95**	**85**
# contigs (>= 0 bp)	1,105,427	1,530,395	2,940,191	1,219,875
# contigs (>= 1000 bp)	123,211	174,783	101,206	123,678
Total length (>= 0 bp)	2,986,522,925	3,120,864,494	3,248,261,172	3,130,442,764
Total length (>= 1000 bp)	2,816,772,758	2,722,727,585	2,781,574,681	2,723,610,005
# contigs	148,566	230,523	145,563	357,221
Largest contig	450,136	328,004	647,153	511,459
Total length	2,834,716,067	2,759,831,626	2,812,117,028	2,876,483,297
Reference length	3,209,286,105	3,209,286,105	3,209,286,105	3,209,286,105
GC (%)	40.85	40.79	40.81	40.85
Reference GC (%)	40.99	40.99	40.99	40.99
N50	51,478	30,587	75,171	47,515
NG50	43,934	24,917	62,773	40,836
N75	25,416	14,938	35,750	21,404
NG75	16,233	8,297	21,026	13,120
L50	15,771	25,163	10,699	16,811
LG50	19,713	33,314	13,592	20,589
L75	35,206	57,375	24,183	39,145
LG75	48,901	87,218	34,903	53,906
# misassemblies	11,957	9,994	94,817	45,534
# misassembled contigs	8,253	6,744	26,754	27,055
Misassembled contigs length	174,189,232	120,646,106	1,011,516,226	163,393,558
# local misassemblies	35,234	192,948	249,213	154,550
# unaligned contigs	4907+2888part	7386+5466part	9985+17478part	30572+24164part
Unaligned length	12,415,493	16,035,711	56,287,399	37,064,037
Genome fraction (%)	85.84	84.48	81.88	85.05
Duplication ratio	1.03	1.01	1.05	1.04
# N's per 100 kbp	130.44	413.98	2,568.61	149.75
# mismatches per 100 kbp	103.99	99.95	104.66	107.81
# indels per 100 kbp	26.32	29.93	44.40	27.53
Largest alignment	449,930	328,004	643,390	511,425
NA50	50,116	29,844	62,031	46,361
NGA50	42,858	24,249	51,178	39,803
NA75	24,732	14,405	27,042	20,792
NGA75	15,701	7,772	13,226	12,699
LA50	16,177	25,658	12,571	17,250
LGA50	20,218	34,014	16,096	21,124
LA75	36,136	58,839	29,549	40,164
LGA75	50,247	90,172	44,879	55,374

Table A.13: LASER results for H5 contigs.

Program:	**ABySS**	**SGA**	**SOAPdenovo2**	**SAGE2**
Overlap/*k*-mer Length:	**90**	**105**	**90**	**85**
# contigs (>= 0 bp)	2,389,267	6,619,070	5,920,123	1,752,129
# contigs (>= 1000 bp)	365,963	474,155	554,370	271,826
Total length (>= 0 bp)	3,013,395,723	3,870,912,558	3,445,435,878	3,066,749,378
Total length (>= 1000 bp)	2,645,790,974	2,631,647,246	2,528,901,821	2,618,000,286
# contigs	456,508	631,181	748,510	397,597
Largest contig	140,640	115,971	116,964	194,702
Total length	2,709,602,256	2,740,913,281	2,667,422,165	2,705,466,683
Reference length	3,209,286,105	3,209,286,105	3,209,286,105	3,209,286,105
GC (%)	40.81	40.77	40.76	40.79
Reference GC (%)	40.99	40.99	40.99	40.99
N50	11,652	8,198	6,328	16,141
NG50	9,281	6,592	4,828	12,816
N75	5,805	4,017	3,039	7,935
NG75	2,981	2,206	1,462	3,820
L50	66,000	91,775	112,497	47,251
LG50	90,060	123,682	161,555	64,779
L75	148,116	211,108	264,792	106,746
LG75	236,494	327,510	454,171	173,471
# misassemblies	2,032	2,689	2,353	2,215
# misassembled contigs	1,911	2,457	2,238	1,918
Misassembled contigs length	13,788,203	15,743,727	9,444,381	21,016,146
# local misassemblies	3,991	4,157	3,474	6,295
# unaligned contigs	5214+3092part	7984+5032part	5876+8367part	5905+5459part
Unaligned length	7,770,535	10,736,211	7,918,912	8,156,392
Genome fraction (%)	83.80	84.44	82.37	82.95
Duplication ratio	1.01	1.01	1.01	1.01
# N's per 100 kbp	0.00	0.00	0.00	0.00
# mismatches per 100 kbp	95.89	95.06	86.30	95.71
# indels per 100 kbp	21.91	15.76	17.13	24.67
Largest alignment	140,640	115,971	116,964	194,702
NA50	11,624	8,170	6,313	16,070
NGA50	9,258	6,568	4,815	12,756
NA75	5,782	3,994	3,026	7,890
NGA75	2,951	2,179	1,444	3,766
LA50	66,154	92,062	112,729	47,451
LGA50	90,276	124,080	161,913	65,055
LA75	148,525	211,912	265,520	107,237
LGA75	237,463	329,331	456,365	174,553

Table A.14: LASER results for H5 scaffolds.

Program:	**ABySS**	**SGA**	**SOAPdenovo2**	**SAGE2**
Overlap/*k*-mer Length:	**90**	**105**	**90**	**85**
# contigs (>= 0 bp)	1,251,442	2,010,126	2,791,956	1,883,890
# contigs (>= 1000 bp)	121,232	297,873	99,014	99,279
Total length (>= 0 bp)	3,009,785,847	3,184,095,416	3,240,808,403	3,289,299,138
Total length (>= 1000 bp)	2,827,746,712	2,636,743,305	2,793,843,276	2,660,520,753
# contigs	146,669	442,000	142,958	421,934
Largest contig	506,699	243,456	597,474	558,430
Total length	2,845,815,092	2,735,779,854	2,824,281,366	2,872,290,895
Reference length	3,209,286,105	3,209,286,105	3,209,286,105	3,209,286,105
GC (%)	40.84	40.78	40.81	40.85
Reference GC (%)	40.99	40.99	40.99	40.99
N50	58,074	16,353	79,889	56,539
NG50	49,060	12,850	67,230	48,512
N75	27,048	7,393	37,804	25,168
NG75	16,700	3,424	22,568	14,749
L50	13,868	44,406	10,065	14,231
LG50	17,274	60,745	12,693	17,453
L75	31,739	106,432	22,831	33,014
LG75	44,436	175,147	32,607	45,943
# misassemblies	13,621	3,188	100,925	46,187
# misassembled contigs	9,091	2,684	27,372	27,238
Misassembled contigs length	212,666,717	38,060,420	1,129,083,987	180,324,857
# local misassemblies	33,503	36,605	220,347	141,062
# unaligned contigs	5342+2986part	7385+3804part	10711+19087part	37672+22460part
Unaligned length	12,444,183	10,442,541	62,967,105	38,679,758
Genome fraction (%)	86.12	84.44	82.07	84.33
Duplication ratio	1.03	1.01	1.05	1.05
# N's per 100 kbp	152.81	134.84	2,620.33	169.99
# mismatches per 100 kbp	105.21	95.67	103.81	103.89
# indels per 100 kbp	26.66	21.12	46.61	27.12
Largest alignment	506,508	243,429	544,568	542,673
NA50	56,410	16,238	64,798	54,919
NGA50	47,667	12,745	53,583	47,051
NA75	26,105	7,315	28,079	24,398
NGA75	16,112	3,342	13,980	14,249
LA50	14,301	44,748	12,153	14,654
LGA50	17,812	61,219	15,420	17,972
LA75	32,752	107,326	28,556	34,018
LGA75	45,912	177,145	42,769	47,375

Table A.15: LASER results for H6 contigs.

Program: **Overlap/*k*-mer Length:**	**ABySS** **118**	**SAGE2** **130**
# contigs (>= 0 bp)	1,895,662	2,072,442
# contigs (>= 1000 bp)	356,617	515,217
Total length (>= 0 bp)	3,044,475,902	3,216,215,947
Total length (>= 1000 bp)	2,691,900,342	2,611,345,144
# contigs	448,202	757,933
Largest contig	142,396	119,663
Total length	2,755,904,160	2,776,930,276
Reference length	3,209,286,105	3,209,286,105
GC (%)	40.82	40.73
Reference GC (%)	40.99	40.99
N50	12,101	7,233
NG50	9,958	5,874
N75	6,164	3,382
NG75	3,511	1,915
L50	65,756	103,293
LG50	86,417	136,502
L75	145,168	243,514
LG75	217,199	369,545
# misassemblies	4,181	5,159
# misassembled contigs	3,892	4,821
Misassembled contigs length	23,083,598	17,181,841
# local misassemblies	4,532	5,440
# unaligned contigs	8063+5384part	11844+30453part
Unaligned length	11,756,418	17,394,649
Genome fraction (%)	85.20	83.26
Duplication ratio	1.00	1.03
# N's per 100 kbp	0.00	0.00
# mismatches per 100 kbp	96.69	101.11
# indels per 100 kbp	19.32	20.23
Largest alignment	142,396	119,663
NA50	12,052	7,202
NGA50	9,913	5,844
NA75	6,128	3,350
NGA75	3,469	1,874
LA50	65,979	103,645
LGA50	86,726	137,009
LA75	145,778	244,783
LGA75	218,445	372,658

Curriculum Vitae

Michael Molnar

EDUCATION

University of Western Ontario, London, Ontario, Canada

- Doctor of Philosophy (PhD) in Computer Science — Jan 2013 – Dec 2017
  - Thesis: Genome Assembly from Next Generation DNA Sequencing Data
  - Adviser: Dr. Lucian Ilie
  - Research area: Bioinformatics.
- Master of Science (MSc) in Computer Science — Sep 2011 – Dec 2012
  - Thesis: Error Correction in Next Generation DNA Sequencing Data
  - Adviser: Dr. Lucian Ilie
  - Research area: Bioinformatics.
- Bachelor of Science (BSc) in Bioinformatics (Biochemistry Concentration) — Sep 2001 – Aug 2011
  - Graduated with Honors.

Fanshawe College, London, Ontario, Canada

- Diploma in Business Information Systems — Sep 1997 – Aug 1999

PUBLICATIONS

JOURNALS

[1] Molnar, M., Haghshenas, E., and Ilie, L. (2017). SAGE2: Parallel Human Genome Assembly. *Bioinformatics*, btx648.

[2] Molnar, M., and Ilie, L. (2014). Correcting Illumina data. *Briefings in Bioinformatics*, **16**(4), 588-599.

[3] Ilie, L., Haider, B., Molnar, M., and Solis-Oba, R. (2014). SAGE: string-overlap assembly of genomes. *BMC Bioinformatics*, **15**(1), 302.

[4] Ilie, L., and Molnar, M. (2013). RACER: Rapid and accurate correction of errors in reads. *Bioinformatics*, **29**(19), 2490-2493.

JOURNAL REVIEWS

- Bioinformatics — Nov 2013 – Present
  - Impact factor 7.307.
- Briefings in Bioinformatics — May 2015 – Present
  - Impact factor 5.134.

WORK EXPERIENCE

University of Western Ontario, London, Ontario, Canada

- Guest Lecturer — Feb 2016
  - CS 2124/2125 - Medical Computing.
  - Presentation: Next-Generation DNA Sequencing and De Novo Genome Assembly.
- Guest Speaker — Nov 2015
  - Topical Research In Computer Science Seminar.
  - Presentation: Genome Assembly using DNA Sequencing Data.
- Teaching Assistant — Sep 2013 – Apr 2017
  - Taught and prepared labs.
  - Consulted with students.
  - Graded assignments and exams.
  - Proctored exams.

SCHOLARSHIPS AND GRANTS

- Queen Elizabeth II Graduate Scholarship in Science and Technology — 2015 – 2016
- Queen Elizabeth II Graduate Scholarship in Science and Technology — 2012

PUBLISHED PROGRAMS

- SAGE2: String-overlap Assembly of GEnomes
  - De novo genome assembly from Next-Generation DNA sequencing data.
- RACER: Rapid and Accurate Correction of Errors in Reads
  - Corrects errors in Next-Generation DNA sequencing data.
- kmerSearch
  - Error correction evaluation of k-mers in reads.
- readSearch
  - Error correction evaluation of whole reads.

AWARDS

- UWO Research in Computer Science Conference, University of Western Ontario Apr 2016
  - 2nd place.
- UWO Research in Computer Science Conference, University of Western Ontario Apr 2015
  - 2nd place.
- UWO Research in Computer Science Conference, University of Western Ontario Apr 2013
  - 1st place.
- Dean’s List, Fall 2001 through Spring 2002, University of Western Ontario 2001 – 2002

SKILLS

Unix, Linux, C++, OpenMP, JAVA, Python, Perl, Visual Basic, HTML, LaTeX, MATLAB, R, Microsoft Office, Photoshop, Dreamweaver.

LANGUAGES

- English: Native language.

[*CV compiled on 2017-11-28*]