Analyzing the Effects of Sequencer Discrepancies on Next-Generation Genome Assembly Tools

by

Michael J. Pritchard Jr.

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
August 6, 2016

Keywords: Bioinformatics, Genomics, Genome Assembly

Approved by

Weikuan Yu, Co-chair, Associate Professor of Computer Science and Software Engineering
Saad Biaz, Co-chair, Associate Professor of Computer Science and Software Engineering
Hari Narayanan, John J. and Gail Watson Professor of Computer Science and Software Engineering

Abstract

The advent of Next-Generation Sequencing (NGS) techniques in the early 21st century massively increased genetic sequencing throughput while dramatically reducing associated costs. This is turn lowered barriers of entry sufficient to permit vastly expanded research interests. To handle the resulting explosion of sequencing data being produced, new techniques for assembling genomes, transcriptomes, and proteomes were required. In the last 15 years, numerous tools for each of these assembly categories have arisen, each purporting superiority relative to other tools. In particular, de novo genome assembly has spawned more than 75 tools utilizing different assembly pipelines, error correcting methods, and novel data structures. Previous works have shown that no one tool can lay claim to general supremacy - some are, by design or happenstance, better suited to certain data types (e.g. human, plant, or bacteria genomes). What these works have not done is shown how variations in sequenced libraries affect assembly or explained why these effects occur. The goal of this work therefore is to analyze these effects. Execution of this goal is split into two primary parts: an in-depth architectural analysis of several popular de novo genome assemblers including expected behavioral changes across sequencer variations, and evaluations of these tools using data sets permuted over a range of coverage depths, read lengths, and read types. The focus of this work is to assess the flexibility of several popular de novo genome assemblers (which can grouped as either utilizing de Bruijn graphs or a hybridized approach for their assembly) with respect to sequencer variations over a single genome. The results of the evaluations revealed a startlingly high sensitivity to variation in the de Bruijn based assemblers even with libraries that would, at first glance, appear far better suited to assembly. Though error detection and correction methodologies worked exceptionally for both de Bruijn assemblers, the maximum contig length and other important metrics degraded rapidly as library coverage

increased. As expected, the hybrid de Bruijn/String graph approach was not as vulnerable to these same variations, but had its own shortcomings. The minimum threshold of coverage for reasonable assembly was higher than the pure de Bruijn approaches; additionally, the incidence of misassembled contigs was much higher. The analysis performed in this work provides useful and practical insights into the behaviors of genome assemblers which can both ease assembly tuning and expedite the process of choosing appropriate data sets for future research.

Acknowledgments

I would like to thank Dr. Weikuan Yu, my research advisor and committee co-chair, first and foremost for welcoming me into his research group. His willingness to expand his existing research into the realm of bioinformatics gave me an unprecedented opportunity to learn and grow as a student. Aside from providing the opportunity for me to delve into the fields of both bioinformatics and high-performance computing systems, Dr. Yu's tutelage improved my technical writing, presentation, and research skills. I am honored to have been able to work with you.

I would also like to thank the other members of my committee, Dr. Saad Biaz (co-chair) and Dr. Hari Narayanan, for their support and guidance. Outside the context of my committee, I have worked with Dr. Narayanan as a teaching assistant for his introduction and advanced algorithms courses – an opportunity I am pleased to have been able to pursue. Working with Dr. Narayanan and aiding my fellow students in this capacity has both further developed my own understanding of the material and improved my ability to communicate ideas effectively.

Much of my tenure as a graduate student at Auburn University was spent as a member of the Parallel Architecture and Systems Laboratory (PASL), and so I'd like to thank in no particular order my fellow lab-mates: Kevin Vasko, Hai Pham, Xinning Wang, Dr. Bin Wang, Dr. Zhuo Liu, Dr. Cong Xu, Dr. Hui Chen, Dr. Jianhui Yue, Huansong Fu, Lizhen Shi, Fang Zhou, Teng Wang, Yue Zhu, and Hao Zhou. The PASL group was enormously helpful and constantly piqued my research interest in a variety of topics. Working with all of you was a pleasure and an honor, and I wish each and every one of you all the best.

I'm immensely grateful for my family members, without whose endless support, love, and faith in me I would not be where I am today: Carol Pritchard (mother), Michael Pritchard

(father), Grace Pritchard (sister), Laurel Novack (sister), Scott Novack (brother-in-law), and Isabelle Novack (niece). You are all anyone could ask for in a family, and I love you all.

Finally, I'd like to express my gratitude to my friends, new and old. Each of you has helped shape the person I am today, and I'm glad to have met you all. You're the best.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The advent of Next-Generation Sequencing (NGS) techniques in the early 21st century completely revolutionized the landscape of genomic, transcriptomic, and proteomic research. The combination of drastically reduced per-base sequencing costs coupled with huge throughput increases dramatically lowered financial barriers-of-entry for researchers, resulting in an explosive growth of both quantity and variety of sequenced genomes. At the outset of the Human Genome Project in 1990, the estimated cost of to sequence the entire 3 billion base pair human genome was a staggering three billion dollars [9]. In addition, it was expected that fifteen years would be necessary for the sequencing itself to be carried out. By comparison, current generation solutions can sequence an entire human genome for approximately one thousand dollars in under three days [3].

Classically, genetic sequencing was performed by the chain-termination method, popularly known as Sanger sequencing. The method was developed independently by Frederick Sanger[12][13] and Walter Gilbert[6] in 1977, and would prove to be a cornerstone of genetic sequencing for decades to come. Sanger sequencing was first automated in 1986, resulting in a huge throughput increase and representing a milestone on the path towards next-generation sequencing. Though a slow and expensive process (relative to next-generation sequencing), modern Sanger sequencing produces relatively long sequences (in the range of 300-1000 base pairs in length on average) with a high degree of accuracy. These qualities make Sanger sequencing appealing for individuals working with very small data sets, or for secondary verification of particular next-generation assemblies.

Currently, there are four primary second-generation sequencing platforms in use: Roche 454, SOLiD, Illumina, and Ion Torrent. Roche 454, the first successful next generation

system from a commercial standpoint, was released in 2005 and had a read length of 100-150 base pairs and a throughput of 20 Mb per run [21]. Current Roche 454 models provide reads up to 1000 base pairs in length with approximately 700 Mb per run [2]. Sequencing by Oligo Ligation Detection, or SOLiD, had a 2006 release with initial specifications of 35 bp reads and 3 Gb per run. The latest offering from SOLiD, the SOLiD 4 System, generates 80-100 Gb of data per run at 35-50 bp read lengths [1]. Solexa, which was purchased by Illumina in 2007, originally released its Genome Analyzer (GA) in 2006. At the time, it produced 1 Gb per run with length 75 reads [19]. The latest offering from Illumina dwarfs these figures with 1500 Gb per run and length 150 reads [3]. Finally, there is Ion Torrent. Released in 2010, the initial offering yielded 200 bp reads and emphasized its low execution time of two hours [19]. Ion S5, the latest revision of the platform, can produce up to 15 Gb per run with 200 bp reads or 8 Gb per run with 400 bp reads [4].

This work in particular examines the effects of varying read coverage, read length, and read type (e.g. single, paired) on three de novo genome assemblers: SOAPdenovo, Velvet, and StriDe. The research presented in this work differentiates itself predominantly by focusing on broadly permuting sequencer attributes over a single reference genome – previous efforts at analyzing the quality of assembly tools provide one or more representative genomes (e.g. human, plant, bacteria) with either fixed sequencer attributes or a very limited selection of sequencer variations. In addition, a detailed architectural analysis of the tools and how the variations affect the structure of produced graphs set it apart from previous works. The data sets used were generated *in silico* with ART, a suite of sequencer read simulator tools. A portion of the reference genome for *Caenorhabditis elegans* was the basis for the simulated reads. Finally, the assembled reads for each permuted data set were analyzed against the reference using the Quality Assessment Tool for Genome Assemblies (QUAST).

The remainder of this work proceeds as follows: Chapter 2, background and motivation, will provide a primer on genetic sequencing concepts and terminology and an overview of key techniques utilized by next-generation sequence assemblers. In addition, a high-level

description of the tools used in the work will be presented. An exploration of previously published efforts at analyzing assembler quality will be presented in Chapter 3. Chapter 4 characterizes and compares the architectures of the tools utilized in the work and details the expected behavior of the tools as variations are introduced. An evaluation of the previously mentioned assemblers over applicable data sets will be discussed in Chapter 5. Finally, Chapter 6 will contain concluding remarks and suggestions for future work.

Chapter 2

Background and Motivation

This chapter has two particular aims: providing a more detailed coverage of the material mentioned in the Chapter 1, and providing the overall motivation for the work. As bioinformatics is a topic encompassing multiple disciplines, primers on key concepts will be presented with the hope of resolving unfamiliar aspects. Section 2.2 will provide a detailed introduction to genetic sequencing concepts and terminology from a computational perspective. Section 2.3 provides details on two popular construction techniques critical to modern assembly methods: Overlap Layout Consensus and de Bruijn graphs. Finally, a brief introduction and high-level description of the tools utilized in this work will be given in section 2.4. Further information on each of these topics can be acquired by perusing referenced works.

2.1 Sequencing as a Computational Problem

Before delving into the particulars of sequencing platforms and assemblers, it is useful to frame the overall problem of genome assembly (in particular, genome assembly via shotgun sequencing) as a computational problem. Each strand in the double-helix structure of DNA is composed of nucleotides. In particular, those nucleotides contain one of four bases: adenine, thymine, guanine, or cytosine. Each base from one strand bonds with one base from the other, forming a base pair. These pairs obey base pairing rules – adenine only bonds with thymine, and cytosine only bonds with guanine. We can simplify the visualization of these bonds by imagining each strand to be a string of characters, the alphabet of which is size four (A, T, G, and C). The act of sequencing a strand of DNA is doing just that – examining the molecular structure of each nucleotide and resolving it to a character in a string. Having

established the correlation between genetic (DNA) sequences and strings, several terms which will be used frequently through the course of this work can be given definitions relative to strings and characters:

- A **sequence** or **read** (the terms will be used interchangeably) is a string comprised of the alphabet {A, T, G, C}
- A **base** is a character in a string
- A **base pair** is a character and its complement, each occupying the same position in their respective sequences

The concept of coverage, which is of some importance to this work, is a numerical value denoting the average representation of each base. It can be calculated by multiplying the total number of reads by the ratio of average read length to genome length.

2.2 Next-Generation Sequencing

Second-generation sequencing techniques (the first generation being Sanger sequencing) operate on the principle of **shotgun sequencing**. This methodology, visualized in Figure 2.1, is predicated upon generating numerous random sequences of relatively small size. The size of the sequenced fragments is restricted by underlying chemical and technological limitations. One important aspect of shotgun sequencing not explicitly represented in Figure 2.1 is amplification, the process by which DNA is copied. Restructuring the original sequence without amplification would be impossible, as randomly fragmentation does not preserve any ordering information. By producing numerous copies of the original sequence and randomly fragmenting each, we produce sequences which overlap with one another. This overlap serves as the basis for sequence reconstruction.

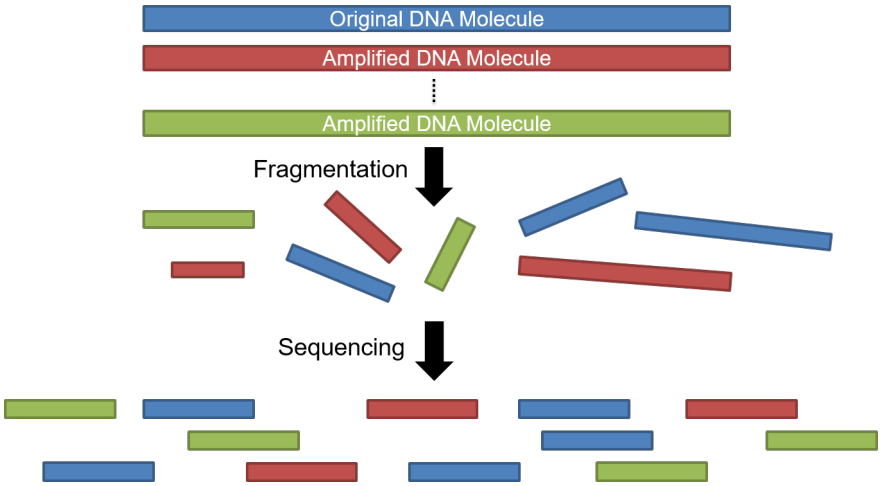

Figure 2.1: Example of shotgun sequencing [17]

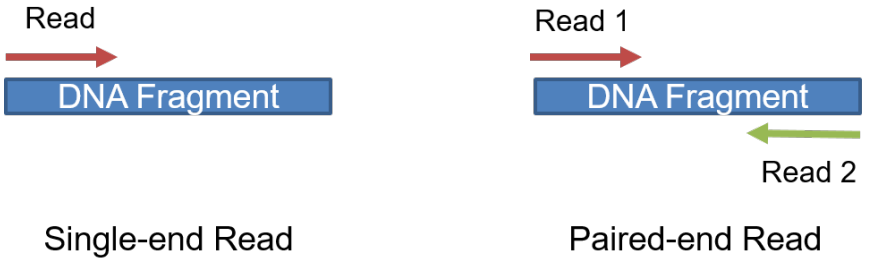

Figure 2.2: Illustration of Single-end vs. Paired-end Read [5]

2.2.1 Read Types

Two varieties of reads are utilized in this work: single reads and paired-end reads. The former, as the name implies, sequences each fragment only once. Paired-end sequencing, however, sequences each fragment from both ends. Since the total length of the fragment is known, we know the exact distance between the two reads [5]. This is turn improves the chances of correctly assembling the original sequence. Figure 2.2 illustrates the structure of a paired-end read with respect to the reference.

2.3 Genome Assembly Techniques

2.3.1 Overlap Layout Consensus

Prior to the advent of high-throughput short-read sequencers, Overlap Layout Consensus (OLC) was the primary technique utilized by assemblers [23]. With OLC, each read is represented as a node, with directed edges between nodes indicating pairwise alignment. An ordered Hamiltonian cycle of the graph produces a candidate genome. The limitation of this method lies with the inherent complexity of finding a Hamiltonian Cycle – an NP-hard problem. As such, the number of reads constituting the genome must be small (else the problem becomes computationally intractable). This is turn limits application of this technique to either long reads or short genomes.

The string graph variant of OLC simplifies the graph by removing transitive edges. Further reductions in construction cost have arisen by utilizing the Ferragina-Manzini Index to efficiently compute the overlaps necessary for graph construction [27].

2.3.2 de Bruijn Graphs

The de Bruijn graph, originally proposed in 1946 [8], has largely come to replace previous overlap-consensus based construction methods. This is due in large part to the computational complexity involved in generated contiguous sequences with overlap-consensus techniques. De Bruijn graphs were first utilized with regard to assembly by the EULER assembler in 2001 [24].

The construction of a de Bruijn graph can be summarized as follows:

- Consider a set of strings, each of length n. For every unique substring of length n - 1, create a vertex and label it as the substring.
- For each pair of vertices V1 and V2, add a directed edge from V1 to V2 if the last n - 2 characters in V1 (its *prefix*) correspond to the first n - 2 characters in V2 (its *prefix*).

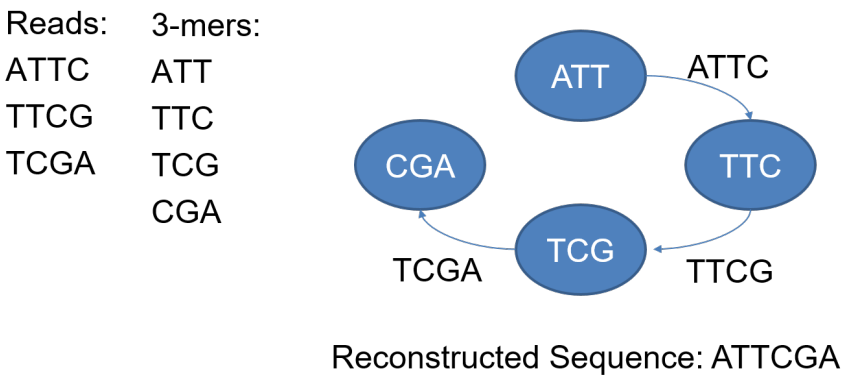

Figure 2.3: Example construction of a de Bruijn graph [10]

- Label each edge with the length n string formed by joining the prefix of V1 with the last character in V2.

Figure 2.3 provides a visual example of a de Bruijn graph (n = 4) comprised of binary numbers. In the context of sequence assembly, de Bruijn graphs are especially useful because they can repeat k-mers (of which there are typically many) without inflating the size of the graph.

2.4 Tools Utilized

In this section a brief introduction to the various tools used as part of this work will be provided. Aside from the three assemblers evaluated (SOAP, Velvet, and StriDe), ART was utilized to produce /textitin silico reads based on a reference genome, and QUAST generated evaluation metrics based on contig output from the assemblers.

2.4.1 ART

ART, the sequencing read simulator used in this work, is an actively maintained suite of tools originally published in 2012 [15]. Chief among the features of ART is its ability to accurately simulate sequences based on Roche 454, SOLiD, or Illumina sequencers. For each of these platforms, ART has built-in empirically determined error profiles capable of

reproducing both substitution and indel errors as appropriate. Additionally, custom error profiles or scaled error profiles (e.g. 1/10th or 2x standard) can be specified. For the read lengths used in this work (75 bp and 150 bp), the Illumina Genome Analyzer II and Illumina HiSeq 2500, respectively, were used as the error profiles.

2.4.2 SOAPdenovo

Short oligonucleotide alignment program (de novo), or SOAPdenovo [25] (SOAPdenovo2 [20] to be more precise) is the first of the assembly tools being analyzed in this work. Released in 2010, SOAPdenovo was a *de novo*) variant of the original SOAP [18], a strictly *ab initio* assembler. SOAPdenovo contains a full assembly pipeline based on de Bruijn graph construction.

Prior to graph construction, SOAPdenovo analyzes the read libraries to prune erroneous reads. This has the side effect of drastically reducing the number of distinct k-mers generated from the library, thus saving memory. For the human genome used in the original SOAPdenovo paper, the number of distinct k-mers generated was reduced from 14.6 million to 5 million – a reduction of almost 3 times [25].

After the initial de Bruijn graph has been constructed, SOAPdenovo performs four correcting techniques: short tip clipping, low-coverage removal, tiny repeat resolution, and bubble merging. Tips shorter than 50 bp are removed, as are low-coverage nodes that appear only once. Read path information is used to resolve small repeats, and low-difference bubbles were merged by analyzing depth information at the bubble site. From this corrected de Bruijn graph contigs are generated and those of length greater than 100 are reported.

The last two phases in the assembly pipeline for SOAPdenovo are scaffolding and gap closure. Scaffolding is achieved by aligning the original paired-end reads with the set of contig sequences. Gaps are closed by taking paired reads that match a contig well at one end are in the insert region on the other. Typically these reads would have been marked as duplicates and masked prior to scaffold construction.

2.4.3 Velvet

Released in 2008, Velvet was one of the first de Bruijn-based assemblers designed to handle very short reads [28]. Velvet's pipeline begins by constructing a standard de Bruijn graph (see Chapter 2 for more information), then simplifies the graph by iteratively collapsing "blocks." A block consists of two nodes, with one node's only outgoing edge corresponding to the other's only incoming edge. Following simplification, Velvet undergoes a graph correction phase very similar to SOAP's: tips, bubbles (referred to "bulges" in the original source), and erroneous connections. Tips shorter than twice the k-mer value are cut, while bubbles are collapsed utilizing the Tour Bus algorithm. Tour Bus detects redundant paths by executing a breadth-first search from an arbitrary start node. A distance metric based on arc multiplicity gives high coverage paths priority. When a previously-visited node is discovered along a path, a common ancestor is found between the convergent paths, with the lower multiplicity path being merged. Erroneous connections are removed based on a coverage cutoff value (typically set by the user).

2.4.4 StriDe

StriDe, an integrated string and de Bruijn-based assembler, is the newest of the assemblers being evaluated in this work (it was released in January of this year). The main concept behind the StriDe assembler is the identification of two distinct types of read regions: repetitive and error-prone [16]. In repetitive regions, a walk of feasible extensions to paired-end reads is performed. The result of this walk transforms the paired-end read into a long read. Paired-end reads which cannot be extended are considered to be an error-prone regions. In these regions, the reads are decomposed into smaller, overlapping sub-reads at potential breakpoints (i.e. those points at which an error may exist). This ensures flexibility when the final assembly is performed. The extended and decomposed reads are combined into a string graph, and error correction procedures are performed.

2.4.5 QUAST

The QUality ASessment Tool for Genome Assemblies, QUAST [14], is a fully featured assembly evaluation suite. Released in 2013, QUAST was designed to provide a comprehensive selection of metrics with maximum ease-of-use. One major advantage to QUAST over other assessment tools (e.g. GAGE, Assemblathon) is that QUAST can evaluate assemblies without the assistance of a reference genome. Additionally, multiple assemblies can be few to QUAST simultaneously to provide competitive analysis – it can even automatically generate plots. More than 30 metrics are generated in the presence of a reference genome, eight of which were included as evaluation criteria in this work.

Chapter 3

Related Work

In this chapter two pre-existing works related to genome assembler quality analysis, GAGE and Assemblathon, will be discussed. Assemblathon 1 was published in 2011, while GAGE was published in 2012. Each has since released a follow-up work. It is typical for creators of new assemblers to provide some manner of evaluation against existing tools. Though no malice is assumed on the part of creator, several biasing factors have to be considered. First and foremost, there is an inherent imbalance in relative expertise: the tool's creator has an inherent advantage with regard to parameter tuning for his/her assembler. The same intimate level of familiarity will not be present when determining which combination of parameters to utilize with competing assemblers. In addition, assemblers are not all designed with an agnostic attitude toward the type of data being assembled. These biasing factors, along with the explosive growth of assembler variety, prompted the publication of the works discussed in this chapter.

This research is differentiated from previous efforts in several notable ways. First, the focus of this work is to assess the relative flexibility of assemblers with regard data set variations arising from the sequencers – overall coverage, sequencer read length, and single/paired end reads. Previous works have presented several differing data sets with fixed sequencer attributes per data set or a very small number of sequencer variations. Furthermore, the coverage variations used in this work (10x to 70x) are well below the 60x to 300x coverages provided previously. As a final note utilizes the most up-to-date variants of the representative assemblers and includes an assembler absent from previous efforts.

3.1 Assemblathon

Assemblathon 1, a competitive assessment of de novo short read assembly methods, was published in 2011 by Earl et al.[11] The team responsible for the competition generated a novel genome based partially on human chromosome 13 and then asked teams to assemble the genome blind. Competitors were given just over a month to submit their entries. As an additional metric, several assemblers were used with default parameters to evaluate "naive" executions. In total, 17 assemblers with 41 unique submissions were gathered.

Assemblathon 2, which took place in 2013, maintained the same competitive spirit of its predecessor, but modified the underlying principles of the competition by providing three real data sets (fish, snake, and bird) which did not have high quality reference genomes [7]. As a result, alternative metrics were employed to evaluate the quality of the submitted assemblies. Four months were allotted to teams this time around, and intermediate "evaluation" assemblers were allowed. 43 unique assemblies from 21 participating teams were evaluated, with the overall conclusion being that there was still significant room for improvement in the field.

3.2 GAGE

The Genome Assembly Gold-Standard Evaluations, or GAGE, was published in 2011 by Salzberg et al.[26] Eight genome assemblers considered to be among the best were chosen for evaluation against four distinct data sets. GAGE was conducted using in "in-house" approach whereby a team of individuals possessed of expertise working with genome assemblers personally tuned each tool for optimal results. Two bacteria (*S. aureus* and *R. sphaeroides*), human chromosome 14, and *Bombus impatiens*, the common bumblebee, were chosen as the representative data sets. These sets ranged from three million base pairs to 250 million base pairs, and were all actual Illumina-sequenced data (not simulated data). With the exception

of *Bombus impatiens*, each data set possessed a high-quality reference genome from which evaluations could be based.

The eight assemblers chosen were ABySS, ALLPATHS-LG, Bambus2, CABOG, MSR-CA, SGA, SOAPdenovo, and Velvet. The full details concerning the parameter tuning of these assemblers were released as supplemental material to the original publication. Though GAGE determined that ALLPATHS-LG delivered the most consistent performance with the best trade-off between size and error rate, concluding remarks for the work stressed the importance of continued evaluation of assemblers.

Chapter 4

Characterization and Analysis of Evaluated NGS Tools

This chapter contains a characterization of the assemblers evaluated in the work. Among the factors characterized are ease-of-use, sequencer platform compatibility, architectural differentiations, and expected behaviors as parameters and sequencer attributes vary. Given the popularity of de Bruijn graph-based assemblers, particular attention is given to the underlying effects library variations have on the construction and correction of de Bruijn graphs.

4.1 Ease of Use

Given the myriad assembly tools available for use, intangible characterizations are worth considering. Factors such as documentation quality, ease of compilation/installation, broad sequencer platform compatibility, and the presence or absence of convenient features can play a non-trivial role in the selection process. In this section, each of the assemblers will be analyzed with respect to these features.

4.1.1 SOAP

Though SOAP is available as a pre-compiled binary, the source code can also be downloaded and compiled manually. It is designed for use with 64-bit Linux, and has memory requirements ranging from 5 GB to 150 GB depending on the size of the input genome.

Among the three assemblers, SOAP is unique in requiring a configuration file. This file specifies the location of the read libraries to be assembled and contains some basic information on the libraries (such as maximum read length, average insert size, and whether the sequence should be reversed). Input files can be either FASTA or FASTQ format.

Documentation, including descriptions of input parameters and examples of configuration files, are available on the main website for SOAP. Though there are discrepancies among the parameters listed on the site and the actual parameters included in the latest versions of the program, the most important and likely to be used parameters are covered. A full description of the output files generated by the pipeline and a brief FAQ are also included.

The SOAP pipeline contains four primary phases (pregraph, contig, map, and scaffold); however, a single command suffices to execute the full pipe. Fairly extensive intermediate outputs are generated as the pipeline executes, giving users insight into the error correction processes and providing significant quantities of useful data aside from generated metrics.

4.1.2 Velvet

Like SOAP, Velvet primarily designed for use with 64-bit Linux operating systems, though some support for Mac OSX and Solaris is also provided. Unlike SOAP, there is no pre-compiled binary avaiable (as there are several compiler options which the user may wish to modify).

The author of Velvet provides an extensive manual. Contained within the manual are both "quick-start" instructions and a far more comprehensive set of instructions for those wishing to tune the assembler. The latter contains compilation instructions (with details on the various compilation settings) and running instructions filled with example commands and parameter details. Additionally, there is a short FAQ for considerations such as k-mer size and coverage cutoff size.

Of the three assemblers, Velvet is the only one which requires more than a single command to execute the entire pipeline. In particular, there are two commands which must be executed: velveth and velvetg. The velveth command takes care of hashing the read k-mers and producing some intermediate output files necessary for the execution of velvetg. It supports a wide variety of standard input formats such as FASTA, FASTQ, SAM, and BAM. The velvetg command generates the de Bruijn graph and handles the remainder of the

pipeline (error correction, contig generation, etc). The only required input is the directory containing the data generated by velveth, though there are a number of parameters which can be optionally modified.

4.1.3 StriDe

Also supporting 64-bit Linux, StriDe provides both public source code and a pre-compiled binary for download. Compilation does not require specification of any additional parameters. StriDe supports both FASTA and FASTQ formats, but requires paired-end reads for execution. This is in sharp contrast to SOAP and Velvet, as they support both single-end and paired-end reads.

In terms of provided documentation, StriDe is the sparsest of the three. A few basic examples of command execution are provided, but no comprehensive list of parameters (optional or mandatory) is provided. Parameter descriptions can be gleaned from the command line by specifying a particular stage in the pipeline and appending "–help."

The seven stages in the StriDe pipeline can be executed individually or as part of a single command. Depending on the data contained within the read libraries, execution can be achieved without specifying any additional parameters beyond the input files.

4.2 Architectural Differences

This section will break down the primary architectural differences amongst the three evaluated assemblers. In particular, two approaches will be considered: the practical differences between the de Bruijn-based assemblers and StriDe's hybrid approach, and the differences between the error-correcting methods of SOAP and Velvet.

4.2.1 de Bruijn vs. Hybrid

From a high-level perspective, StriDe is predominantly differentiated from SOAP and Velvet through its use of a string (overlap consensus) graph to construct contigs. Below this

level of abstraction, a de Bruijn graph is utilized to resolve errors in short, repeated regions of the reference while an FM-Index is employed to fill the gaps in and extend paired-end reads. As a result of how the hybrid approach produces these extended contigs, single-end reads cannot be utilized. This is in direct contrast to the de Bruijn-based assemblers, which depend on paired-end reads for ordering or scaffolding contigs.

4.2.2 SOAP vs. Velvet

Both SOAP and Velvet perform various error-correcting and graph simplification methods at different points during their pipeline, not all of which occur directly on the constructed de Bruijn graph. For example, SOAP analyzes the initial library (or libraries) and prunes reads which can through consensus be determined as containing errors. With both SOAP and Velvet, most of the error correction methods mentioned below are performed iteratively until no new corrections are made.

In an effort to reduce memory consumption, Velvet simplifies the de Bruijn graph it generates by combining singly-linked nodes into a single larger node. For example, if a node A has an outgoing arc to node B, which has no other incoming edges, then the two nodes can be combined without affecting the overall assembly process.

SOAP's first method of correction is a simple removal of any node which does not exceed the minimum coverage threshold – a value which can be set by the user, but has a default value of one. For reasonable coverage depths, the presence of a one-off node highly suggests a sequencer error is present. Velvet employs a similar method of removing erroneous connections, but does not utilize it until after bubbles have been resolved.

With regard to shared correction methods, the two assemblers are first differentiated by their approaches to clipping tips, which is a node or a series of nodes disconnected at one end. In other words, a tip is a set of vertices and their corresponding edges which can be disconnected from the main graph by removing all outgoing edges from a single vertex in the set. The circled portion in 4.1 shows an example of a tip of length three. SOAP handles

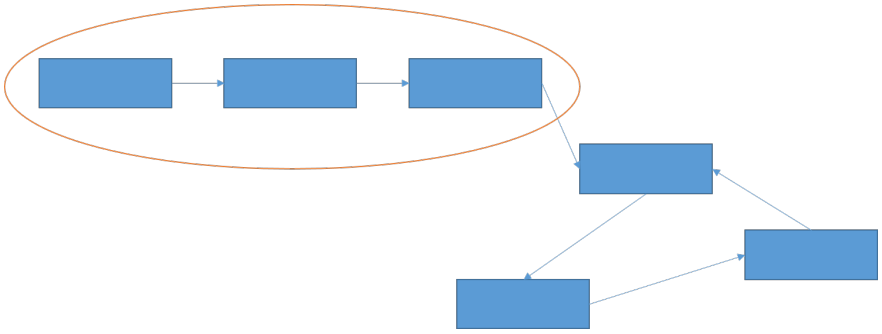

Figure 4.1: Example of a length three tip in a graph

tips through a predetermined cutoff length. By default, this length is 50 base pairs. Velvet, by comparison, modifies the cutoff to be twice the length of the k-mer value.

Perhaps the most complex portion of error correction in either tool is resolving bubbles, which are paths in the graph sharing both a start and end node. Since, natural biological variation (like single-nucleotide polymorphisms), highly similar regions, or sequencer errors can be the underlying cause of a bubble forming, some care is when attempting to collapse bubbles. Before analyzing a bubble to determine the perceived correct path, the average coverage (multiplicity) of the contigs formed by the parallel paths are considered. Only if both paths have a coverage lower than (by default) 60% of the average will one of the paths be collapsed. If this criteria is met, the analysis continues. Paths containing only a single base-pair difference are immediately resolved by examining path coverage. In addition, paths with 90% or greater similarity but fewer than four base-pairs difference are resolved in the same manner.

By contrast, Velvet employs the "Tour Bus" algorithm: progressing from an arbitrary node in the graph, a breadth-first search is employed across the graph. If at any point an already-visited node is discovered, a backtracking procedure from the current node and the already-visited node is engaged to determine their closest ancestor. These paths are then extracted and compared to determine the consensus path. The method of comparison is a variant on distance computed by dividing the length (in nodes) of the paths by the total multiplicity of the arcs. As with SOAP, there are some criteria which must be met if the

paths are to be resolved: branch length, divergence rate, and gap count. The first of these, branch length, is an indicator of commonality. If two paths don't have a common k-mer of at least 100 base pairs (this being the default value), then it is assumed the paths are sufficiently divergent to represent distinct portions in the reference. Similarly, if the alignment of the parallel paths diverges by more than 20%, bubble resolution does not occur. Finally, if more than three base pairs in the longest aligned sequences of the paths are unaligned, then we assume the paths are distinct.

Of the two methods of bubble resolution, Velvet's is the more strict of the two. As a result, we expect that in general (other things being equal), Velvet will produce a smaller number of total contigs. This expectation arises from there being fewer possible permutations of walks in the corrected and simplified de Bruijn graphs.

4.3 Expected Behavior with Input and Parameter Variation

In this section, an analysis of the behavior expected by each of the assemblers coverage depth, read length, single-end vs. paired-end read types, and, in the case of SOAP and Velvet, k-mer values vary will be presented. The behaviors expected will span both metrics utilized in the evaluation portion of the work and ancillary metrics.

4.3.1 Coverage

Coverage (sometimes referred to as depth), the average representation of each base in a library of reads, plays an important (and deceptively complex) role in assembly. On one hand, increasing coverage increases the probability of detecting and removing reads (or k-mers) which contain erroneous base calls. For basic error correction, reads or k-mers which appear only once (or below some minimum specified threshold) are considered to be erroneous and are subsequently pruned. For de Bruijn graphs, increased coverage can also mitigate the disconnection of components due to pruning – as coverage increases, the likelihood of cut-edges (edges whose removal disconnects a graph or component) is reduced. Clearly, a

true 1x coverage library (i.e. a library in which every base is represented exactly once), is unacceptably low; by its definition, there can be no overlap amongst reads, so reassembly is impossible. In addition, error detection cannot be performed. 2x coverage similarly prevents a guarantee of error detection, and still cannot be used to correct errors. In short, a true 3x coverage is the absolute minimum threshold for which a reasonable expectation of error correction can occur. Unfortunately, the library preparation methods currently employed cannot be relied upon to provide a true coverage; in fact, significant biases in coverage depth can be present [22]. As such, coverages in excess of 10x are usually employed to improve the odds of each region containing sufficient overlapping reads to effect quality reassembly.

The most straightforward effect of increasing coverage is an increase in the amount of system memory required to execute the assembly pipelines. Though the increase in library size is linear with the coverage, only a sub-linear growth in system memory requirements due to the size of the de Bruijn graph is expected. As coverage continues to increase, fewer additional unique k-mers will be added to the overall graph; most of the k-mers pulled from the added reads will be duplicates, which merely change the multiplicity of an existing edge or create a new edge.

Though a somewhat counterintuitive notion, increased read coverage does not always result in improved assemblies. Error correction methods for de Bruijn graphs are heavily dependent on multiplicity (the number of times a given k-mer appears in the library); furthermore, the emphasis both SOAP and Velvet place on correct assemblies (rather than maximal length assemblies) results in their methodologies being highly conservative. Increased coverage tends to result in more highly-connected graphs bearing edges with larger multiplicity values – both attributes that make it difficult to prune and simplify. As a result, excessively high coverage can hamper the performance of de Bruijn graph-based assemblers. One method of increasing the coverage threshold at which this performance degradation occurs is to increase the k-mer value – as explained below, larger k-mer values tend to result in graphs with increased node counts but less overall connectivity. It should also be noted,

however, that larger k-mer values increase the minimum threshold of coverage for meaningful assembly as well.

With StriDe, the read extension portion of the assembler handles improved coverage quite well (since the extension process is not dependent on k-mer multiplicity); however, at very high coverages the de Bruijn portion of StriDe is expected to encounter the same manner of performance degradation as seen in SOAP and Velvet.

4.3.2 Read length

One natural effect of increasing read length (while keeping other parameters constant) is a monotonic increase in the minimum length of a maximum trail in the graph. That is, an ordered set of adjacent edges (no one of which appears more than once) which cannot be made longer and is at least as long as any other maximal trail in the graph. This effect can be demonstrated by examining three cases: a read which contains on a single unique k-mer, one which has the maximum (n - k + 1) number of unique k-mers, and one which has a number of unique k-mers greater than one but strictly less than the maximum. In the first case, the subgraph generated is a single vertex with self-referencing directed edge of multiplicity n - k + 1. Clearly, the length of any maximal trail in this subgraph is one; furthermore, this length cannot be changed by increasing or decreasing the length of the read, since only the multiplicity tied to length. In the second case, a maximum trail is constructed by starting with the edge incident to the vertex containing no incoming edges and ending with the edge incident to the vertex containing no outgoing edges. The length of this trail will be exactly n - k + 1, where n is the length of the read and k is the choice of k-mer length. Holding k constant, we can see that increasing the read length will result in a strict increase in the length of the trail. For the third case, consider a read of arbitrary length containing a non-maximum number of unique k-mers. If we extend that read by one base pair, two outcomes are possible: either the additional base produces a non-unique k-mer, in which case a directed edge is produced pointing to an existing node, or a new unique k-mer

is produced, resulting in the creation of a new node. In the first case, the larger read has a maximum path of equal length; in the latter, it has a longer maximum path.

From the observation above, we can see that any tips present in the de Bruijn graph will, as read length increases, have a tendency to grow in length as well. As a result, fewer tips arising from non-erroneous portions of the graph should be clipped.

If the average insert/fragment size of a read is held constant across increasing read length, StriDe should show a significant increase in overall assembly quality. This is due to the reduced FM-walk length required to bridge the gaps between each of the paired-ends of a particular read. For example, if the average fragment size is 500 and read length is 75, there is a 350 base pair gap between reads. Increasing the length of the reads to 150 reduces this gap to 200 base pairs, resulting in a walk that is both statistically more likely to succeed and computationally faster to perform.

4.3.3 Read type

From the perspective of assembler architectures, no differentiation is made between single and paired-end reads for all phases up to scaffold construction. For SOAP and Velvet, no additional correlations or considerations are given to paired-end data, and StriDe was constructed with the specific goal of utilizing paired-end reads. For the evaluated de Bruijn-based assemblers, the primary benefit to paired-end reads is the improved ability to produce scaffolds. By utilizing the original paired-end reads as a skeleton, assembled contigs can be assigned relative orderings.

Despite a lack of differentiation at the architectural level, the underlying structure of paired-end reads compared to single-end reads nonetheless has the potential to affect overall library (and, by extension, assembly) quality.

4.3.4 *k*-mer Value

The choice of k-mer value has a number of effects on the execution of de Bruijn graph-based assemblers. The most immediate of these effects is memory consumption – as k-mer size increases, the number of unique k-mers which can be generated grows. Consider a k-mer size of 23 (the default value used for the two assemblers in this work): given that each vertex is constructed of a unique string of length k-1, there are on the order of 10^{13} possible strings in this space. Changing the k-mer size to 31 increases the size of this space to approximately 10^{18}. Though both of these string spaces far exceed the typical length of genomes being assembled, the number of unique nodes in generated graphs is nonetheless higher for all but the most trivial of graphs (e.g. references composed entirely or almost entirely of repeated sequences). As a result, increasing k-mer size usually results in a correspondingly larger memory requirement. Increases in read length and coverage depth magnify this effect.

It is usually the case that larger values of k result in simpler graphs, as there is a higher level of uniqueness amongst nodes. As a trade-off, deeper levels of coverage and higher read lengths are required for optimum assembly. For example, setting the k-value equal to the read length would result in only those reads which shared k-1 common bases being connected. As this is a fairly uncommon occurrence, the connectivity of the graph would be quite poor, resulting in low average contig length.

Chapter 5

Evaluation

This chapter details the experimental evaluations performed as well as an analysis of the findings. For each of the 16 datasets, SOAPdenovo (henceforth referred to as SOAP) and Velvet were run using k-mer sizes of 23 and 31. For the eight paired-end datasets, StriDe was also run using its default configuration. Several important metrics are recorded for each execution (displayed in a table for convenience), and graphical representations of data are included were relevant. Prior to the actual evaluation and analysis, a brief synopsis on the testing environment and a description of the evaluation criteria will be provided.

5.1 Testing Environment

5.1.1 Hardware Specifications

All sequence simulations, assemblies, and analyses were performed independently across a set of uniform nodes. Each node possesses a 24-core Intel Xeon X5650 clocked at 2.67 GHz, 24 Gigabytes of system memory, and a 500 Gigabyte Western Digital SATA hard disk drive, 200 Gigabytes of which were allocated to the partition experiments were performed in.

5.1.2 Assembler Parameters

Aside from the variations in k-mer for SOAP and Velvet, the only parameter modification introduced was a reduction in minimum contig size in Velvet to 100. This was done to match the default minimum contig size of SOAP.

Table 5.1: Library size in Megabytes

	10x	30x	50x	70x
75 BP, Single	316	948	1581	2217
75 BP, Paired	320	960	1600	2244
150 BP, Single	301	905	1509	2112
150 BP, Paired	304	912	1518	2126

5.1.3 Data sets

In total, sixteen datasets were generated for evaluation purposes. Each data set is based on ART simulated Illumina reads from *C. elegans* Chromosome 1. Both single and paired-end reads were generated, and each of these read types was produced with a length 75 and length 150 variant. For paired-end reads, an average fragment size of 500 was used. Average coverage was swept from 10x to 70x in intervals of 20, for a total of four coverage depths. Table 5.1 below shows the size of each generated dataset.

5.2 Evaluation Criteria

For each set of data, eight metrics were collected to evaluate the relative performance of each assembler:

- The **# Contigs** metric is a numerical value denoting the total number of separate contiguous segments of length greater than or equal to 100 that were generated by the assembler.
- **Max Contig** denotes the length of the single largest contiguous fragment assembled.
- **Total Length** is the sum length of all contigs reported by the assembler.
- **N50** is a commonly used evaluation metric providing a succinct description of contig length distribution. The N50 length is determined by adding contig lengths in descending order until 50% of the total assembly size has been reached – the contig length at which this value was reached is N50. In other words, N50 length is the value at which

the total length of all contigs less than N50 is approximately equal to the length of all contigs greater than N50. As an evaluation metric, larger N50 values are considered desirable.

- **N75** is analogous to N50, but with 75% of the total assembly length. N75 will always be less than or equal to N50, and as with N50 higher numbers are generally considered desirable.

- **L50** indicates the minimum number of contigs that sum to 50% of the total assembly length. A small L50 value (relative to the total number of contigs) means that the assembly possesses higher quantities of large contigs.

- **Misses** represents the total number of misassembled contiguous fragments as determined by comparison against the reference genome.

- **Miss Length** adds context to misses by providing the sum length of the erroneous contigs.

5.3 Length 75 Single-end Reads

In examining table 5.2, several correlations amongst evaluation criteria and assembler/sequencer variations should noted. The first of these is the presence of misassembled contigs in the final assembly. For both assemblers operating with either a size 23 or 31 k-mer, a 10x coverage produces multiple errors. This indicates the presence of at least one region wherein coverage depth was insufficient. Considering Velvet's superior performance relative to this metric and taking into account the known architectural differences between the two assemblers, the probably nature of these errors can be analyzed. Given the stricter tip-cutoff threshold of SOAP with a k-mer value of 23 (length 50 versus length 46 for Velvet), it can be concluded that erroneous tips are not the cause of SOAP's greater number of misassemblies. Certainly, errors resulting from erroneous tips are possible; however, a tip which failed to

Table 5.2: Evaluation of Length 75 Single-end Reads

	# Contigs	Max Contig	Total Length	N50	N75	L50	Misses	Miss Length
SOAP K23								
10x	57685	1371	11713964	219	153	17882	63	22414
30x	57200	926	9788405	176	131	19490	1	221
50x	46400	546	6470853	136	114	18082	1	232
70x	36957	466	4708010	123	109	15379	1	279
SOAP K31								
10x	53575	1635	9160016	175	131	18227	39	14250
30x	21616	8238	14146131	1187	611	3442	3	2748
50x	25790	7252	14072723	983	504	4203	0	0
70x	41366	3734	13931617	494	264	8552	0	0
Velvet K23								
10x	47131	2672	10714619	258	165	13027	17	7968
30x	45961	1312	8852114	205	144	14447	0	0
50x	35100	649	5279587	149	119	12957	0	0
70x	19112	502	2468233	124	109	7832	0	0
Velvet K31								
10x	63296	1470	10797060	174	132	21730	18	6009
30x	24743	9038	13717099	1034	488	3604	0	0
50x	31205	6343	13950388	789	383	5014	0	0
70x	45202	4249	14147248	474	238	8790	0	0

be pruned by SOAP would also pass Velvet's clipping measures at this k-mer value. The most likely cause of these errors (and, in particular, the discrepancy in their quantity) is due to the different standards employed in resolving bubbles. One or more of the bubbles successfully resolved by Velvet was considered a distinct contig by SOAP, thus introducing a misassembly for each contig which included the erroneous path.

As expected, increase in total coverage translated into a decline in misassemblies for all of the assembler/k-mer combinations. Velvet was more quickly able to completely eliminate its own errors, owing to the aforementioned stricter bubble resolution method: by 30x coverage, both k-mer variants were error free. Comparatively, the 23-mer variant of SOAP carried a single misassembly throughout the 30x-70x coverage ranges, and the 31-mer variant was able to purge its final three errors by 50x coverage.

In assessing assembly quality by the length of the maximum contig and the N50 value, the trends are quite straightforward. As seen in both Figures 5.2 and 5.1, the 23-mer

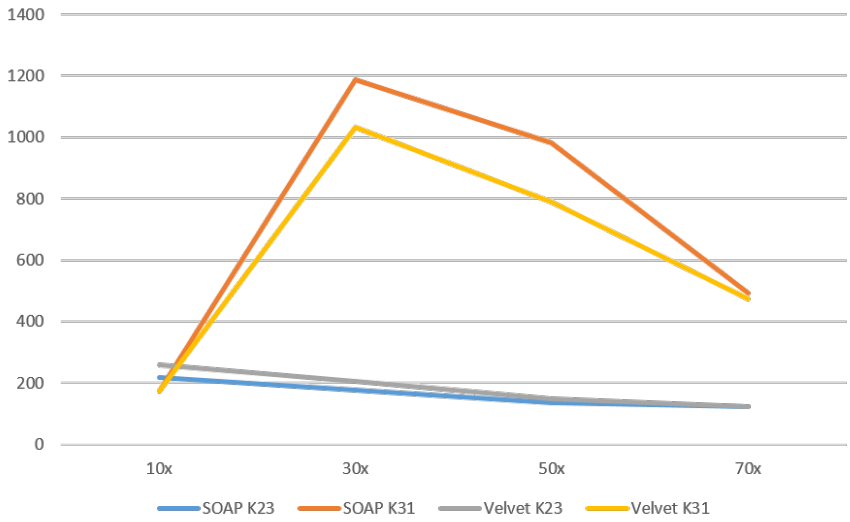

Figure 5.1: Length 75 Single-end Read N50 Values

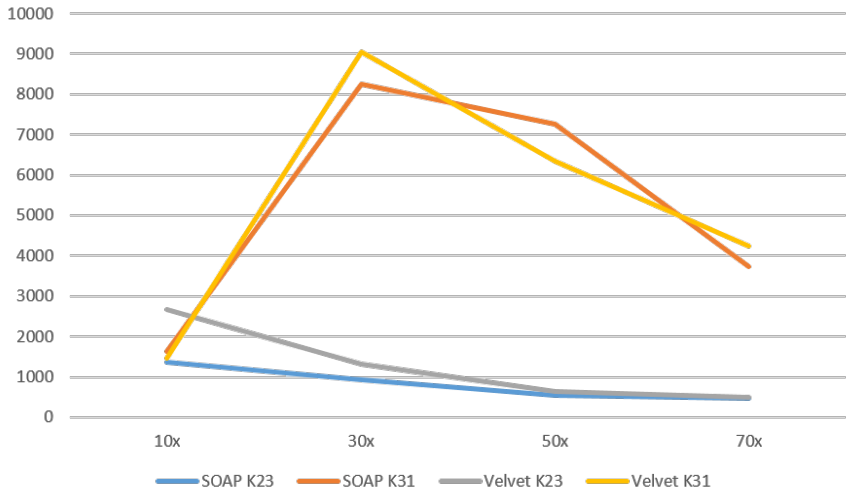

Figure 5.2: Length 75 Single-end Read Max Contig Values

variants of both SOAP and Velvet reached a coverage saturation threshold between 10x and 30x, resulting in successively worse assemblies for higher coverages. A similar threshold was crossed between 30x and 50x for the 31-mer variants.

An interesting differentiation between the 23-mer and 31-mer variants of both SOAP and Velvet is how they response to coverage saturation. For the 23-mer variants, a successive reduction in both contig count and total contig length is evident; conversely, the 31-mer variants begin to produce larger quantities of contigs while maintaining a total contig length that is relatively stable (within 3% of the median).

5.4 Length 150 Single-end Reads

For both 23-mer assemblers, the increase in read length compared to the length-75 datasets resulted in superior assemblies with respect to all of the recorded metrics in the 10x coverage case. Of particular note are SOAP's sharp decrease in errors and Velvet's complete elimination of misassemblies.

Overall, the increased read length drastically reduced the quantity of errors across all four assembler/k-mer variants. Both Velvet variants were error-free across the board, and SOAP produced only five and three errors respectively for the 23-mer and 31-mer variants at 10x coverage. Though the number of individual reads is roughly halved compared to the length 75 data sets,

With regard to coverage thresholds, the 23-mer variants display the same behavior as the length 75 tests; however, the threshold for the 31-mer variants changed from between 30x-50x to 10x-30x. Additionally, the magnitude of the performance drop-off is far more significant after crossing the threshold. For the length 75, 23-mer evaluations, SOAP and Velvet saw, respectively, approximately 35% and 50% reductions in max contig length between 10x and 30x. The same coverage difference for the 150 length evaluations produced maximum contigs reduced by approximately 65% for both. As seen in Figure 5.3, 23-mer SOAP and Velvet maintained the same trends of total read length degradation as in the length 75 case;

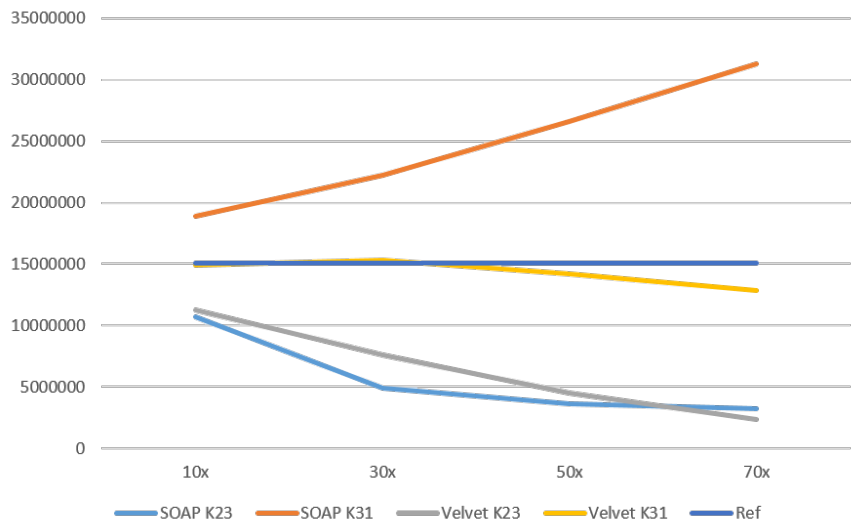

Figure 5.3: Length 150 Single-end Read Total Length

however, as Figure 5.4 displays, 23-mer Velvet saw a small increase in contig count at 30x before reverting to the length 75 trend.

31-mer Velvet also maintained a similar trend with regard to the length 75 coverage saturation effects; however, the total contig length did not possess the same level of stability. 31-mer SOAP, by contrast, exhibited new behavior: accompanying a drastic increase in total contig count, the total length became unbounded, resulting in the 70x coverage assembly containing more than twice the total length of the reference.

5.5 Length 75 Paired-end Reads

Though executed on several different nodes (and attempted several times on each), the 70x 31-mer variant of SOAP was never able to successfully run to completion. An examination of system resource utilization indicated this was due to insufficient available memory. As shown in Table 5.1, the length 75 paired-end library was the largest 70x coverage library in size at 2,244 MB.

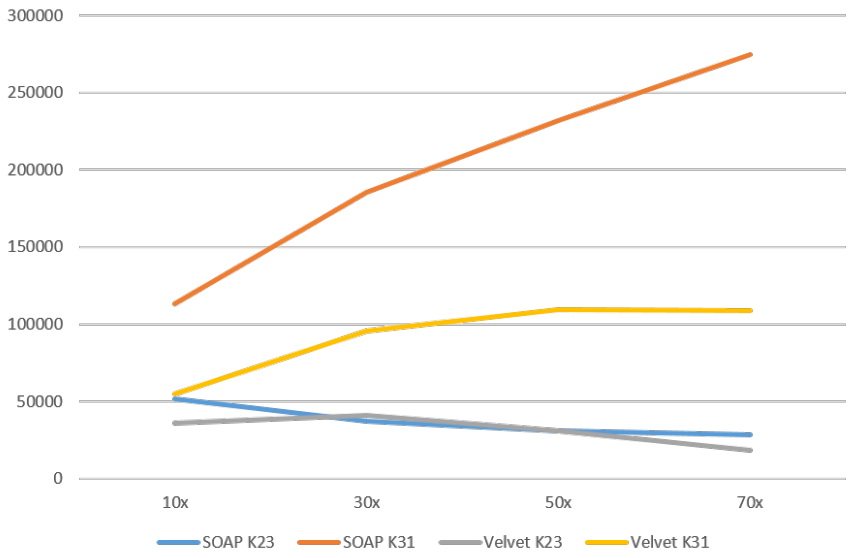

Figure 5.4: Length 150 Single-end Read Contig Count

Table 5.3: Evaluation of Length 150 Single-end Reads

	# Contigs	Max Contig	Total Length	N50	N75	L50	Misses	Miss Length
SOAP K23								
10x	51863	1427	10682585	231	150	14951	5	2303
30x	37561	498	4937472	125	108	14920	0	0
50x	30884	404	3633955	112	104	13534	0	0
70x	28640	340	3247480	109	103	12972	0	0
SOAP K31								
10x	113329	1481	18893817	162	114	32292	3	1186
30x	185606	777	22250175	114	106	80374	0	0
50x	231595	410	26559093	112	105	105081	0	0
70x	274748	374	31293086	111	105	125548	0	0
Velvet K23								
10x	36284	3544	11227901	412	229	8066	0	0
30x	41148	1196	7631244	199	136	12730	0	0
50x	31163	835	4543248	143	114	11467	0	0
70x	18701	457	2366482	120	107	7686	0	0
Velvet K31								
10x	55160	3872	14922057	434	164	9562	0	0
30x	95736	1561	15321033	1561	113	28317	0	0
50x	109397	1177	14238789	119	107	43290	0	0
70x	108754	582	12877708	113	106	47818	0	0

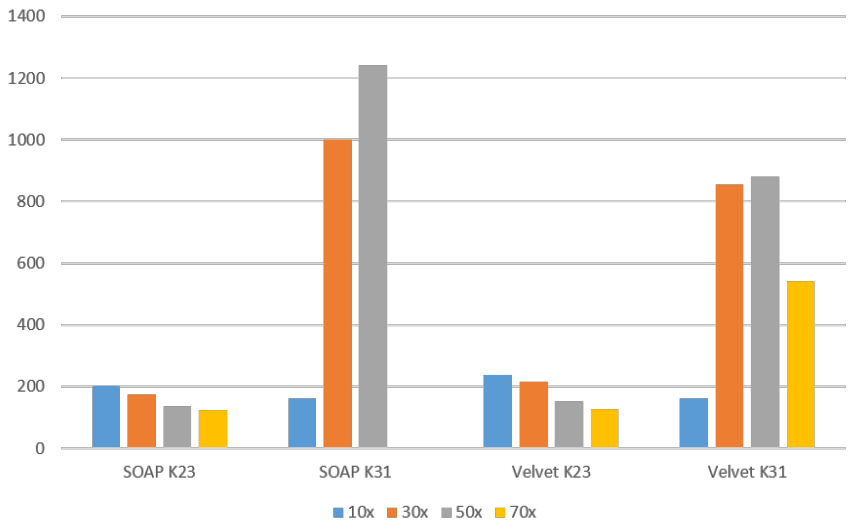

Figure 5.5: Length 75 Paired-End Read N50 Values

The presence of paired-end reads appears to shift the coverage saturation threshold up for each of the de Bruijn graph-based assemblers. Though the threshold is still between 10x-30x for the 23-mer variants, the differences in max contig length are only (approximately) 5% and 25% for SOAP and Velvet, respectively. The threshold for 31-mer SOAP moved up a bracket to 50x-70x, with significant performance gains being realized across all metrics at 50x coverage. As shown in Figure 5.6, max contig value at this coverage was a particularly large improvement. Velvet saw a slight degradation in max contig length between 50x and 70x, but N50 and N75 values were improved (as seen in Figure 5.5).

StriDe performed abysmally at 10x coverage, managing only a tiny fraction of the reference length; however, increases in coverage successively improved the quality of contigs. Because StriDe's metrics exceeded the others by as much as two orders of magnitude, it was excluded from the relevant figures. Compared to the pure de Bruijn assemblers, StriDe was more error prone; furthermore, the errors produced had a tendency to be far longer in length.

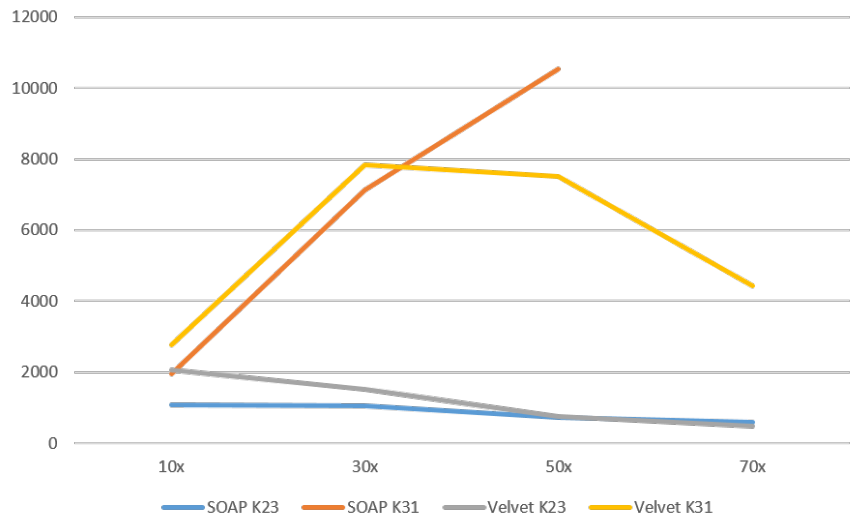

Figure 5.6: Length 75 Paired-end Read Max Contig Values

Table 5.4: Evaluation of Length 75 Paired-End Reads

	# Contigs	Max Contig	Total Length	N50	N75	L50	Misses	Miss Length
SOAP K23								
10x	58877	1086	11324513	204	145	18835	52	19068
30x	58264	1048	9924720	175	131	19857	4	1209
50x	47652	724	6671573	137	114	18460	1	417
70x	38639	597	4920418	123	108	16062	2	489
SOAP K31								
10x	49883	1948	7987105	161	125	17742	34	11077
30x	24098	7129	14159596	1001	512	4186	4	3343
50x	22253	10527	14089207	1243	611	3228	1	437
70x	-	-	-	-	-	-	-	-
Velvet K23								
10x	48515	2069	10474720	238	157	13914	18	8169
30x	45957	1528	9143003	214	148	14071	0	0
50x	36624	755	5601363	152	120	13344	0	0
70x	21057	470	2749860	126	110	8562	0	0
Velvet K31								
10x	61718	2765	9915372	161	126	22055	30	9917
30x	28053	7851	13632213	854	399	4400	0	0
50x	29104	7505	13951805	882	418	4439	0	0
70x	41360	4429	14151474	540	270	7663	0	0
StriDe								
10x	45	2960	39782	1247	1026	12	7	7076
30x	3800	22925	9758376	3639	1908	768	263	9386
50x	2520	89869	13007164	13103	4520	245	30	180152
70x	2381	99180	12953732	14828	4972	222	5	33263

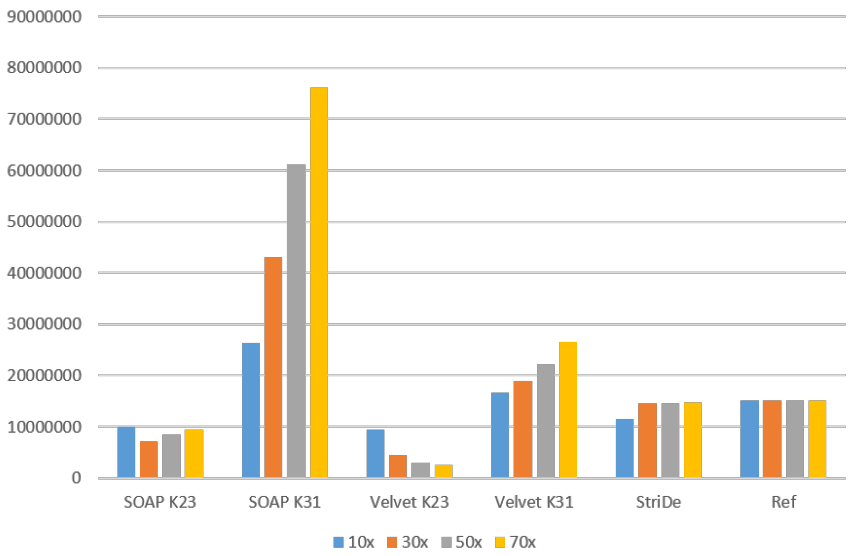

Figure 5.7: 150 Length Paired-End Read Total Length

5.6 Length 150 Paired-end Reads

The unbounded total contig length growth pattern displayed by 31-mer SOAP in the length 150 single-end reads appears once again; however, with paired-end reads the effect is magnified significantly. At 70x coverage, the total length of the assembly is more than five times the reference length! Figure 5.7 shows that Velvet also begins to display this behavior, albeit at a far more subdued severity.

As with the length 75 evaluations, StriDe is still quite error prone compared to the de Bruijn assemblers; however, the other metrics are comparatively staggering. In each of the remaining evaluation criteria, StriDe outperforms SOAP and Velvet by at least one order of magnitude.

Table 5.5: Evaluation of Length 150 Paired-End Reads

	# Contigs	Max Contig	Total Length	N50	N75	L50	Misses	Miss Length
SOAP K23								
10x	67922	883	9979364	141	113	24273	2	573
30x	62221	325	7084690	111	104	28275	0	0
50x	76421	266	8574654	109	104	35153	0	0
70x	85059	268	9494960	109	103	39243	0	0
SOAP K31								
10x	192329	836	26298417	128	111	74301	0	0
30x	367562	391	43116968	115	106	165117	0	0
50x	523424	312	61168943	115	106	236060	0	0
70x	654013	338	76270424	115	106	295340	0	0
Velvet K23								
10x	48094	1796	9445505	220	137	13556	0	0
30x	36003	520	4564080	118	106	14594	0	0
50x	26430	333	2950035	108	103	12152	0	0
70x	24225	204	2642031	107	103	11371	0	0
Velvet K31								
10x	99526	2050	16746772	156	114	27345	0	0
30x	156172	702	19026051	116	107	67144	0	0
50x	192552	442	22151657	113	106	87714	0	0
70x	232743	337	26516469	112	105	106818	0	0
StriDe								
10x	6062	17831	11492721	2884	1543	1197	454	1089840
30x	2938	180648	14533882	22734	7448	151	30	174234
50x	2967	158641	14656661	23154	8351	153	25	114089
70x	2947	147100	14689865	23386	8480	153	15	143404

Chapter 6

Conclusion and Future Work

With continued reductions in sequencing costs and increased throughput, the quantity and variety of sequencing data will continue to expand. The variety and quality of genome assembly tools will likely continue to grow unabated as well. More than ever, there will be a need for periodic, comprehensive evaluations.

While other evaluation metrics focus on optimizing parameters for datasets with fixed or only marginally variable sequencer attributes, this work presents an in-depth analysis of several popular de Bruijn graph-based assemblers as well as a recently released hybrid de Bruijn/String graph assembler. In addition, detailed behavioral expectations across a variety of sequencer attributes are explored through evaluation and examination of the underlying graph structures being generated. In general, the evaluations matched up quite nicely with the analysis; however, novel behaviors arising from specific combinations of sequencer attributes were also identified. By utilizing the analysis presented in this work, it is the hope of the author that both tuning of assemblers employing these architectures and selection of appropriate data sets will be enhanced.

For future work, several directions could be taken to further enhance this work. Comparing the evaluations included in this work with error-free versions of the same simulated libraries could provide further insight into the behaviors of the assemblers by isolating one of the major sources of aberrant behavior. Additionally, increasing the granularity and range of coverages, k-mer values, and read lengths could reinforce the observations made during this work and possibly identify optimal combinations without rigorous empirical trials. Including additional reference genomes of larger and smaller sizes would also provide opportunities to observe assembler behavior. Adding additional assemblers (especially those like StriDe which

haven't been previously evaluated) and refreshing existing assemblers using newer releases would add to the robustness of the work and maintain its relevance. Finally, packaging the evaluations into an automated suite would greatly reduce the overhead associated with testing and enable developers to help maintain and improve the body of evaluations.

Bibliography

[1] Applied biosystems solid 4 system.

[2] Gs flx+ system.

[3] Hiseq x series of sequencing systems.

[4] Ion s5 and ion s5 xl next-generation sequencing system specifications.

[5] Paired-end sequencing.

[6] Maxam A.M. and Gilbert W.A. A new method for sequencing dna. *Proc. Natl. Acad. Sci. USA*, 1977.

[7] Keith R Bradnam, Joseph N Fass, and et al. Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species. *GigaScience*, 2013.

[8] N.G. De Bruijn. A combinatorial problem. *Nederl. Akad. Wetensch. Proc.*, 1946.

[9] H Chial. Dna sequencing technologies key to the human genome project. *Nature Education*, 2008.

[10] Phillip E C Compeau, Pavel A Pevzner, and Glenn Tesler. How to apply de bruijn graphs to genome assembly. *Nature Biotechnology*, 2011.

[11] Dent Earl and et al. Assemblathon 1: A competitive assessment of de novo short read assembly methods. *Genome Research*, 2011.

[12] Sanger F., Air G.M.and Barrell B.G., Brown N.L., Coulson A.R., Fiddes J.C., Hutchison C.A., and Smith M. Nucleotide sequence of bacteriophage phx174dna. *Nature*, 1977.

[13] Sanger F., Nicklen S., and Coulsen A.R. Dna sequencing with chain-terminator inhibitors. *Proc. Natl. Acad. Sci. USA*, 1977.

[14] Alexey Gurevich, Vladislav Saveliev, Nikolay Vyahhi, and Glenn Tesler. Quast: Quality assessment tool for genome assemblies. *Bioinformatics*, 2013.

[15] Weichun Huang, Leping Li, Jason R. Myers, and Gabor T. Marth. Art: a next-generation sequencing read simulator. *Bioinformatics*, 2012.

[16] Yao-Ting Huang and Chen-Fu Liao. Integration of string and de bruijn graphs for genome assembly. *Bioinformatics*, 2016.

[17] Darryl Leja. Shotgun sequencing.

[18] Ruiqiang Li1, Yingrui Li, Karsten Kristiansen, and Jun Wang. Soap: short oligonucleotide alignment program. *Bioinformatics*, 2008.

[19] Lin Liu, Yinhu Li, Siliang Li, Ni Hu, Yimin He, Ray Pong, Danni Lin, Lihua Lu, , and Maggie Law. Comparison of next-generation sequencing systems. *Journal of Biomedicine and Biotechnology*, 2012.

[20] Ruibang Luo, Binghang Liu, Yinlong Xie, Zhenyu Li, Weihua Huang, Jianying Yuan, Guangzhu He, Yanxiang Chen, Qi Pan, Yunjie Liu, Jingbo Tang, Gengxiong Wu, Hao Zhang, Yujian Shi, Yong Liu, Chang Yu, Bo Wang, Yao Lu, Changlei Han, David W Cheung, Siu-Ming Yiu, Shaoliang Peng, Zhu Xiaoqian, Guangming Liu, Xiangke Liao, Yingrui Li, Huanming Yang, Jian Wang, Tak-Wah Lam, and Jun Wang. Soapdenovo2: an empirically improved memory-efficient short-read de novo assembler. *GigaScience*, 2012.

[21] ER Mardis. The impact of next-generation sequencing technology on genetics. *Trends Genet.*, 2008.

[22] Maura Costello Andrew Hollinger Niall J Lennon Ryan Hegarty Chad Nusbaum Michael G RossEmail author, Carsten Russ and David B Jaffe. Characterizing and measuring bias in sequence data. *Genome Biology*, 2013.

[23] Niranjan Nagarajan and Mihai Pop. Sequence assembly demystified. *Nature Reviews Genetics*, 2013.

[24] Pavel A. Pevzner, Haixu Tang, and Michael S. Waterman. An eulerian path approach to dna fragment assembly. *Proc. Natl. Acad. Sci*, 2001.

[25] Li R, Zhu H, Ruan J, Qian W, Fang X, Shi Z, Li Y, Li S, Shan G, Kristiansen K, and et al. De novo assembly of human genomes with massively parallel short read sequencing. *Genome Research*, 2010.

[26] Steven L. Salzberg, Adam M. Phillippy, Aleksey Zimin, Daniela Puiu, Tanja Magoc, Sergey Koren, Todd J. Treangen, Michael C. Schatz, Arthur L. Delcher, Michael Roberts, Guillaume Marais, Mihai Pop, and James A. Yorke. Gage: A critical evaluation of genome assemblies and assembly algorithms. *Genome Research*, 2011.

[27] Jared T. Simpson and Richard Durbin. Efficient construction of an assembly string graph using the fm-index. *Bioinformatics*.

[28] Daniel R. Zerbino and Ewan Birney. Velvet: Algorithms for de novo short read assembly using de bruijn graphs. *Genome Research*, 2008.