2009

Parallel methods for short read assembly

Benjamin Grant Jackson
Iowa State University

Parallel methods for short read assembly

by

Benjamin Grant Jackson

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:
Srinivas Aluru, Major Professor
Patrick S. Schnable
David Fernandez-Baca
Suraj C. Kothari
Joseph A. Zambreno

Iowa State University

Ames, Iowa

2009

DEDICATION

I dedicate this work to my wife Adrianna; her support and patience were essential to my sanity during its completion.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

List of Algorithms

CHAPTER 1. INTRODUCTION

This work is on the parallel *de novo* assembly of genomic sequences from short sequence reads. In it, we advance the field of short sequence assembly in a number of ways. First, we extend models and ideas proposed and tested with small genomes on serial machines to large-scale distributed memory parallel machines. Second, we present novel ideas for assembly that are especially suited to reconstruction of very large genomes on these machines. Additionally, we present the first assembler that specifically takes advantage a variable number of fragment sizes or insert lengths while still working well for data with one insert length.

In describing this work, we do not assume that the reader has a background in biology. While we assume a familiarity with computer science,[1] we do not expect a specific background in parallel processing. In our software architecture, we encapsulate the parallel implementation details in a few high level computational concepts. Ultimately these concepts require a parallel mindset, but readers interested more in the general method may skip the implementation details.

As a biological background is not assumed, we will start our discussion with a brief description of DNA. The deoxyribonucleic acid molecule, as shown in *Fig. 1.1*, is a double-stranded molecule, each strand a chain (or *polymer*) of simpler molecules known as *nucleotides*. A nucleotide consists of a sugar-phosphate backbone with an attached base taken from the set {Adenine, Guanine, Cytosine, Thymine}, as we highlight in the figure. As the base differentiates each nucleotide, a nucleotide is often referred to by its base. In fact, we think of the polymer as a string over the alphabet $\Sigma = \{a, c, g, t\}$, and the DNA molecule as two such strings.

[1] This background would include an understanding of the design and analysis of algorithms, complexity theory, and discrete mathematics.

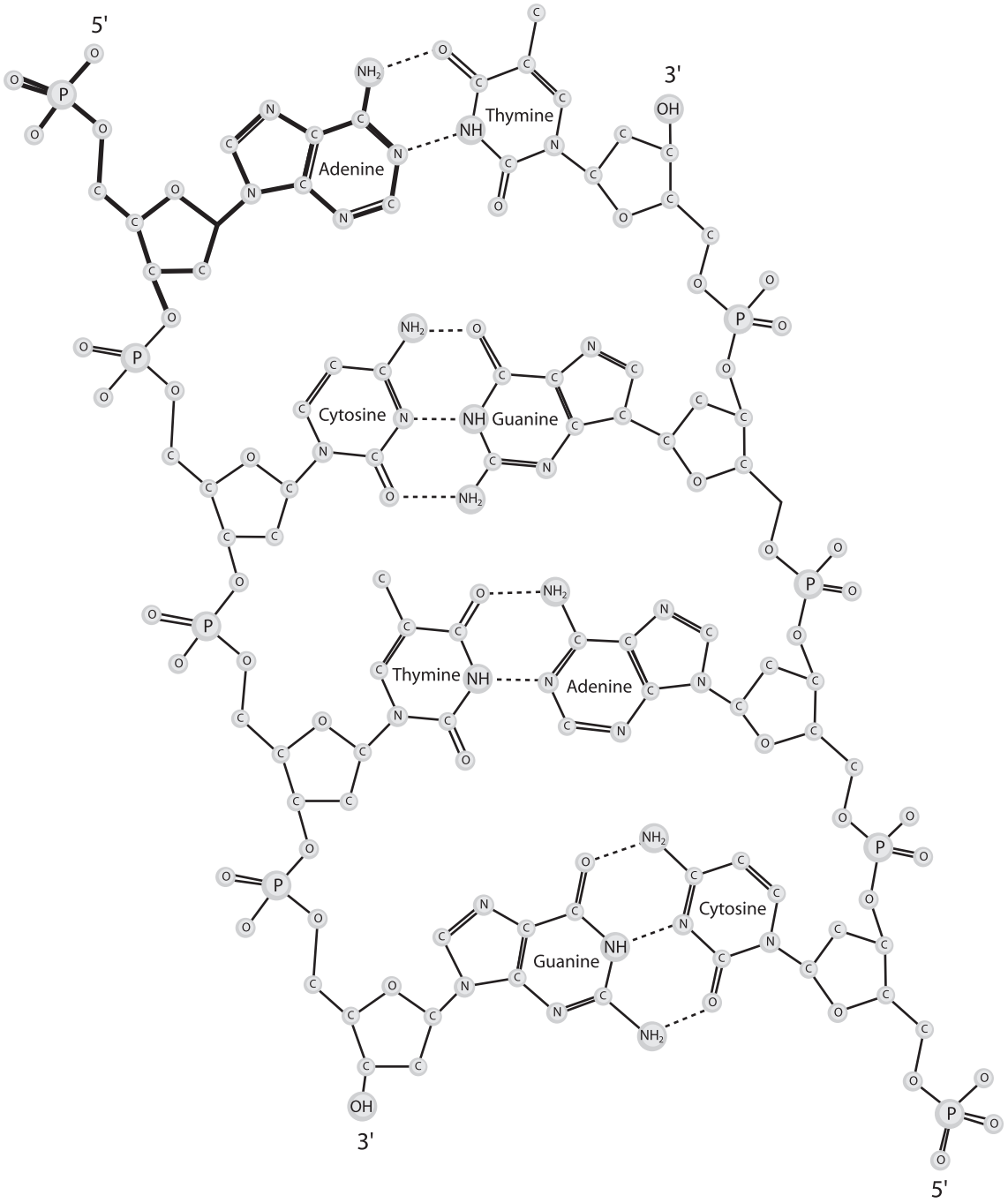

Figure 1.1 The chemical structure of DNA. Molecules are labeled with their component elements: phosphorous (P), oxygen (O), hydrogen(H), nitrogen(N), and carbon(C). Covalent bonds are shown as solid lines in the graph, while hydrogen bonds are shown as dashed lines. We highlight a single nucleotide in the upper left. We show in the figure the double stranded nature of DNA, the complementary bases Adenine, Cytosine, Guanine, and Thymine, and the strands' opposite orientations.

The two strands of DNA have opposing orientations, and therefore we term them *anti-parallel*. We say the forward direction of the DNA molecule is from the 5' to 3' end, the same direction in which bases are incorporated by a *DNA polymerase*. Each nucleotide is uniquely paired with a nucleotide on the opposite strand (a with t, c with g, and vice versa), each pair called complementary. For this reason, given one strand's string representation as S, the second strand can be constructed by reversing the characters of S and taking the complement of each, producing the string denoted S'. This is known as finding the *reverse complement*.

In most eukaryotes, such as humans, the nuclear DNA (or *genome*) of an organism consists of a small number of pairs of DNA molecules, each pair called a *chromosome*. Because of this pairing we refer to such organisms as *diploid*. When describing the length of the genome of an organism, we count the two molecules in a single chromosome once. We count this way because the two molecules are nearly identical, differing in only a few nucleotides. Initially, when finding a reference assembly for an organism that is diploid, scientists are interested in only one reference sequence for each chromosome, essentially ignoring these differences.[2] Humans have 23 chromosomes totaling over 3 billion bases.

Unlike eukaryotes, prokaryotes like bacteria are *haploid*, having only one molecule of DNA connected in a circle. The DNA of bacteria tends to be much shorter than that of eukaryotes; for example the bacteria *E. coli* has 5.4 million bases.

DNA has been called the blueprint of an organism. This is because of its primary role as information encoder. An organism's *genes* are subsequences of the DNA which are translated into proteins and other cellular molecules, which in turn interact with other organic and inorganic molecules in the complex system we call life. As the details of this process are far beyond the scope of this work, we proceed with the assumption that our understanding of molecular biology is enhanced by knowing an organism's DNA; The assembly problem is important to solve.

Like DNA, *ribonucleic acid* (RNA) is a polymer composed of four nucleotides, but with Thymine replaced by Uracil. Unlike DNA, RNA is single stranded. One of RNA's primary

[2] One way in which scientists are making use of short read technologies is in attempting to characterize genetic diversity (differences from the reference) in populations of individuals of the same species.

roles in the cell is that of a messenger, transferring the information stored in the DNA to other parts of the cell. The combined sequences of all the messenger RNA in the cell is called the *transcriptome* of the organism. We discuss the assembly of the transcriptomes in *Section 3.1.*

A DNA sequencing machine reads a small portion of a single strand of DNA along its direction. In the next section we will outline some experimental processes used to achieve this goal, but in this work we are especially concerned with the properties of the resulting data, as needed to reconstruct the originating genome.

Importantly, our work is in direct response to changes in the sequencing processes and the corresponding changes to the properties of the experimental data. For nearly three decades from its invention, Sanger sequencing - which produces 700 to 1000 base-pair reads - dominated the field of DNA sequencing and genome assembly. New developments in high-throughput short read sequencing are proving a disruptive technology that allows concurrent generation of millions of reads at a significantly lower per base cost, albeit with limitations on read length (35-75 bases typically, with the exception of 454, which can produce 160 base paired reads). Several such platforms are available and seeing rapid adoption in the experimental biology community (454 Life Sciences system [38], Illumina Solexa [3], Applied Biosystems SOLiD [47], and Helicos Biosciences Heliscope [70]).

Researchers initially aimed to use short-read sequencing to resequence individuals when a template genome of that individual's species was previously known. This involves aligning the reads to the reference genome, avoiding a *de novo* assembly, and provides an easy way for biological analysis such as identifying single nucleotide polymorphisms (SNPs). The Illumina system has been used for resequencing [4][66], identifying repeats [69], and characterizing population diversity [46].

As mentioned above, this work instead focuses on the computational methods necessary for successfully reconstructing an unknown genomic sequence from the short read data. Given the multimillion dollar expense associated with traditional genome sequencing projects using Sanger sequencing, high-throughput technologies offer the only hope in sequencing a much larger number of species. Also, *de novo* assembly is important in cases where significant

genomic rearrangements are expected, such as when sequencing multiple inbred lines of the same plant species.

Several *de novo* short read assemblers have recently been developed – ALLPATHS [6], Euler-SR [7], SHARCGS [12], Shorty [23], Edena[22], Medvedev *et al.* [40], SSAKE [67], Velvet [71], and ABySS [60]. Each of these works have included novel ideas on how to tackle this important problem, but they all (save ABySS, which was published shortly after our work) face the same fundamental limitation: they ignore resource problems inherent with limiting a solution to a single processor architecture. We address this limitation by developing an assembler that effectively uses massively parallel high performance computers and the large distributed memories available on these machines. Our techniques allow for the *de novo* reconstruction of gigabase sized genomes from short sequence reads.

1.1 Sequencing Technologies

A high level understanding of sequencing methods is essential to successfully developing a sequence assembler. For this reason, we outline the laboratory experiments that produce the sequence reads.

All sequencing techniques rely on three high level concepts. The first is replication. Each method relies on the ability to duplicate a DNA molecule use of biological enzymes or bacteria. The second is random shearing, or breaking DNA into smaller pieces called fragments. Scientists combine replication and shearing to create a *DNA fragment library.* Finally all methods offer the ability to read the end (or ends, as first described in [13]) of the randomly sheared DNA. With enough fragments, we expect most of an organism's DNA to be *covered* by these *sequence reads.*

Taken together, the above pattern is known as *shotgun sequencing* (illustrated in *Fig. 1.2*), perhaps because of its random and violent nature. When sequencing both ends of a DNA fragment, this technique has been playfully referred to as double-barreled shotgun sequencing.

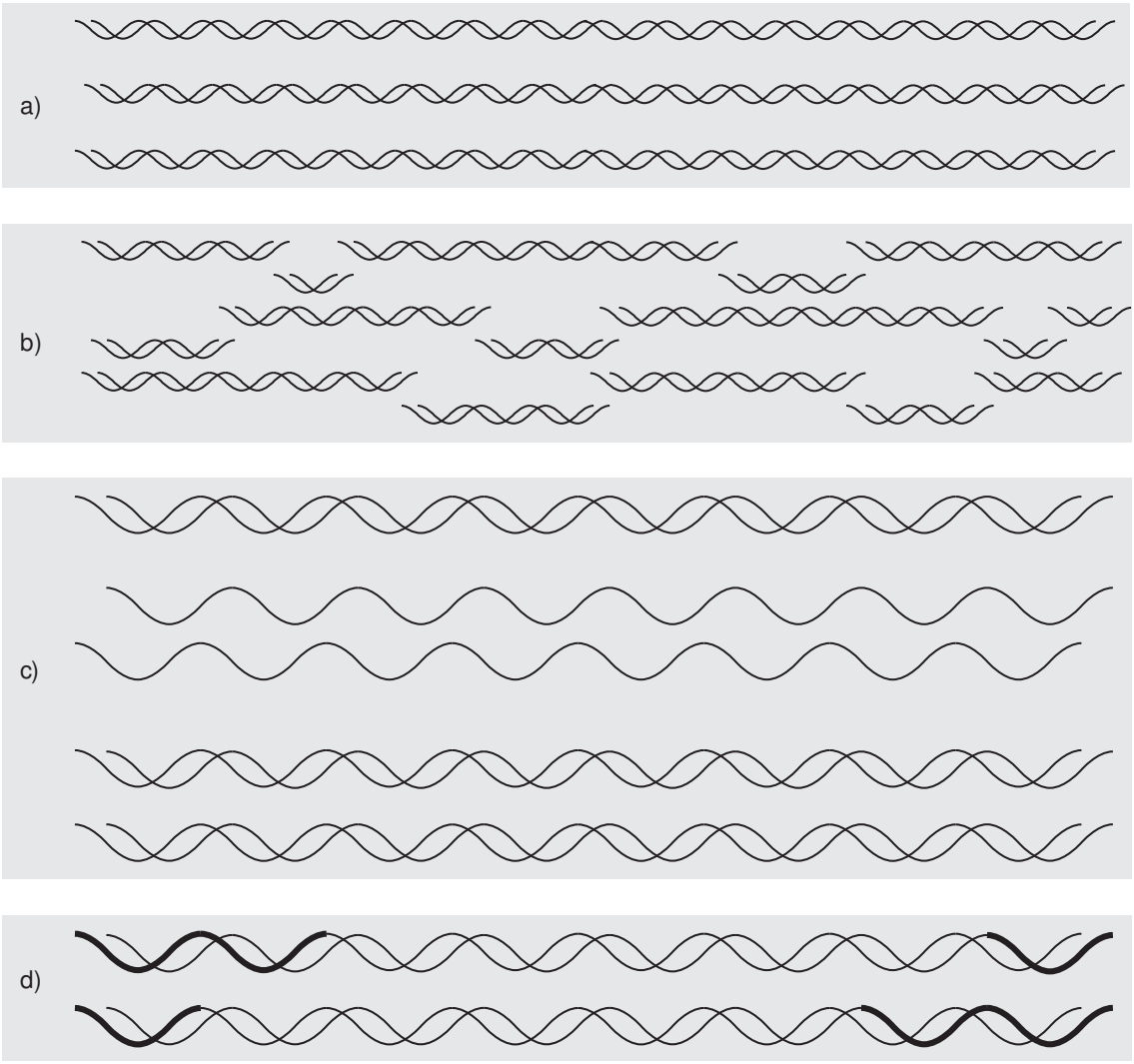

Figure 1.2 An overview of the shotgun sequencing process. In (a) we show many copies of the target genome. These are randomly sheared in (b). In (c) we show the duplication of a single sheared fragment. In (d) we show reading the ends of a fragment.

1.1.1 Chain Termination

The principal sequencing method used by experimental biologists during the decades of the 1980's and 1990's was sequencing by chain termination.[3] In this process, a piece of DNA is first exponentially duplicated in a process known as a polymerase chain reaction (PCR). The resulting copies are *denatured*,[4] the pool of denatured DNA separated into four bins, and each bin combined with a chemical cocktail that includes a number of important ingredients.

1. **a DNA primer:** The primer allows a copy of the DNA to be created by creating a starting point from which the polymerase will work.

2. **a DNA polymerase:** A DNA polymerase is a molecular machine that creates a copy of a strand of DNA by attaching itself to a template strand, and then incorporating nucleotides one at a time to match the template.

3. **deoxynucleotides (dNTPs):** dATP(deoxyadonine diphosphate), dCTP, dGTP, and dTTP. Mentioned previously, these are the molecules that naturally comprise the DNA polymer.

4. **dideoxynucleotides (ddNTPs):** Each bin receives only one of ddATP(dideoxyadonine triphosphate), ddCTP, ddGTP, and ddTTP. After being incorporated at the end of the polymer, these special nucleotides inhibit the polymerase's ability to continue extension.

When these ingredients are combined, the result is a set of DNA molecules separated into four bins such that all molecules within a bin end in the same base. The molecules within a bin are separated by size using, for example, gel electrophoresis. When all four buckets are separated, one can infers the DNA sequence by looking at the position of each group of molecules on the gel.

A variation on this method allows for sequencing using a single bin by coloring each of the four terminating molecules (ddATP, ddCTP, ddGTP, and ddTTP) with a different dye [52]. The dye-based method tends to be used in modern chain terminating sequencers, which

[3] Also known as Sanger sequencing after its inventor, Fredrick Sanger [53]
[4] The two complementary strands separated

automate the above process and produce reads of length 600-1000 bases. The software Phred interprets the laboratory data and produces the DNA sequence [15].

1.1.2 Pyrosequencing

The primary difference between short sequencing methods and the chain termination methods is the ability to measure the synthesizing reaction rather than having to measure the length of the synthesized result. In *pyrosequencing*, which is a type of *sequencing by synthesis*, the action of the DNA polymerase is measured in real time. This is done by including a chemiluminescent enzyme in the solution which emits a pyrophosphate (hence the name) when the polymerase incorporates a new deoxynuleotide triphosphate [51].

Unlike Sanger sequencing, in which all four dioxynucleotides are present in the reaction, only one of the four (dATP, dCTP, dGTP, or dTTP) are added to the solution at a time. Because of the presence of the chemiluminescent enzyme (for example ATP sulfurylase and luciferase), the incorporation of the dNTP molecule releases a visible light that is measured by a camera. The presence of apyrase in the solution quickly breaks down any dNTP molecules that were not incorporated, and the process continues with the next base.

Many such reactions are observed simultaneously on a single chip. For example, 454 life sciences produced a sequencer in which each piece of DNA is captured in its own well on the chip after being attached to an enzyme bead. Using a variation on PCR called emulsion PCR (or sometimes called mini-PCR) the DNA is duplicated within each well [38]. Using a centrifuge, the beads and attached DNA molecules are kept in place during the sequencing process, which involves repeated application of one of the four dNTP molecules and measurement of the resulting reaction. The parallel nature of this process means that this sequencing happens much more quickly and cheaply than chain termination sequencing. Currently, this technology can read 400-600 million bases (also called *megabases* or *Mb*) in ten hours. The technology continues to improve, and currently can produce 400 base unpaired reads at 99% accuracy.

The 454 sequencing platform can also produce paired reads at the cost of read length. Paired reads are reads that are separated by some known *insert length*, and are very important

for *de novo* assembly, as they allow for the reconstruction of repeated regions of the genome. For example, the GS FLX Titanium paired end adapter generates 140base paired reads, with insert lengths of 3kb (3 *kilobases* or 3,000 bases), 8kb, or 20kb. First, the target DNA is sheared and fragments of the appropriate size, for example 3kb, are isolated. Then a circularization adapter is attached to each end of each 3kb fragment. A ligase is used to to connect the two ends, creating a circular DNA molecule. This molecule is then sheared randomly into 400 base fragments, and the fragments containing the circularization adapter are isolated for amplification and sequencing. These 400 base fragments will have both ends of the original fragment on either side of the circularization adapter.

1.1.3 Reversible-Terminator Sequencing

The Illumina sequencing platform [3] and Helicos platform [70] also use a form of sequencing by synthesis, through a technology known as a reversible terminator. Unlike in Sanger sequencing, the sequence terminators used in the Solexa and Helicos technologies are reversible which allows, as in the 454 technology, a scientist to design a chemical process wherein bases are incorporated one at a time. As the reversible terminator allows incorporation to continue in a controlled way, the DNA sequence is read as it is grown.

The Illumina system uses a transparent chip. DNA is randomly fragmented and then attached to this chip. Once attached, a process known as bridge amplification (in which each new copy of DNA is attached to a nearby location on the chip) is used to create clusters of DNA on the chip, each cluster containing many copies of the initial fragment that attached itself to that location. Then, a solution containing four reversible terminators with attached florescent dyes is washed over the chip, allowing a camera to record which of the four bases were incorporated at each cluster location. A second chemical wash reverses the terminators and removes the dye, allowing the process to continue.

Illumina's latest sequencers generate paired 75 base reads with a total of 20 billion bases in a single run, which takes around a week including sample preparation. Ends of the fragment are read in much the same way one one copies a two sided piece of paper. The sequencer first

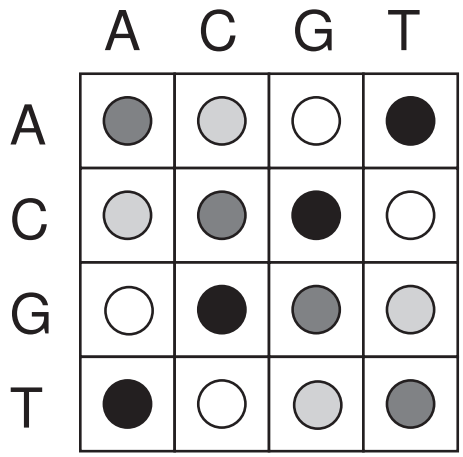

Figure 1.3 The mapping of each two base motif to the four dyes used in the SOLiD 2 system.

reads one end of the each DNA fragment during a normal machine run. Then the fragments are turned over (the opposite end attached to the chip at the same location), such that the machine can read the other end of the fragment. Thus, the Illumina system produces the same read length for both unpaired and paired reads, with paired reads coming at half the speed for twice the data. The platform produces paired reads with insertion lengths between 200 bases to 500 bases and 2 kilobases to 3 kilobases.

The Helicos system works similarly to the Illumina system, using different underlying technology [70]. The Helicos system incorporates one type of base at a time, eliminating the need for multiple dyes. This simplifies the data processing, allowing the system to process the reads in real time. The Helicos sequencer currently produces 25-35 base unpaired reads.

1.1.4 Sequencing by Ligation

The Applied Biosystems SOLiD 2 system also uses a sequencing by synthesis approach, but relies on ligase rather than polymerase to incorporate new molecules in the polymer [56]. As with the 454 sequencer, the first stage of the sequencing process involves attaching DNA fragments to beads, and then using emulsion PCR to amplify the DNA locally. In the SOLiD system, the beads are affixed to a chip for sequencing.

As in any sequencing by synthesis approach, the SOLiD system takes measurements as new bases are incorporated into the polymer. However, the method for incorporation is fundamentally different from the other sequencing methods, which use a polymerase. Instead, the SOLiD system uses a ligase to attach an eight base probe to the end of each polymer. This eight base probe has two bases that must align with the template sequence being read and six junk bases. Thus there are 16 differentiable binding motifs for the probes. Each motif is associated with one of four possible dies. The mapping of the four dyes into the 16 motifs is shown in *Fig. 1.3*.

Sequencing proceeds in a complicated way, but the end result is that a measurement is taken for at each base in fragment. First, a primer is attached to the base of the bead, of total length n. An eight base probe is incorporated, and a color measurement taken. At this point the last three junk bases are cleaved, leaving five bases attached. Ligation continues, offset five bases from the previous incorporation point. We incorporate seven probes in this manner for a length 35 read, at which point the DNA is denatured and the probes washed away.

In order to measure a color for each position in the read, we repeat the above process five times, each time reducing the size of the primer by one base. For example, in the second round, we use a primer of length $n - 1$. Thus the color measurements for the sequence of bases are are interleaved between the five rounds, with each color corresponding to one of four possible two-base motifs. Because of the chosen mapping function, given a starting DNA base, we can deterministically translate the measured sequence of colors into a sequence of bases.

Applied Biosystems claims that the quality of the sequencing by ligation method is higher than for sequencing by polymeration for two reasons. First, ligase is less likely than polymerase to introduce an incorporation error. Second, each base is measured twice because of the tiled color measurement.

1.2 The Sequence Assembly Problem

While many attempts have been made to formally define the assembly problem, such attempts invariably give way to heuristic methods. In fact, many formulations of the problem

(for example shortest superstring, Eulerian superpath, and string graph-based approaches) have been shown to be NP-hard [17, 41], as has the simpler problem of deciding from which of the two strands each sequence was read [35]. In practice, the properties of real data allow us to find good assemblies despite these results.

The shotgun assembly problem has, as its input, a large set of *reads*. As described above, sometimes the data are pairs of reads from complementary strands with an approximately known intervening distance, or *insert length*. The insert length is known because we only include fragments of a chosen size during library creating and approximate because this selection is inexact. In the presence of paired reads, we expect that the assembled genome conform to the distance constraints. The *coverage* of the genome by the data is the ratio of the total length of all reads to the total length of the genome. Assembly projects using chain termination methods of sequencing have typically generated 6-fold to 20-fold coverage.

The output of an assembly method is a set of reconstructed DNA molecules that were likely to have been the template for the reads. The problem is difficult because the genome of an organism contains repeats of various lengths, frequencies, and similarities. Also, the reads are not perfect; they contain errors that need to be identified and corrected.

1.2.1 The Overlap-Layout-Consensus Method

Traditionally, Sanger sequencing projects have relied on a heuristic assembly method known as the overlap-layout-consensus method. In this method, *overlaps* between reads are used to guide the the assembly. In the absence of sequencing errors, one would expect that two reads taken from overlapping spans of the genome would share a common substring (some prefix of one string would exactly match some suffix of the second string). In the presence of sequencing errors, such matches are inexact, and a sequence alignment method is used to produce a score measuring the quality of the overlap [32]. In practice, the overlap scores are not computed between all pairs of reads, but only those reads that pass some initial test, for example those containing shared substrings of length k [2], and then the alignments are calculated using a seed and extend approach similar to that used in [1].

Figure 1.4 The overlap graph for BAC 238O23 from the *Zea mays* genome sequencing project, with nodes drawn using a force directed graph layout algorithm. Unitigs are long consistent chains of overlaps in the graph.

An overlap graph is a graph in which each node corresponds to a read, and an edge occurs between two nodes if and only if a good overlap exists between the corresponding reads [35]. An overlap is good if it is sufficiently long and its alignment score is high. *Fig. 1.4* shows an example of a force directed graph layout algorithm[5] applied to the overlap graph of a BAC[6] from the maize assembly project, created as part of the work we did on validating that assembly. The example shows how the existence of long repeats in the genome complicates the assembly problem. One sees multiple *unitigs* (defined as the "maximal interval subgraph for which there are no conflicting overlaps to an interior vertex"[43]) in the graph converging at a number of junctions. This convergence happens because a sequence longer than the read length ($>$ 1000 bases in this case) is repeated in the BAC.

In practice, the overlap graph is not actually built for whole genome shotgun assembly projects. With around three billion bases in large genomes, a sequencing project with 10-fold coverage and 800 base average read length results data comprised of around forty million reads and thirty billion characters. Thus, working with the entire graph is not practical on a uni-processor machine. Instead, the graph model has inspired many heuristic methods used in sequence assembly such as the TIGR [63], Celera [45], Phrap [19], CAP3 [24, 25], Atlas[20], and ARACHNE [2, 33] assemblers. There are many differences in the details of these assembly methods, but they each proceed with the same general structure, and I will describe the overlap-layout-consensus method by taking examples from each.

Starting with the best overlapping sequences, contiguous assembled regions (*unitigs* or, more commonly, *contigs*) are extended by adding new reads one at a time. In the absence of the complications introduced by repeats, this extension would be very straightforward; one could start with a good overlap and then continue by finding reads that extend the contig by examining overlaps with reads that have already been incorporated. The presence of repeats complicates this process. One way the Celera assembler handles repeats is by masking them,

[5] A force directed graph layout algorithm treats nodes in the graph as entities that repel each other with some force and treats edges as springy constraints, and then tries to find an energetically stable layout of the graph.

[6] BAC stands for bacterial artificial chromosome. For a short description see *Section 1.2.3* on hierarchical sequencing.

by either creating a library of known repeats and removing all reads that align to the repeats in this library, or by identifying reads with many good alignments as repeats [45]. The ARACHNE assembler [2] watches for branching points to identify the boundaries of repeats. A branching point occurs when read A aligns with reads X and Y, but X and Y do not align with each other. The CAP3 assembler [24] uses paired read information to break apart contigs that had been inappropriately combined due to repeats.

Paired reads are often used in these methods only after the initial contigs are created,[7] to find an ordering of the contigs produced in the first step. This ordering is called a scaffolding or supercontig. The ARACHNE assembler, as an example, first identifies which contigs correspond to unique regions,[8] and then uses clone pairs to create a scaffolding of these unique regions. Contigs from the repeat regions of the genome are then placed in the gaps of the scaffolding. Ultimately, overlapping contigs are merged to produce the final assembly. The ARACHNE II assembler presented an even more sophisticated means of using the paired reads, including a method of ordering of multiple repeat contigs between unitigs and breaking apart missassembled contigs [33]. The TAMPA tool allows one to analyze the quality of the assembly by using the clone pairs to detect insertions, deletions, inversions, and transpositions [11].

Once the sequences are placed in their approximate location on the genome (the layout phase of these methods), a consensus sequence is inferred. The CAP3 sequencer uses a multiple sequence alignment to align each character in each read to a sequence of columns, one column for each position on the genome. It generates the consensus sequence by finding the consensus character for each column [24], at the same time deciding if a difference seen in a column is due to error or due to a true difference on the genome.[9]

[7] In the ARACHNE assembler, paired read information is used to create super reads as a first step. Two pairs of reads are merged if the component reads from both ends of the fragment overlap.

[8] In the ARACHNE paper, the authors term contigs corresponding to unique regions of the genome *unitigs*. Myers *et al.* [45] call any contig a *unitig* and a unique unitig a *U-unitig*. Butler [6] *et al.* use Myers' choice when naming an assembly graph a *unipath graph*.

[9] Small differences between the two chromosomes in diploids, or small differences between two copies of the same repeat could cause a disagreement in a column.

1.2.2 Approaches Using Graph Traversal

With short reads eliminating the reliability of read overlaps in predicting genomic co-location, a revival of graph-based methods has underpinned the development of short-read assemblers. These graph-based methods permit a more global view when resolving repeats. While these methods predate short read technology, their reach has not extended significantly beyond bacterial genomes due to the memory resources required in their use. These memory limitations are exacerbated by the high coverage needed to compensate for shorter read lengths. As a result, prior to our work, short-read *de novo* assembly has been demonstrated on relatively small genome sizes, ranging from single BACs to bacterial genomes with a few million bases.

1.2.2.1 The Fragment Assembly String Graph

Myers [44] proposed converting the overlap graph to a fragment assembly string graph, or a graph in which the edges are labeled with strings. His ideas were expanded by Medvedev and Brudno in the development of a short sequence assembler [40]. Taking as its basis the overlap graph, Myers' string graph is quite different from the k-string graph we describe in *Chapter 2*, which has as its basis the de Bruijn graph. On the other hand, in a more general sense, a fragment assembly string graph is a graph in which the concatenation of edges of some graph tour corresponds to the underlying genomic sequence. For that reason, we call Myer's graph the overlap string graph. Both the overlap string graph and the k-string graph are *bidirected* graphs in which a single tour of the graph corresponds to both strands of the DNA.

To construct an overlap string graph, we create two nodes for every read. We remove from the graph any nodes corresponding to reads fully contained in some other read in the data. Next, we draw two directed edges in the graph for each overlap between reads, labeling each edge with the suffix not part of the overlap the corresponding sequence direction. We find transitive reduction of this graph,[10] and we convert the graph to a bidirected graph, a graph in which each edge has two directions, one for each incident node. We describe the bidirected graph model for assembly in *Chapter 2*.

[10] Myers proposes a linear time algorithm in the number of edges.

Both Myers and Medvedev *et al.* propose that the assembly should be a tour of the overlap string graph that conforms to some traversal constraints. They generate these constraints by calculating an expected traversal count for each edge using the coverage information in the data. Myers formulated this as a minimum cost network flow problem, later shown to be NP-hard [41] by reduction from Hamiltonian cycle. Medvedev *et al.* propose instead modeling the assembly problem as a convex min-cost biflow, with variations allowing for both minimum and maximum edge traversal bounds. In doing so, they adapt a known convex min-cost biflow algorithm to work with bidirected graphs. Both formulation as min-cost and convex min-cost biflow allow a good assembly to be found without relying on paired reads. However, these methods are heavily reliant on uniformity in coverage along the length of the genome in order to correctly assign traversal constraints to each edge.

In addition to modeling the assembly problem as a flow problem, Medvedev *et al.* also described the *conflict graph* and the set of transformations that give rise to it [40]. The conflict graph is the graph in which every edge with multiple incident edges has both multiple in and multiple out edges. We make use of the conflict graph in our transcriptome assembly work, but we also show that one of the graph transformations proposed in [40] may invalidate the string graph property; after applying this rule the concatenation of edge labels for some tour of the graph may no longer correspond to the genome.

1.2.2.2 De Bruijn Graphs

Pevzner *et al.* [50] used the following formulation of sequence assembly, expanding on a model proposed by Idury and Waterman [26], which in turn was a conceptual successor of the sequencing by hybridization approach to genome assembly. Each node in the graph corresponds to a unique *k-mer* (length k string) present in some input sequence or its reverse complement. A directed edge connects two nodes labeled $a\alpha$ and αb, where α is a string of length $k-1$, if and only if $a\alpha b$ is present in some read. This graph is a subgraph of a de Bruijn graph of k-mers, and each input sequence a path.

The genome assembly problem becomes that of finding the shortest tour of the graph

that includes each sequence path. While finding a tour of the graph is polynomially solvable, Medvedev *et al.* showed that the superpath problem is NP-hard by reduction from the shortest superstring problem *et al.* [41]. The superstring problem, which asks to find the shortest string that has all input strings as substrings, has itself been proposed as a model for the assembly problem, but it is unsuitable due to the abundance of repeats in genomes. This problem was shown to be NP-hard in [17].

Directed de Bruijn graphs have been used extensively in short sequence assemblers including [12], [67], [71], [7], and [22]. Butler *et al* present an assembler that uses a graph that is *nearly* a string graph based on the directed de Bruijn graph [6].

Medvedev succinctly describes the bidirected de Bruijn graph model. To our knowledge, other than the work described in the previous section and our work, no other modern short read assemblers use a bidirected model. The bidirected k-string graph and bidirected de Bruijn graph are described extensively in *Chapter 2*. After we specify the k-string graph, we shall revisit how it relates to other graph models and explain why we chose it as the basis of our assembler.

1.2.3 Hierarchical Sequencing

We are concerned with what is known as *whole genome* shotgun sequencing. The input sequences can come from any genomic DNA; other than the paired read information, there is no structure to them. In fact, we do not even know which of the chromosomes each sequence comes from *a priori*. As all reads are in the same pool, the problem of whole genome shotgun sequencing is more difficult than what is known as *hierarchical sequencing*.

Traditional hierarchical sequencing requires us to create what is known as a BAC library. First we break the DNA into longer 100,000 to 400,000 base sequences. These sequences are incorporated into what is known as a Bacterial Artificial Chromosome (BAC) which is in turn inserted into an *E. coli* bacteria. When the bacteria reproduces, it also replicates the BAC. Each colony of bacteria is then processed to extract the many copies of the original DNA fragment, which is randomly sheared and sequenced, as described above. We select which

BACs to sequence by finding what is known as a BAC tiling path that covers the genome, using genetic and physical maps.[11]

Sundquist *et al.* [62] propose the SHRAP hierarchical short sequencing protocol and method for assembling hierarchical short read data in parallel. They break the genome into a number of large pieces that they call clones. These large pieces cover the genome at relatively high coverage (7-fold to 10-fold). Reads are taken from the clones, marked with a clone identifier, at relatively low coverage (1.5-fold to 2.5-fold). All reads are unpaired; they show the method to work well on long (length 200+) reads, which cannot be generated in pairs by current short read technologies.

Assembly proceeds in four major steps. A tiling of clones is inferred by looking at all overlaps between pairs of reads. Reads are grouped into overlapping sets to assemble, pooling data from multiple clones. Contigs from step two are treated as reads and assembled using the same process. The larger newly assembled contigs are scaffolded using the clone tiling.

Hierarchical sequencing is a useful paradigm. It mutes the impact of repeats, as only repeats within a particular BAC cause assembly issues, and it allows for a natural decomposition of the computational problem: each region of DNA assembled independently and then combined.

At the same time, hierarchical sequencing comes with a price; it adds complexity and cost to the experimental design. This was especially true for the traditional BAC by BAC assembly model described above. As far as we are aware, the SHRAP protocol has not been validated on experimental data or taken up in sequencing projects. Because of this drawback, whole genome shotgun sequencing remains undisputed as an important and challenging research area.

1.3 Method and Software Organization

To impose structure on our description of the presented parallel methods, we decompose them into a sequence of permutations and transformations of arrays of tuples. Each permutation achieves a partitioning of tuples based upon some key by bringing together all elements with the same key into consecutive array positions, which we call a bucket. Each transfor-

[11] A genetic an ordering of markers along chromosomes. In a separate piece of work, we conceived novel methods for finding consensus genetic maps from multiple sources [28, 29].

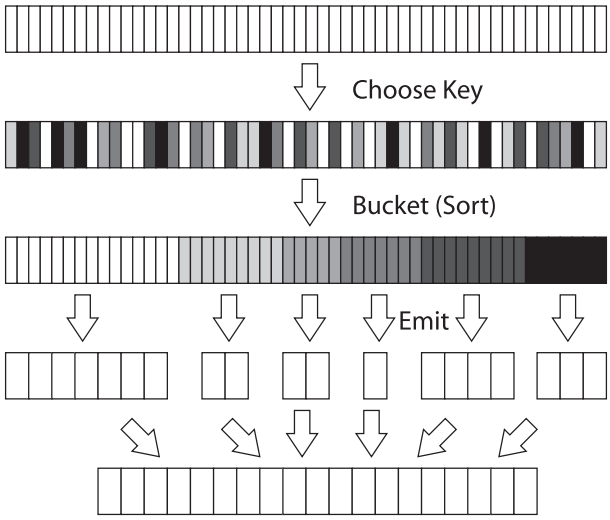

Figure 1.5 A diagram of the application of a permutation and transformation of data. We organize our method around this computational concept.

mation involves processing elements within a bucket and emitting elements of a new type. In general, we allow two buckets from separate tuple arrays with a common key to be processed by a single emitting function, resulting in the following two possible notations for a permutation followed by a transformation.

$$\langle b_1, b_2, \ldots \rangle \quad \xleftarrow{\textbf{Function}} \quad \mathcal{A} : \langle \mathbf{a_1}, a_2, \ldots \rangle$$

or

$$\langle c_1, c_2, \ldots \rangle \quad \xleftarrow{\textbf{Function}} \quad \begin{array}{l} \mathcal{A} : \langle \mathbf{a_1}, a_2, \ldots \rangle \\ \mathcal{B} : \langle \mathbf{b_1}, b_2, \ldots \rangle \end{array}$$

Bold fields are used to identify the keys that define the partitioning scheme. Thus in this notation, we require the partitioning of tuples to be based upon a subset of fields within the tuple. While our implementation allows for arbitrary partitioning of tuples, this flexibility is not needed for the description of the assembly method, and we feel that the above notation is clean. If a partitioning of a single array results in each tuple belonging to singleton partition, no fields will be written in bold to emphasize this fact. In this case, any permutation of the data achieves the required partitioning.

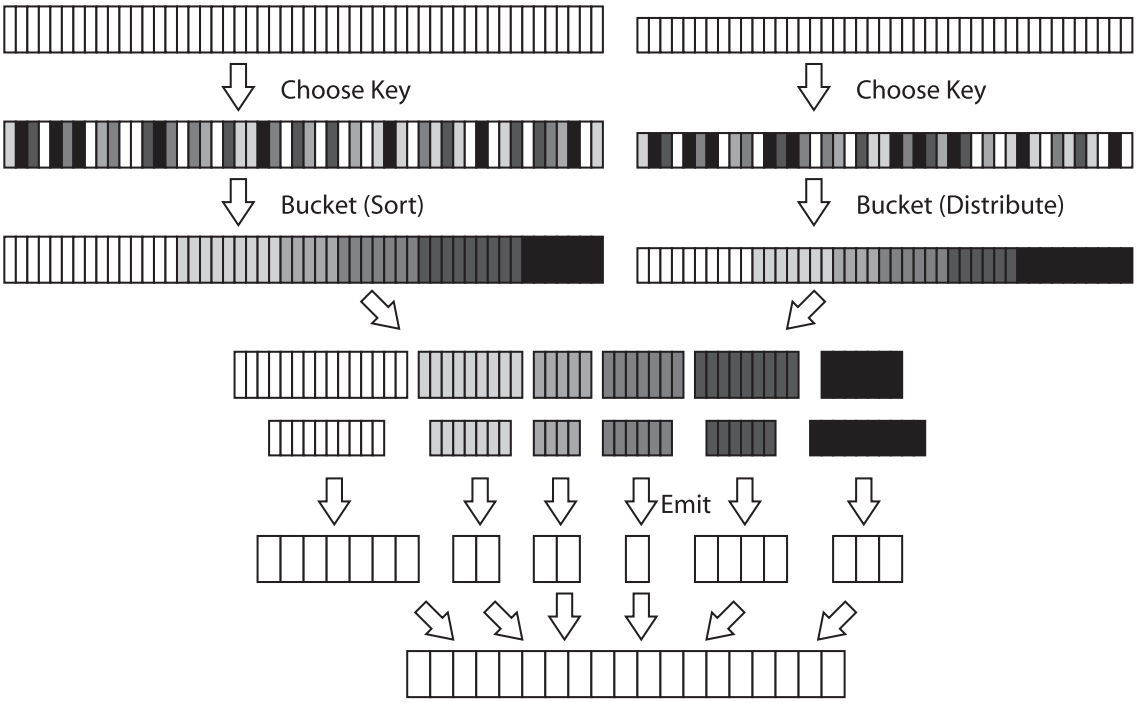

Figure 1.6 A diagram of the bucket-emit logic when combining two collections.

The transformation is performed by the named **Function**, this function having access to tuples within a bucket (or pair of buckets) and emitting zero or more tuples of a new type.

The notation above will be reduced to the following more natural notation within the context of a higher level algorithm. The types of the various collections of tuples will be clear from context.

$$\mathcal{B} \quad \overleftarrow{\textbf{Function}} \quad (\mathcal{A})$$

or

$$\mathcal{C} \quad \overleftarrow{\textbf{Function}} \quad (\mathcal{A}, \mathcal{B})$$

A well known implementation of this pattern is Map Reduce [8], a library developed by Google™ for processing large data sets. Hadoop is an open source implementation of the pattern. This bucket-emit pattern is useful because it reduces the method design to finding a sequence of such operations, without worrying about the underlying library implementation,

effectively encapsulating the parallel details.

In addition to the permute-transform operation, for optimization purposes we describe a second transformation in which we map each key to a unique integer identifier in the range $[0, n-1]$ where n is the number of unique keys (and therefore buckets). This operation allows us to reduce the space requirements of our method by replacing arbitrarily sized keys with integers.

$$\mathcal{A} : id \quad \overleftarrow{\textbf{Assign}} \quad \mathcal{A} : \textbf{key}$$

Because we describe our method using these general computational structures, our assembler can be adapted to many targets – including cloud architectures, heterogeneous architectures, serial architectures, and disk-based architectures – by replacing a few library elements. We will describe two such adaptations: one targeting a single processor and a second targeting a distributed memory high performance parallel computer.

1.3.1 Serial Implementation

A serial implementation of the pattern is very simple. Tuples are stored in a single array. We then permute the elements by sorting this array by some key specified by the user. The resulting buckets are then processed serially in this sorted order. In the case of two input arrays, both are sorted using keys specified by the user that are comparable. The buckets are processed together through a single scan of the two arrays, advancing pointers for each arrays in the same pattern used for merging two sorted lists. Finally, we specify the assign function, in which we step through the array, assigning IDs to keys in sorted order.

1.3.2 Parallel Implementation

We now outline our implementation of the communication patterns above on distributed memory parallel machines. We will describe the parallel model and the all-to-all and many-to-many collective communication primitives used as a basis for parallel sorting by regular

sampling. Finally, we will outline how the patterns above are achieved using these basic operations.

To ensure practical applicability, we use the distributed memory model of parallel computation. Let p denote the number of processors on the parallel machine, with each processor given an integer identifier in the range $[0, p-1]$, with the i^{th} processor denoted as p_i. Each processor has access to its local memory, and remote memory access is achieved through communication over an interconnection network. A parallel algorithm in this model combines local computation and message passing between the processors.

In our computational framework, the n tuples in the collection are distributed evenly among processors, $\frac{n}{p}$ tuples per processor, stored in an array. To achieve the permutation of elements based on their keys, we make use of the *all-to-all* communication pattern, in which each processor p_i sends a distinct message size $O\left(\frac{n}{p^2}\right)$ to every other processor p_j. All-to-all functionality is provided by the Message Passing Interface (or MPI), a standard parallel programming environment used for programming high performance computers. It is known as a collective communication function. The specific implementation of this pattern varies and can be optimized for the underlying communication network.

We can choose, for instance, to think of the parallel communication time of all-to-all on the permutation network model,[12] in which each processor can simultaneously send/receive a message of m bytes in a single communication round, provided no two source/destination processors have the same identifier. The cost of a communication is given by some startup latency factor τ and some transfer cost multiplied by the message size $m\mu$. Given this model, the all-to-all pattern can be achieved in $O\left((\log p)\tau + \frac{n}{p}\mu\right)$ communication time using a butterfly communication pattern, or in $O\left(p\tau + \frac{n}{p}\mu\right)$ in p permutations with a lower constant factor.

In practice the messages we wish to send are not uniform in size as is required in the description above, and we instead use a *many-to-many* communication pattern, which is similar to all-to-all with the difference being that each processor sends and receives variable sized chunks of data to and from every other processor. A bounded many-to-many communication

[12] It is useful to reason about parallel algorithms on the permutation model as we can embed this model in many practical network topologies.

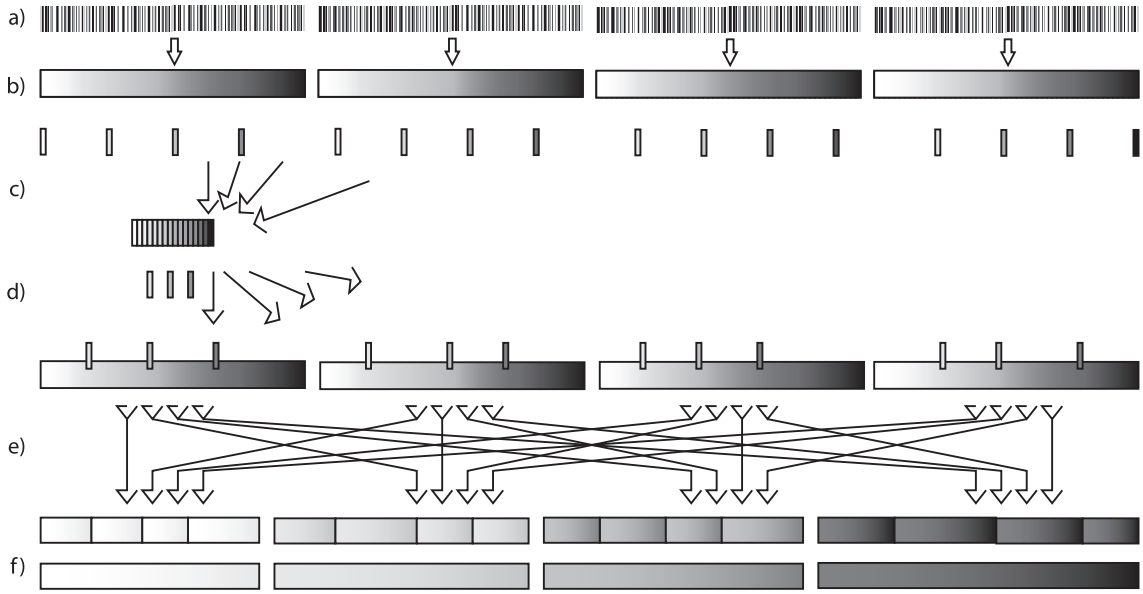

Figure 1.7 A diagram of the regular sample sort algorithm. We show in a) the original data, distributed between 4 processors. In b) we show the result of sorting the data locally. In c) we gather samples from each processor to a single location. In d) we choose splitters and broadcast these to every processor. In e) we divide the locally sorted data according to the splitters, sending and receiving between all processors using a many-to-many collective communication primitive. In f) we merge the data received in d).

decomposes into two regular all-to-all communications with total size $r + s$, where s is the maximum number of elements initially sent by any processor and r is the maximum number of elements finally received by any processor [55]. In practice, however, the many-to-many collective communication function provided by the MPI standard library is often implemented using the same communication structure as a single all-to-all.

We use two higher level parallel algorithms when implementing the bucket and emit framework — parallel sorting and parallel redistribution.

The best algorithm for parallel sorting on distributed memory machines that achieves a good final distribution of the sorted values is regular sample sort [57, 37]. In a simple regular sample sort (shown in *Fig. 1.7*), data is sorted locally, samples are taken from the data at

regular intervals, these samples are gathered to a single processor, merged, and then $p-1$ *splitters* are chosen. These splitters define the range of sorted values for which each processor is responsible. Each processor divides its locally sorted lists into p buckets, and these are then distributed using a many-to-many pattern. Finally, all p received buckets are merged (generally resorted) on the destination processors. As the maximum number of elements any processor receives is bounded by $\frac{2n}{p}$ as long as at least p^2 samples are taken, regular sample sort uses a constant number of bounded many-to-many communications and $O\left(\frac{n}{p}\right)$ local computation for integer sort, and $O\left(\frac{n}{p}\log\frac{n}{p}\right)$ local computation time for comparison sort.

The possible load imbalance after sort might seem unacceptable, but in practice it does not come close to reaching the $\frac{2n}{p}$ proven bound. [21] gives a regular sample sorting algorithm that achieves even better load balancing guarantees but requires both a regular all-to-all communication and a highly balanced many-to-many communication. In practice, the benefit of good load balancing does not outweigh the cost of the more complicated algorithm. Instead, we periodically rebalance our arrays directly using a bounded many-to-many communication after sorting.

In the case of one tuple array being processed, each processor steps through its locally sorted array, passing each bucket into the emitting function and storing the results. In the context of a larger method, the resulting array of tuples will be sorted by some key in a subsequent bucket-emit phase.

In the case of two tuple arrays being processed together, the first array is sorted in parallel as previously described. The tuples from the second array are sorted locally. Then, using the last element from the first array in the first $p-1$ processors (giving $p-1$ splitters), the tuples in the second array are divided into p buckets per processor, each bucket containing keys to be handled by different destination processors. These buckets are then distributed using the many-to-many communication pattern, before finally being merged or sorted at the destination processor. After sorting and distribution, buckets from the two lists sharing the same key are passed into the processing function, using a single linear scan of the two arrays on each processor.

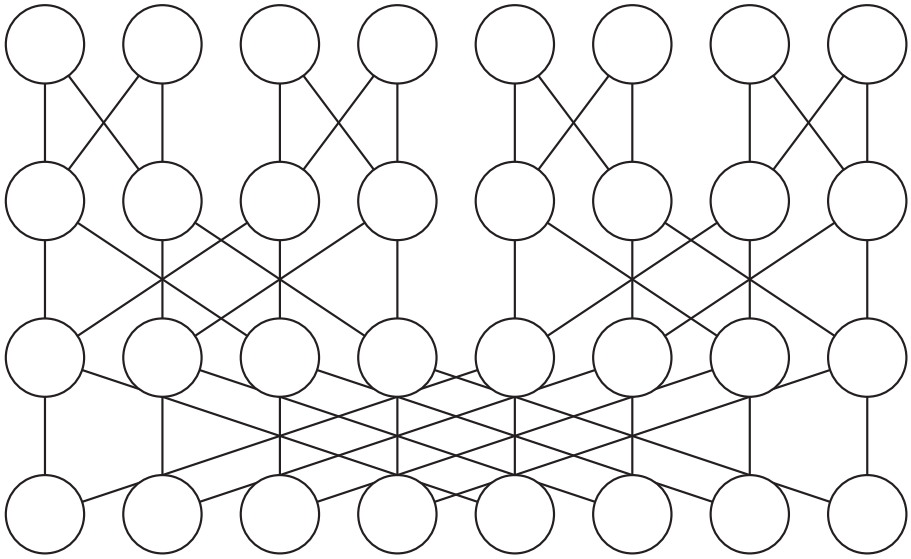

Figure 1.8 A diagram of the butterfly pattern sometimes used when implementing the prefix-sum computation on parallel machines. Each column corresponds to a processor, and each row corresponds to a communication step the in pattern. Edges correspond to communication between processors.

To implement the **Assign** function, we use what is known as a *prefix sum* operation. Each processor p_i has a value v_i, and we wish to perform a summation over values using a binary associative operator $\oplus$. After the summation, we require that processor p_i know the partial summation $v_0 \oplus v_1 \oplus ... \oplus v_i$. Assuming the result of the operator $\oplus$ is of the same size as the operands, a prefix sum is calculated in $O\left(\log p(\tau + |v_i|\mu)\right)$ parallel communication time under the permutation network model using a *butterfly pattern* as shown in *Fig. 1.8*. Some machines, such as the IBM Blue Gene /L have special communication networks specifically designed to handle computations like prefix sum. MPI provides a collective communication function (which it names *Scan*) that provides built in prefix sum functionality for basic data types and common operators like add, multiply, bitwise and, bitwise or, minimum, and maximum.

To assign unique identifiers to each key, we first sort all elements based on the key of interest in parallel. Then we count the number of unique keys present on a single processor p_i by scanning the local array and storing it in variable c_i. We perform a prefix sum using the

addition operator, recording value s_i. Finally, processor p_i, starting at identifier $s_i - c_i$, assigns unique identifiers to each key in its local array from the range $[s_i - c_i, s_i - 1]$, through a single traversal of the array.

1.4 Contributions

In this work, we make a number of important contributions to the research area of genome assembly. We present a short-read sequence assembly framework that we use to successfully assemble large genomes with high coverage. We have developed parallel methods and used the large distributed memory present in high performance parallel computers to handle the memory-intensive phases of the assembly.

We present parallel methods for constructing a bidirected de Bruijn graph of k-molecules and converting this graph into a bidirected string graph. Input to the algorithm is a set of reads of total length n. The algorithm outputs a list of edges representing the graph distributed across processors.

We present methods for assembling transcriptomes. We describe a parallel method for constructing what is known as a conflict graph by processing sets of independent nodes in the graph. We have conceived methods for simplifying the graph (and thus improving assembly) by using the differential coverage manifest in these data sets.

We have conceived of a feature of paired read data which we term a $(k+1)$-pair cluster. We present a parallel method for computing these features and an algorithm that finds paths corresponding to contigs by starting with unambiguous long edges as seeds, and then extending each path using heuristic rules that rely on the computed features. We present the first assembly method designed to flexibly incorporate any number of insertion lengths as a first principle. We compare the features of our short sequence assembler with other assemblers in the field in *Table 1.1*.

We present experimental results of our assembly method demonstrating is validity. We validate our assembler on an experimental data set taken from *E. coli* and compare the results to other short sequence assemblers. We also measure the scaling performance of the software

	Whole Genome	Error Correction	Paired Reads	Single Length	Multiple Lengths	Bidirected Model	Parallel [13]
OurName	X		X	X	X	X	X
ABySS [60]	X		X	X			X
Medvedev [40]	X		X	X		X	
Myers [44]	X					X	
ALLPATHS [6]	X	X	X		X[14]		
Euler-SR [7]	X	X	X	X			
Velvet [71]	X		X	X			
Shorty [23]	X	X	X	X			
Sundquist [62]		X					X
SSAKE [67]	X						
Edena[22]	X						

Table 1.1 A comparison of short sequence assemblers by functionality as has been described in the cited papers. [13]We do not record parallelism at the node level (effectively using both cores in a dual-core processor), but rather large scale parallelism across nodes. [14]The ALLPATHS assembler requires that the data contain exactly three types of paired reads: a short insert length (for example 250 bases), a medium insert length (for example 2000 bases), and a long insert length (for example 10,000 bases).

using this data set. Using synthetic data, we demonstrate a 99.9% accurate assembly of the *Drosophila Melanogaster* genome in four hours, with 50% of the genome covered by contiguous assembled regions of length at least 102Kb.

CHAPTER 2. CONSTRUCTION AND COMPACTION OF k-STRING GRAPHS

A *sequence assembly string graph* is a graph in which edges are labeled with strings and for which the concatenation of edge labels on some path corresponds to the genomic sequence. To solve the assembly problem using a sequence assembly graph, we must find this path. A *bidirected* sequence assembly string graph is a string graph in which each edge has two labels, one for each direction of traversal. The bidirected string graph is a natural model for the assembly problem because the two string labels correspond to the two strands of a DNA molecule. Specifically we are interested in the bidirected string graph that is the lowest order graph homeomorphic with the subgraph of the *bidirected de Bruijn graph* defined by the spectrum of $(k+1)$ length molecules taken from the genome, which we call the *k-string graph*.

This model is closely related to many of the models other researchers have proposed, and it is reasonable to say that all of these models are functionally equivalent in that finding the assembled genome is equivalent to finding a path in the graph. In introducing the model, we will first describe the bidirected de Bruijn graph, which has been concisely described by Medvedev *et al.* [40], and is inspired by the work of Idury and Waterman [26] and others. Then we will describe how we transform this graph into the k-String graph. After specifically laying out our model, we will compare and contrast it with prior models, before finally describing its representation and construction using the bucket-emit paradigm presented in the introduction.

2.1 The Bidirected de Bruijn Graph

For a string α of length $|\alpha| = l$, we denote the i^{th} character of α as $\alpha[i]$, $1 \leq i \leq l$. We denote the substring from $\alpha[i]$ to $\alpha[j]$, inclusive, as $\alpha[i, j]$, $1 \leq i \leq j \leq l$. A *DNA strand* is

a string with alphabet $\Sigma = \{a, g, c, t\}$. We call the characters of a DNA strand *bases*. The *complement* operation on a base $\alpha[i]$, denoted by $\alpha[i]'$, is defined by the following bijection of Σ onto Σ: $\{t \to a, c \to g, a \to t, g \to c\}$. The *reverse complement* operation on a DNA strand α, denoted by α', is the operation of reversing α and complementing each base ($\alpha'[i] = \alpha[l-i+1]'$). Note that $\alpha[i] = \alpha[i]''$ and $\alpha = \alpha''$.

A *DNA molecule* is a pair of complementary DNA strands, $m = \{\alpha_m, \alpha'_m\}$. We denote the length of m as $|m| = |\alpha_m| = h$ and call m an h-molecule. We designate the lexicographically larger of the two strands as the positive strand, denoted m^+, and the lexicographically smaller of the two strands as the negative strand, denoted m^-. We choose the ordered tuple $m = \langle m^+, m^- \rangle$ as a canonical representation of the molecule. Note that because we can find m^- from m^+ using the reverse complement operation, we represent a molecule using only m^+. For this reason we also call m^+ the *representative* strand. If $\alpha_m = \alpha'_m$, then $m^+ = m^-$ and we say that the molecule is a *reverse complement palindrome*. Note that a molecule cannot be a reverse complement palindrome if its length is odd.

A sub-molecule $m_{[i,j]}$ of molecule m is the molecule $\{m^+[i,j], m^-[|m|-j+1, |m|-i+1]\}$. We denote the positive strand of a sub-molecule as $m^+_{[i,j]}$ and the negative strand as $m^-_{[i,j]}$. Note that either the substrand $m^+[i,j]$ or the substrand $m^-[|m|-j+1, |m|-i+1]$ might be $m^+_{[i,j]}$.

We say that we *extend* a molecule by adding a base to the end of one strand and the complementary base to the beginning of the other strand. If we add a base to the end of the positive strand, we call this operation a *positive extension*. We denote a positive extension as mc, where c is the base appended to the positive strand. If we at the same time remove a base from the beginning of the positive strand and the end of the negative strand, this operation becomes a *positive shift*, denoted $\overrightarrow{mc}$. Correspondingly, we denote a *negative extension* as cm, where c is the base appended to the negative strand. If we at the same time remove from the beginning of the negative strand, we call this operation a *negative shift*, denoted $\overleftarrow{cm}$.

A *bidirected graph* is a graph $G = \{V, E\}$ where edges are of the form $(\langle u, d_u \rangle, \langle v, d_v \rangle)$, where $d_i \in \{out, in\}$ is the direction of the edge at end point i. A traversal of a bidirected

Figure 2.1 The four ways in which nodes are connected in the bidirected de Bruijn graph, with the edges labeled with the initial characters in the bidirected k-string graph (before edge compaction).

graph must respect edge directions at each node – if entering on an out direction, one must exit on an in direction, and vice versa.

In a *bidirected de Bruijn graph* $G_d = \{V_d, E_d\}$, (see *Fig. 2.1*), the set of nodes in the graph are all possible molecules of length k, $|V_d| = |\Sigma|^k$. An edge $(\langle u, d_u\rangle, \langle v, d_v\rangle) \in E_d$ if and only if there exists a shift that transforms the molecule corresponding to u to the molecule corresponding to v. Notice that $|E_d| = |\Sigma|^{k+1}$. If u is transformed to v by a positive shift, then $d_u = out$. If u is transformed to v by a negative shift, then $d_u = in$. The arrow head at v is defined similarly. We show in *Fig. 2.1* the four possible edge types in the bidirected de Bruijn Graph. Each edge in the de Bruijn graph corresponds to a $(k+1)$-molecule.

The *genomic DNA* of an organism is a set of molecules $\mathcal{M} = \{M_1, M_2, ...M_c\}$. A *read* is a sub-molecule taken from this set, $M_{h[i,j]}$, $1 \leq h \leq c$, $r_{min} \leq j - i + 1 \leq r_{max}$, where r_{min} and r_{max} are the minimum and maximum possible read lengths determined by the experimental process. Current sequencing methods read only one strand of the DNA molecule, and from this strand we construct the canonical representation. We expect that reads contain errors, but this complication will be ignored for now.

An *edit distance graph* is a graph in which nodes correspond to objects, and two nodes are connected if and only if the edit distance between the objects represented by those nodes is one. The de Bruijn graph is an edit distance graph. For sequence assembly, we could define the edit distance graph $G_e = \{V_e, E_e\}$ as the graph in which nodes are all length k sub-molecules of reads, also called the *k-spectrum* of the reads, and the edit operation is the shift operation defined above. This graph is a subgraph of the de Bruijn graph induced by a subset of nodes: $\{\langle u, d_u\rangle, \langle v, d_v\rangle\} \in E_e \Leftrightarrow (\{\langle u, d_u\rangle, \langle v, d_v\rangle\} \in E_d) \wedge (u \in V_e) \wedge (v \in V_e)$, and we gave a parallel

method for its construction in [27]. The edit distance graph is nearly suitable for assembly, but some edges correspond to molecules that do not exist in the genome, as we show below.

The *occurrences* of molecule m is the set of tuples $\check{m} = \{\langle x, s\rangle |\ m_{[1,|m|]} = M_{x[s,s+|m|-1]}\}$. A molecule m is a *genomic repeat* if $|\check{m}| > 1$. A genomic repeat m is *right maximal* if there exists two occurrences of m such that the molecules obtained by appending the next base from the genomic sequence for each occurrence in the direction of m's positive strand are different, *i.e.* there exists some i and j such that:

$$m'_i \neq m'_j$$

$$m'_i = \begin{cases} M_{x_i}[s_i, s_i + |m| + 1] & \text{if } m^+[1,|m|] = M^+_{x_i}[s_i, s_i + |m| - 1] \\ M_{x_i}[s_i - 1, s_i + |m|] & \text{otherwise} \end{cases}$$

$$m'_j = \begin{cases} M_{x_j}[s_j, s_j + |m| + 1] & \text{if } m^+[1,|m|] = M^+_{x_j}[s_j, s_j + |m| - 1] \\ M_{x_j}[s_j - 1, s_j + |m|] & \text{otherwise} \end{cases}$$

Similarly, a genomic repeat m is *left maximal* if and only if there exists some i and j such that:

$$m'_i \neq m'_j$$

$$m'_i = \begin{cases} M_{x_i}[s_i - 1, s_i + |m|] & \text{if } m^+[1,|m|] = M^+_{x_i}[s_i, s_i + |m| - 1] \\ M_{x_i}[s_i, s_i + |m| + 1] & \text{otherwise} \end{cases}$$

$$m'_j = \begin{cases} M_{x_j}[s_j - 1, s_j + |m|] & \text{if } m^+[1,|m|] = M^+_{x_j}[s_j, s_j + |m| - 1] \\ M_{x_j}[s_j, s_j + |m| + 1] & \text{otherwise} \end{cases}$$

A *maximal* genomic repeat is a genomic repeat that is both left and right maximal.

Observation 2.1.1. *The edit graph contains at most* $(|\Sigma|^2 - |\Sigma|)w$ *edges that do not correspond to genomic submolecules, where* w *is the number of* $(k-1)$*-molecules that are maximal repeats.*

Proof. Each edge in the graph connects nodes corresponding to two k-molecules $c_1 m$ and $m c_2$, where c_1 and c_2 are extensions of $(k-1)$-molecule m. The edge connecting these nodes

corresponds to a $(k+1)$-molecule, c_1mc_2.

- Case 1: m is a maximal repeat. In the worst case, $|\check{m}| = |\Sigma|$ and each occurrence of m is followed by and preceded by each member of Σ. In this case there are $|\Sigma|^2$ edges in the graph for $|\Sigma|$ occurrences.

- Case 2: m is not maximal. Then m is either not right maximal or not left maximal. In the case the m is not right maximal, then every occurrence of m is succeeded by c_2. By construction, c_1m occurs in the genome. This occurrence must be succeeded by c_2, and therefore c_1mc_2 must occur in the genome. The case that m is not left maximal is handled similarly.

Summing the cases, we find that if w is the number of maximal $(k-1)$-molecules, then the maximum number of invalid edges in the graph is $(|\Sigma|^2 - |\Sigma|)w$. □

Definition 2.1.2. *The observed bidirected de Bruijn Graph is the graph $G_o = \{E_o, V_o\}$ defined by the (k+1)-spectrum of the reads (all length $(k+1)$ submolecules from the reads) in which nodes correspond to the k-spectrum of reads and an edge exists between nodes corresponding to molecules c_1m and mc_2, where m is a $(k-1)$-molecule and and c_1 and c_2 are extensions, if and only if the $(k+1)$-molecule c_1mc_2 is in the $(k+1)$-spectrum.*

Observation 2.1.3. *The observed bidirected de Bruijn Graph is a subgraph of the edit distance graph with $V_o = V_e$ and $E_o \subseteq E_e$.*

Each node u has a set of incident edges $\mathcal{A}_u$ that we partition into two sets, one set $\mathcal{I}_u$ of edges pointing into the node and the other $\mathcal{O}_u$ of edges pointing out of the node. We partition nodes of the observed bidirected graph into four groups based upon underlying repeat structure of the genome, assuming perfect coverage and error correction:

1. A node is a maximal repeat if and only if $|\mathcal{I}_u| > 1 \wedge |\mathcal{O}_u| > 1$
2. A node is a right maximal repeat if and only if $|\mathcal{O}_u| > 1$
3. A node is a left maximal repeat if and only if $|\mathcal{I}_u| > 1$

4. A node is a molecule that is neither a left nor right maximal repeat if if $|\mathcal{I}_u| \leq 1 \wedge |\mathcal{O}_u| \leq 1$

The bidirected de Bruijn graph naturally models both strands of the DNA molecule concurrently, and when finding a path in the bidirected graph that corresponds to the genomic DNA, we are finding both strands concurrently. The bidirected de Bruijn graph constructed using the canonical node and edge labeling as described has the desirable property that if we construct the subgraph corresponding to some molecule α that is shared between two reads r_1 and r_2 independently for r_1 and r_2, both the structure and *labeling* of the graph is deterministic. This property enables constructing such a graph possible in parallel.

2.2 k-String graph

We label each edge with two strings to create a bidirected string graph. Each string corresponds to the first character of each strand of the $(k+1)$-molecule corresponding to the edge. Each string is associated with a traversal direction, again corresponding to the strand directions of the $(k+1)$-molecule (see *Fig. 2.1*).

Let m_u denote the molecule labeling u in the de Bruijn graph and m_v denote the molecule labeling node v. We define strings labeling an edge connecting nodes u and v, c_{uv} and c_{vu} as:

$$c_{uv} = \begin{cases} m_u^+[1] & \text{if } d_u = \text{out} \\ m_u^-[1] & \text{otherwise} \end{cases}$$

$$c_{vu} = \begin{cases} m_v^+[1] & \text{if } d_v = \text{out} \\ m_v^-[1] & \text{otherwise} \end{cases}$$

We convert the bidirected de Bruijn graph to the bidirected k-string graph by compacting all chains in this graph and concatenating edge labels. By construction, given edges $(\langle u, d_u, c_{uw}\rangle, \langle w, d_w, c_{wu}\rangle)$, $(\langle w, d_w, c_{wv}\rangle, \langle v, d_v, c_{vw}\rangle)$ with $in(w) = out(w) = 1$, replace these two edges with a single edge $(\langle u, d_u, c_{uv}\rangle, \langle v, d_v, c_{vu}\rangle)$, with $c_{uv} = c_{uw}c_{wv}$ and $c_{vu} = c_{vw}c_{wu}$ until no such motif exists. The resulting graph is a k-string graph. We show an example genome with the corresponding k-string graph in *Fig. 2.2*.

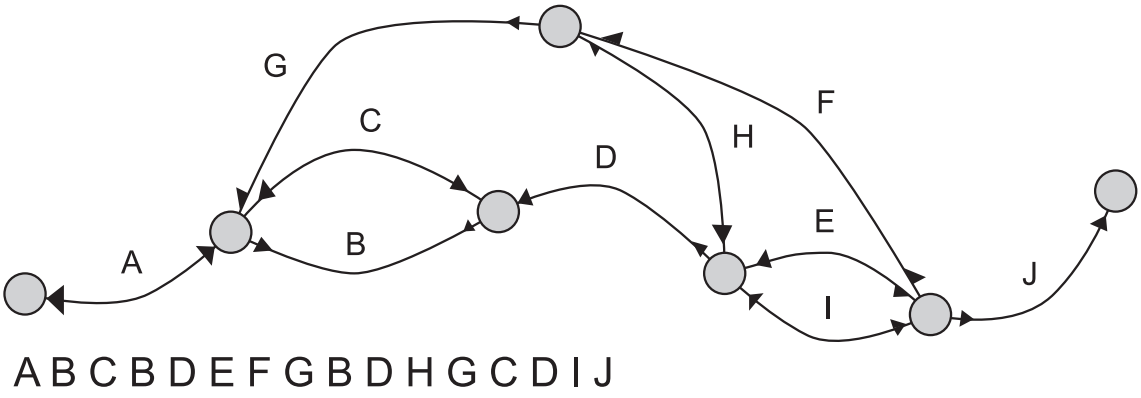

Figure 2.2 An example genome with repeats and the resulting bidirected k-string graph. The genome is given as a sequence of maximal repeat or unique regions, each labeled with a letter from the English alphabet. We draw the graph nodes as gray circles and label the edges using the corresponding letters.

2.3 Relatives of the k-String Graph

The k-string graph has many close relatives in the sequence assembly literature. A directed version of the observed de Bruijn graph is advocated by many researchers [7, 26, 50]. In such a graph, two nodes exist for every node in the bidirected graph, and a traversal through the graph simultaneously moves along two paths at once. Medvedev *et al.* advocated the bidirected de Bruijn graph [41]. The k-string graph is the lowest order graph homeomorphic with this graph, and is edge labeled rather than node labeled.

In the Velvet method of assembly, a directed graph that is structurally similar to the k-string graph is created, but is instead node labeled [71]. In the ALLPATHS assembler, an edge labeled graph called the *unipath graph* is defined. This graph (developed concurrently with this work) is structurally similar to the k-string graph, but unlike the k-string graph, a concatenation of edges along a path does not correspond to the genome. Instead, each subsequent edge in the path overlaps by exactly k characters. Again, the unipath graph is directed rather than bidirected [6].

As described in the introduction, an edge labeled graph closely related to the overlap graph was put forward by Myers. This graph is quite different from the other models presented here,

although Myers advocated both its bidirected nature and the property that the concatenation of edge labels along some traversal of the graph corresponds to the genome. This model was used by Medvedev (who also demonstrated that the formalization of assembly proposed by Myers was NP-hard [41]) in an assembler that first introduced in the literature the idea of a conflict graph [40]. We will discuss the conflict graph in *Chapter 3* when describing our method for transcriptome assembly.

All of these models have strengths and weaknesses, and we developed our specific model, the k-string graph model, after a careful review of the prior literature. The k-string graph has a number of strengths. First, being a bidirected graph, a single traversal of the graph corresponds to the genome. There is no conceptual need to think of two concurrent "linked" traversals as is needed in a directed model. Second, the graph is edge labeled, and a traversal of the graph and the corresponding concatenation of edge labels has significance. This is consistent with other string data structures. For example in a suffix tree the concatenations of edge labels in the tree corresponds to a substring of the string. A weakness of the model is the added complexity of reasoning about a bidirected graph when compared to a directed graph.

2.4 De Bruijn Graph Construction and Representation

We are given a file containing m sequences of total length n, sampled from a genome of total length g. We wish to construct a bidirected string graph with $O(g)$ edges and nodes. First we construct the observed bidirected de Bruijn graph by finding all length $k+1$ molecules present in the input. Then we compact all chains in the graph by converting the problem of chain compaction to undirected list ranking. This conversion is useful because of our requirement that the resulting assembler run on high performance computers.

As we might have $n >> g$, we may not be able read all of the data at once because all reads do not fit into memory, even on large, distributed memory machines. For this reason, we read the data in stages, keeping track of all $(k+1)$-molecules seen in the input.

We encode each $(k+1)$-molecule as a base 4 number (in $2k+2$ bits) using its representative stand. As this process proceeds, we construct a collection of tuples of the form

$\langle (k+1)-\text{molecule}, \text{count} \rangle$ where count is an integer holding the number of times a particular $(k+1)$-molecule has been seen. Formally, for each read r which is a string over the alphabet $\{a, g, c, t\}$, we perform the following transformation:

Algorithm 1 : Extract

$\langle (k+1)\text{-molecule, count} \rangle \quad \overleftarrow{\textbf{Extract}} \quad \mathcal{A} : \langle r \rangle$

$num \leftarrow |r| - k - 1$
for i from 1 to num **do**
  $\overleftarrow{\textbf{Emit:}}$ $\langle \textbf{Encode}(r[i, i+k+1]), 1 \rangle$ {Zero to many tuples can be emitted by any function.}
end for

After reading all $(k+1)$-molecules in one stage, we use a simple reduction to update the $(k+1)$-molecule counts.

Algorithm 2 : SumCount

$\langle (k+1)\text{-molecule, count} \rangle \quad \overleftarrow{\textbf{SumCount}} \quad \mathcal{A} : \langle (k+1)\textbf{-molecule}, \text{count} \rangle$
$\mathcal{B} : \langle (k+1)\textbf{-molecule}, \text{count} \rangle$

$final \leftarrow 0$
for all $\langle (k+1)$**-molecule**, count$\rangle$ in $\mathcal{A}$ **do**
  $final \leftarrow final + count$
end for
for all $\langle (k+1)$**-molecule**, count$\rangle$ in $\mathcal{B}$ **do**
  $final \leftarrow final + count$
end for
$\overleftarrow{\textbf{Emit:}}$ $\langle (k+1)\text{-molecule, final} \rangle$

These functions are combined in a framework that allows for reading sequences in S stages (for now we are ignoring the possibility of sequencing errors, which will be addressed shortly).

Algorithm 3 : Read

$\mathcal{K} \leftarrow \emptyset$
for s from 1 to S **do**
  $\mathcal{R} \leftarrow \textbf{GetReads}(s)$
  $\mathcal{K}'$ $\overleftarrow{\textbf{Extract}}(\mathcal{R})$
  $\mathcal{K}$ $\overleftarrow{\textbf{SumCount}}(\mathcal{K}, \mathcal{K}')$
end for

Once all $(k+1)$-molecules have been extracted from the short reads, the next step in the algorithm is to generate a list of tuples corresponding to the edges in the de Bruijn graph. We

choose to store the graph using two tuples for each edge, each tuple of the form $\langle u, e, c_u, d_u \rangle$, where u is a unique node identifier, e is a unique edge identifier, c_u is a character labeling the edge when traveling out of node u, and $d \in \{$ **out, in** $\}$ indicating the direction of the arrowhead at u. This representation mimics our chosen notation for an edge from the previous section: $(\langle u, d_u, c_{uv} \rangle, \langle v, d_v, c_{vu} \rangle)$, with the field e added in order to explicitly name the edge.

This representation adds flexibility when grouping tuples. If we group tuples by u, each bucket holds the adjacency list for u. If instead we group them by e, then both tuples for that edge come together in a single bucket.

We now describe an algorithm for constructing the initial k-string graph (pre-compaction) from the set of $(k+1)$-molecules using the following transformation:

Algorithm 4 : GenerateGraphTuple

$\langle k-\text{molecule}, e, cov, d_u, c_u \rangle$ $\overleftarrow{\textbf{GenerateGraphTuple}}$ $\mathcal{A} : \langle (k+1)\textbf{-molecule}, \text{e, count} \rangle$

$full \leftarrow (k+1)$-molecule
$left \leftarrow full_{[1,k]}$
$right \leftarrow full_{[2,k+1]}$
$d_l \leftarrow$ **if** $left^+[1] = full^+[1]$ **then** *out* **else** *in*
$d_r \leftarrow$ **if** $right^+[1] = full^+[2]$ **then** *in* **else** *out*
$\overleftarrow{\textbf{Emit:}}$ $\langle left, e, count, d_l, full^+[1] \rangle$
$\overleftarrow{\textbf{Emit:}}$ $\langle right, e, count, d_r, full^-[1] \rangle$

We combine this algorithm with the **Assign** functionality (described in *Chapter 1*) to give a final algorithm for converting the $(k+1)$-molecules into the tuple graph representation.

Algorithm 5 : GenerateGraph

$\mathcal{K} \leftarrow \overleftarrow{\text{Read}()}$
$\mathcal{K} : e$ $\overleftarrow{\textbf{Assign}}$ $\mathcal{K} : (k+1)$-molecule
$\mathcal{D}$ $\overleftarrow{\textbf{GenerateGraphTuples}}(\mathcal{K})$
$\mathcal{D} : u$ $\overleftarrow{\textbf{Assign}}$ $\mathcal{D} : k$-molecule

2.5 List Ranking

The bidirected graph generated in the previous section has many long chains, each corresponding roughly to the contigs or unitigs described in the previous chapter. These chains are

then connected in a more interesting topology that must be further analyzed. We compact these chains by using undirected list ranking.

For the undirected list ranking problem, we are given a set of nodes N connected in a weighted undirected list structure defined by the collection of tuples $\langle u, a_1, w_1, a_2, w_2 \rangle$ of size $|N|$, where $a_1, a_2, u \in N$. w_1 is an integral weight associated with a_1 and w_2 is an integral weight associated with a_2.

The undirected list structure is defined by the following properties:

1. *uniqueness*: For two distinct tuples $\langle u, a_1, w_1, a_2, w_2 \rangle$ and $\langle u', a'_1, w'_1, a'_2, w'_2 \rangle$, $u \neq u'$.
2. *continuity*: $a_i \neq u \rightarrow a_1 \neq a_2$
3. *consistency*: For two distinct tuples $\langle u, a_1, w_1, a_2, w_2 \rangle$ and $\langle u', a'_1, w'_1, a'_2, w'_2 \rangle$, if $a_i = u'$ for some i, then there exists j such that $a'_j = u$ and $w_i = w'_j$.

If $a_1 = u$ or $a_2 = u$, then u is called an endpoint. If $a_1 = a_2 = u$ then u is a singleton list.

The solution to the list ranking problem is a set of tuples $\langle u, e_1, r_1, e_2, r_2 \rangle$. r_1 is the rank of u relative to e_1, the list endpoint in the direction of a_1. r_2 and e_2 are respectively defined in the direction of a_2.

2.5.1 List Ranking Transformation

Conceptually, to construct the k-string graph, we replace each chain in the graph with a single edge, labeled by the concatenation of all edge labels along the chain. We identify adjacent edges in a chain by looking at the adjacency list of nodes in the graph. If an adjacency list has exactly two edges and the endpoints of those edges at the node are *consistent* (pointing in opposite directions), then those two edges are adjacent in a chain. All edges adjacent to a node that does not have the previous property are endpoints in the list ranking formulation. In our framework, we identify the adjacencies of an element using *Algorithm 6*.[1]

[1] As shown in the pseudocode, we choose to index the elements in the bucket from 1 to the size of the bucket, given using the notation $|\mathcal{A}|$.

Algorithm 6 : EdgesToAdjacencies

$\langle id, adj \rangle$ $\overleftarrow{\textbf{EdgesToAdjancies}}$ $\mathcal{A} : \langle \mathbf{u}, e, cov, d_u, c_u \rangle$

if $|\mathcal{A}| = 2$ and $\mathcal{A}[1].d_u \neq \mathcal{A}[2].d_u$ **then**
  $\overleftarrow{\textbf{Emit:}}$ $\langle \mathcal{A}[1].e, \mathcal{A}[2].e \rangle$
  $\overleftarrow{\textbf{Emit:}}$ $\langle \mathcal{A}[2].e, \mathcal{A}[1].e \rangle$
else
  for all $\langle u, e, d_u, c_u \rangle$ in $\mathcal{A}$ **do**
    $\overleftarrow{\textbf{Emit:}}$ $\langle e, e \rangle$
  end for
end if

Observation 2.5.1. *Because the transformation* EdgesToAdjacencies *emits exactly one tuple for each edge tuple in the graph representation, this transformation emits exactly two tuples for each edge e.*

To translate the adjacencies to the list ranking formulation is a simple mapping. We check to see if an adjacency indicates an endpoint, and if it does we set the weight of the edge to 0; otherwise we set it to 1. This logic is given as *Algorithm 7*.

Algorithm 7 : AdjanciesToListRanking

$\langle u, a_1, w_1, e_1, r_1, a_2, w_2, e_2, r_2, m \rangle$ $\overleftarrow{\textbf{AdjanciesToListRanking}}$ $\mathcal{A} : \langle \mathbf{id}, adj \rangle$

$a \leftarrow$ **if** $\mathcal{A}[1].id = \mathcal{A}[1].adj$ **then** 0 **else** 1
$b \leftarrow$ **if** $\mathcal{A}[2].id = \mathcal{A}[2].adj$ **then** 0 **else** 1
$\overleftarrow{\textbf{Emit:}}$ $\langle \mathcal{A}[1].id, \mathcal{A}[1].adj, a, 0, 0, \mathcal{A}[2].adj, b, 0, 0, 0 \rangle$

The field m is an integer marking associated with each node, used in the next section.

2.5.2 Undirected List Ranking

The undirected list ranking problem is a modification of the traditional list ranking problem, which has been extensively studied on parallel computers. The sparse ruling set algorithm achieves the best run time on large data sets with a large number of processors [59], and we have accordingly designed a modified version of the sparse ruling set algorithm to operate on undirected lists and to be compatible with our bucket-emit framework.

The sparse ruling set algorithm is a recursive algorithm on a weighted list. In the base case,

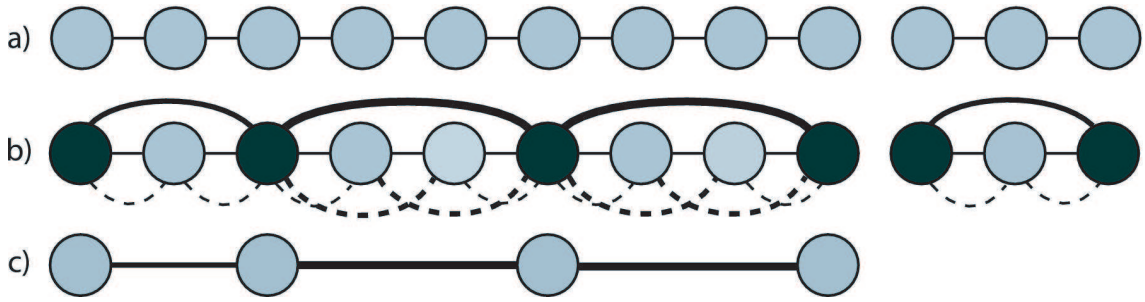

Figure 2.3 The recursive sparse ruling set algorithm. We show in a) the initial problem. We show in b) the recursive formulation, with the thickness of dashed lines corresponding to the distance between unmarked nodes and adjacent marked nodes. We show in c) the recursive problem, with weights given as line thickness. A base case is shown.

a list contains two or fewer elements. For the inductive case, we wish to achieve the following objectives. First, we wish to mark some subset of nodes which include all endpoints and some other nodes, each marked with probability ρ. Second, we wish to find the distance between each unmarked node and its two closest marked nodes. Finally, we wish to find the distance between adjacent marked nodes. We show the recursive formulation used in the algorithm in *Fig. 2.3*.

Once we have this information, we create a new instance of the problem containing only the marked nodes, with the adjacencies of each marked node set to be the nearest marked nodes in the list, and the weights as the distance to those marked nodes. This new instance is then recursively solved (see *Fig. 2.3*). After the recursion, we know $\mathcal{R}$ for all marked nodes. We compute $\mathcal{R}$ for all unmarked nodes by combining this solution with the distances from the unmarked nodes to the marked nodes.

We now formally describe the algorithm in our computational framework. The algorithm can be viewed as passing a message containing four components: $\mathcal{M} = \langle t, s, o, d, e \rangle$, where t is the target of the message, s is the source of the message, o is the id of the originating marked node, d is the distance to the originating marked node, and e is a boolean field indicating if the originating marked node was an endpoint. These messages originate at marked nodes and

are passed along the length of the list until reaching a second marked node.

We use the field m to indicate the marking of a list ranking tuple. Initially, $m = 0$ for all nodes. We set m to 1 for marked nodes. We set m to -1 to identify the base case. The first algorithm we present is used to mark nodes and generate messages. It generates two messages for each marked node, one for each adjacency, and is given in *Algorithm 8*.

Algorithm 8 : GenerateListMessages

$\langle t, s, o, d, e \rangle$ $\overleftarrow{\textbf{GenerateListMessages}}$ $\mathcal{A} : \langle u, a_1, w_1, m_1, r_1, a_2, w_2, m_2, r_2, m \rangle$

$p \leftarrow Rand()$
if $a_1 = u$ **then**
  if $a_2 = u$ **then**
    $m \leftarrow -1$
  else
    $m \leftarrow 1$
    $\overleftarrow{\textbf{Emit:}}$ $\langle a_2, u, u, r_2,$ **true** $\rangle$
  end if
else if $a_2 = u$ **then**
  $m \leftarrow 1$
  $\overleftarrow{\textbf{Emit:}}$ $\langle a_1, u, u, r_1,$ **true** $\rangle$
else if $p < \rho$ **then**
  $m \leftarrow 1$
  $\overleftarrow{\textbf{Emit:}}$ $\langle a_2, u, u, r_2,$ **false** $\rangle$
  $\overleftarrow{\textbf{Emit:}}$ $\langle a_1, u, u, r_1,$ **false** $\rangle$
end if

The second algorithm propagates messages. The first step is to analyze from which direction the message came. This is done by comparing a_1 to s. We call the tuple information in the incoming direction a_i, w_i, e_i, r_i and the adjacency information in the outgoing direction a_n, w_n, e_n, r_n. We record the originating marked node as the temporary endpoint: $e_i \leftarrow o$ and the distance to that node as the temporary rank: $r_i \leftarrow d$. If the node is marked, we do not propagate the message. If the node is an endpoint and the originating marked node was also an endpoint, the algorithm sets m to -1 in order to indicate that the newly recorded e_i and r_i are final. If the node is unmarked, the message is propagated to a_n adding the weight of the next edge w_n to the distance traveled d. This process is given in *Algorithm 9*.

After all messages have been propagated, the recursive algorithm is constructed by copying,

Algorithm 9 : PropogateListMessages

$\langle t, s, o, d, e\rangle$ $\overleftarrow{\textbf{PropogateListMessages}}$ $\mathcal{A} : \langle \mathbf{u}, a_1, w_1, e_1, r_1, a_2, w_2, e_2, r_2, m\rangle$
$\mathcal{B} : \langle \mathbf{t}, s, o, d, e\rangle$

for all $\langle t, s, o, d, e\rangle$ in $\mathcal{B}$ **do**
  $i \leftarrow$ **if** $a_1 = s$ **then** 1 **else** 2
  $n \leftarrow$ **if** $a_1 = s$ **then** 2 **else** 1
  $e_i \leftarrow o$
  $r_i \leftarrow d$
  if ($a_1 = u$ or $a_2 = u$) and e **then**
    $m \leftarrow -1$
  else if $m = 0$ **then**
    $\overleftarrow{\textbf{Emit:}}$ $\langle a_n, u, o, d + w_n, e$ **true** $\rangle$
  end if
end for

for all marked tuples, the temporary endpoint information into the adjacency information: $a_i \leftarrow e_i$ and $w_i \leftarrow r_i$. We emit the new list ranking problem. This step is given in *Algorithm 10.*

Algorithm 10 : GetRecursiveProblem

$\langle \mathbf{u}, a_1, w_1, e_1, r_1, a_2, w_2, e_2, r_2, m\rangle$ $\overleftarrow{\textbf{GetRecursiveProblem}}$ $\mathcal{A} : \langle \mathbf{u}, a_1, w_1, e_1, r_1, a_2, w_2, e_2, r_2, m\rangle$

if $m = 1$ **then**
  $\overleftarrow{\textbf{Emit:}}$ $\langle u, e_1, r_1, e_1, r_1, e_2, r_2, e_2, r_2, 0\rangle$
end if

Finally, after the recursive problem has been solved, each unmarked node must query its two neighboring marked nodes for their ranks and endpoints. This information will be used to calculate the final ranks and endpoints for the unmarked nodes. This task is complicated by adjacency numbering having no inherent meaning. For instance, e_1 returned by one neighboring marked node may be different from e_1 returned by other neighboring marked node. For this reason, we must compare the endpoints returned by the marked nodes to discover which endpoints are equivalent. Finally, we calculate our position in the list relative to each endpoint by making use of the property that all edges in the undirected list ranking formulation have non-negative weight. Therefore, we infer which of the two marked nodes is between the unmarked node and each endpoint of the list using the returned distance information.

We decompose the above method into four steps. The first step reintegrates the solution to the recursive problem with the current problem, given in *Algorithm 11*.

Algorithm 11 : Integrate

$\overleftarrow{\textbf{Integrate}}$ $\mathcal{A} : \langle \mathbf{u}, a_1, w_1, e_1, r_1, a_2, w_2, e_2, r_2, m \rangle$
$\mathcal{B} : \langle \mathbf{u}, a_1, w_1, e_1, r_1, a_2, w_2, e_2, r_2, m \rangle$

if $|\mathcal{B}| > 0$ **then**
  $\mathcal{A}[1].e_1 \leftarrow \mathcal{B}[1].e_1$
  $\mathcal{A}[1].r_1 \leftarrow \mathcal{B}[1].r_1$
  $\mathcal{A}[1].e_2 \leftarrow \mathcal{B}[1].e_2$
  $\mathcal{A}[1].r_2 \leftarrow \mathcal{B}[1].r_2$
end if

The second step sends query messages from unmarked nodes to marked nodes, given in *Algorithm 12*.

Algorithm 12 : QueryMarkedNodes

$\langle t, s \rangle$ $\overleftarrow{\textbf{QueryMarkedNodes}}$ $\langle \mathbf{u}, a_1, w_1, e_1, r_1, a_2, w_2, e_2, r_2, m \rangle$

if $m = 0$ **then**
  $\overleftarrow{\textbf{Emit:}}$ $\langle e_1, u \rangle$
  $\overleftarrow{\textbf{Emit:}}$ $\langle e_2, u \rangle$
end if

The third step returns ranking information from marked nodes to unmarked nodes, given in *Algorithm 13*.

Algorithm 13 : ReturnListInformation

$\langle t, s, e_1, r_1, e_2, r_2 \rangle$ $\overleftarrow{\textbf{ReturnListInformation}}$ $\mathcal{A} : \langle \mathbf{t}, s \rangle$
$\mathcal{B} : \langle \mathbf{u}, a_1, w_1, e_1, r_1, a_2, w_2, e_2, r_2, m \rangle$

for all $\langle t, s \rangle$ in $\mathcal{A}$ **do**
  $\overleftarrow{\textbf{Emit:}}$ $\langle s, t, e_1, r_1, e_2, r_2 \rangle$
end for

The fourth step takes the information returned from marked nodes, and updates the rank information for unmarked nodes, given in *Algorithm 14*.

Given these last four algorithms, we have described all of the components needed to present a recursive list ranking algorithm using our computational framework. It is given in *Algorithm*

Algorithm 14 : ComputeRank

$\overleftarrow{\textbf{ComputeRank}}$ $\quad \mathcal{A} : \langle \mathbf{t}, s, e_1, r_1, e_2, r_2 \rangle$
$\quad \mathcal{B} : \langle \mathbf{u}, a_1, w_1, e_1, r_1, a_2, w_2, e_2, r_2, m \rangle$

$one \leftarrow$ **if** $\mathcal{A}[1].s = e_1$ **then** $\mathcal{A}[1]$ **else** $\mathcal{A}[2]$
$two \leftarrow$ **if** $\mathcal{A}[1].s = e_1$ **then** $\mathcal{A}[2]$ **else** $\mathcal{A}[1]$
if $one.e_1 = two.e_1$ **then**
  $a \leftarrow$ **if** $one.r_1 < two.r_1$ **then** 1 **else** 2
  $b \leftarrow$ **if** $one.r_1 < two.r_1$ **then** 2 **else** 1
else
  $a \leftarrow b \leftarrow$ **if** $one.r_1 < two.r_2$ **then** 1 **else** 2
end if
$e_1 \leftarrow one.e_a$
$r_1 \leftarrow r_1 + one.r_a$
$e_2 \leftarrow two.e_b$
$r_2 \leftarrow r_2 + two.r_b$

15.

Algorithm 15 : ListRank($\mathcal{L}$)

if $|\mathcal{L}| > 0$ **then**
  $\mathcal{M}$ $\overleftarrow{\textbf{GenerateListMessages}}(\mathcal{M})$
  while $|\mathcal{M}| > 0$ **do**
    $\mathcal{M}$ $\overleftarrow{\textbf{PropogateListMessages}}(\mathcal{M}, \mathcal{L})$
  end while
  $\mathcal{L}'$ $\overleftarrow{\textbf{GetRecursiveProblem}}(\mathcal{L})$
  ListRank($\mathcal{L}'$)
  $\overleftarrow{\textbf{Integrate}}(\mathcal{L}, \mathcal{L}')$
  $\mathcal{F}$ $\overleftarrow{\textbf{QueryMarkedNodes}}(\mathcal{L})$
  $\mathcal{R}$ $\overleftarrow{\textbf{ReturnListInformation}}(\mathcal{F}, \mathcal{L})$
  $\overleftarrow{\textbf{ComputeRank}}(\mathcal{R}, \mathcal{L})$
end if

2.5.2.1 Run-Time Analysis

We wish to briefly address the characteristics of *Algorithm 15*. The number of rounds of message passing in the loop in *Algorithm 15* is equivalent to the longest distance between two marked nodes. As each node is randomly marked, this distance is bounded by $3\rho \ln(|\mathcal{L}|)$ with high probability [9]. Therefore, the expected number of iterations in the loop is $O(\log(|\mathcal{L}|))$.

Marked nodes emit two messages during the initial message construction and never again. Unmarked nodes receive and reemit exactly two messages, which originated at the adjacent marked nodes, during some processing step in this loop. Therefore, the summation of the distributed size of the array $\mathcal{M}$ over all of these $O(\log(|\mathcal{L}|))$ iterations is $O(|\mathcal{L}|)$.

Because the size of the list ranking problem is expected to exponentially decrease in $O(\log|\mathcal{L}|)$ recursive calls, the total expected distributed size of array $\mathcal{L}$ over all recursive steps is $O(|\mathcal{L}|)$. Moreover, the total number of iterations in the inner loop of *Algorithm 15* is $O(\log^2|\mathcal{L}|)$ ($\log|\mathcal{L}|$ per recursive problem), and the total distributed problem size is $|\mathcal{L}|$, including all recursive calls of *Algorithm 15* and all iterations of the inner loop over those recursive calls. Therefore, the total amount of work done using our computational framework is proportional to the amount of work needed to processes the original list $\mathcal{L}$.

2.6 k-String Graph Construction

After solving the list ranking problem, we use the solution to construct the k-string graph. We represent the k-string graph using two tuple arrays. The first array stores the edge labels as chains of tuples. We denote this array $\mathcal{C}$ with tuples of the form $\langle ch, e, p_F, p_R, c_F, c_R\rangle$. The components of the tuple have the following meaning:

- ch is an integer identifier for the chain.
- e is an integer identifier of the ($(k+1)$-molecule) present at this position in the graph.
- p_F and p_R hold the position of this element from each end of the chain, p_F the rank in the forward direction, p_R the rank in the reverse direction.
- c_F and c_R hold the characters to be output when reading the chain in the forward and reverse directions, respectively.

The second tuple array holds the topology of the graph. We label this array $\mathcal{T}$, with tuples of the form: $\langle ch, dir, l, u, d_u, i_u, o_u, cov_u, v, d_v, i_v, o_v, cov_v\rangle$. The components are described below:

- ch is an integer identifier for the chain

- u and v are integer identifiers for the endpoints of this chain.
- dir indicates if the chain is read forward or backwards when traversing the edge from u to v.
- l indicates the length of the chain.
- d_u and d_v are either *in* or *out* and indicate the direction of the edge at the corresponding endpoints.
- i_u and i_v indicate the number of edges incident to u and v respectively with in directions.
- o_u and o_v indicate the number of edges incident to u and v respectively with out adjacencies.
- cov_u and cov_v indicate the average coverage of the edge near the endpoints u and v respectively. What we mean by average coverage near the endpoints is defined by a parameter to the assembler, *coverage window*. The coverage window indicates how many consecutive $(k+1)$-molecules, starting at the endpoint, should be used to calculate the average coverage. If the length of the chain is smaller than *coverage window*, then the entire chain is used to calculate the average.

We store two tuples for each edge in the string graph, with u and v interchanged. This duplication of information allows us to find motifs more easily in the next section. In order to create the two arrays of tuples described above, we must first integrate the list ranking solution and the de Bruijn graph tuples using the following function, given in *Algorithm 16.*

Algorithm 16 : SetRank

$\langle u, e, cov, d_u, c_u, ch, r_1, r_2 \rangle$ $\overleftarrow{\textbf{SetRank}}$ $\mathcal{A} : \langle u, \mathbf{e}, cov, d_u, c_u \rangle$
$\mathcal{B} : \langle \mathbf{u}, a_1, w_1, e_1, r_1, a_2, w_2, e_2, r_2, m \rangle$

for all $\langle u, e, cov, d_u, c_u \rangle$ in $\mathcal{A}$ **do**
  $a \leftarrow$ **if** $(e_1 < e_2)$ **then** 1 **else** 2
  $b \leftarrow$ **if** $(e_1 < e_2)$ **then** 2 **else** 1
  $\overleftarrow{\textbf{Emit:}}$ $\langle u, e, cov, d_u, c_u, e_a, r_a, r_b \rangle$
end for

Next we, give the algorithms for constructing the k-string graph, one for $\mathcal{C}$ and three for $\mathcal{T}$. We start with the generation of $\mathcal{T}$. To do so, we arbitrarily assign the direction of the chain from edge e_1 to edge e_2 as the forward direction of the chain. This assignment is necessary to have a consistent interpretation of edge traversal when reasoning independently about the topology tuple list and the chain tuple list. In the first function, we identify the nodes are the endpoints of the chain by finding the node shared between edge with rank $r_1 = 0$ and edge with rank $r_1 = 1$. To make this assignment easier, we first sort the bucket of chain tuples by r_1. We calculate the average coverage of $(k+1)$-molecules near the ends of the chain as previously described with the introduction of the topology tuple. In the second and third functions, we calculate the values for fields i_u, o_u, i_v, and o_v. These algorithms are give in *Algorithms 17, 18,* and *19*.

Algorithm 17 : ExtractToplogy

$\langle ch, dir, l, u, d_u, i_u, o_u, cov_u, v, d_v, i_v, o_v, cov_v \rangle$ $\overleftarrow{\textbf{ExtractTopology}}$ $\mathcal{A} : \langle u, e, cov, d_u, c_u, \textbf{ch}, r_1, r_2 \rangle$

if $|\mathcal{A}| = 2$ **then**
  $\overleftarrow{\textbf{Emit:}}$ $\langle \mathcal{A}[1].e_1, F, |\mathcal{A}|, \mathcal{A}[1].u, \mathcal{A}[1].d_u, 0, 0, \mathcal{A}[1].cov, \mathcal{A}[2].u, \mathcal{A}[2].d_u, 0, 0, \mathcal{A}[1].cov \rangle$
  $\overleftarrow{\textbf{Emit:}}$ $\langle \mathcal{A}[1].e_1, R, |\mathcal{A}|, \mathcal{A}[2].u, \mathcal{A}[2].d_u, 0, 0, \mathcal{A}[1].cov, \mathcal{A}[1].u, \mathcal{A}[1].d_u, 0, 0, \mathcal{A}[1].cov \rangle$
else
  SortByR1$(\mathcal{A})$
  $start \leftarrow$ **if** $\mathcal{A}[1].u = \mathcal{A}[3].u$ or $\mathcal{A}[1].u = \mathcal{A}[4].u$ **then** $\mathcal{A}[2]$ **else** $\mathcal{A}[1]$
  $end \leftarrow$ **if** $\mathcal{A}[|A|].u = \mathcal{A}[|A|-2].u$ or $\mathcal{A}[|A|].u = \mathcal{A}[|A|-3].u$ **then** $\mathcal{A}[|A|-1]$ **else** $\mathcal{A}[|A|]$
  $num \leftarrow$ **if** $|\mathcal{A}| \div 2 < window$ **then** $|\mathcal{A}|$ **else** $window \times 2$
  $cov_a \leftarrow (\sum_{i=1}^{num} \mathcal{A}[i].cov) \div num$
  $cov_b \leftarrow \left(\sum_{i=|\mathcal{A}|-num}^{|\mathcal{A}|} \mathcal{A}[i].cov\right) \div num$
  $\overleftarrow{\textbf{Emit:}}$ $\langle start.e, F, |\mathcal{A}| \div 2, start.u, start.d_u, 0, 0, cov_a, end.u, end.d_u, 0, 0, cov_b \rangle$
  $\overleftarrow{\textbf{Emit:}}$ $\langle start.e, R, |\mathcal{A}| \div 2, end.u, end.d_u, 0, 0, cov_b, start.u, start.d_u, 0, 0, cov_a \rangle$
end if

We give an algorithm for constructing chain representation. Importantly, to complete this action, we must correctly assign the characters to be written when traversing the chain in the forward and reverse directions. This requires remembering which of the two endpoints of an edge was last visited when moving from edge to edge in the two-tuple-per edge representation. It also requires, as was done in the function to construct the topology representation, that

Algorithm 18 : AssignAdjacencyInfo

$\overleftarrow{\textbf{AssignAdjacencyInfo}}$ $\mathcal{A} : \langle ch, dir, l, \mathbf{u}, d_u, i_u, o_u, cov_u, v, d_v, i_v, o_v, cov_v \rangle$

$I \leftarrow 0$
$O \leftarrow 0$
for all $\langle ch, dir, l, \mathbf{u}, d_u, i_u, o_u, cov_u, v, d_v, i_v, o_v, cov_v \rangle$ **do**
  $I \leftarrow$ **if** $d_u = in$ **then** $I + 1$ **else** I
  $O \leftarrow$ **if** $d_u = out$ **then** $O + 1$ **else** O
end for
for all $\langle ch, dir, l, \mathbf{u}, d_u, i_u, o_u, cov_u, v, d_v, i_v, o_v, cov_v \rangle$ **do**
  $i_u \leftarrow I$
  $o_u \leftarrow O$
end for

Algorithm 19 : ExcangeAdjacencyInfo

$\overleftarrow{\textbf{ExcangeAdjacencyInfo}}$ $\mathcal{A} : \langle \mathbf{ch}, dir, l, u, d_u, i_u, o_u, cov_u, v, d_v, i_v, o_v, cov_v \rangle$

$\mathcal{A}[1].i_v \leftarrow \mathcal{A}[2].i_u$
$\mathcal{A}[1].o_v \leftarrow \mathcal{A}[2].o_u$
$\mathcal{A}[2].i_v \leftarrow \mathcal{A}[1].i_u$
$\mathcal{A}[2].o_v \leftarrow \mathcal{A}[1].o_u$

we identify which node serves as the endpoint for the chain. Finally, although this function requires that the bucket be in order according to r_1, we assume that this order has already been achieved by calling *Algorithm 17*. The chain construction method is given in *Algorithm 20*.

Finally, we have done the necessary groundwork to give a complete algorithm that converts the de Bruijn graph representation, which we denoted as collection $\mathcal{D}$, constructed from the short sequence reads in *Algorithm 5*, to the k-string graph representation, denoted as collections $\mathcal{T}$ and $\mathcal{C}$. It is given in *Algorithm 21*.

2.7 Sequencing Errors

Up until this point, we have purposefully ignored the complications introduced by sequencing error in order to simplify the discussion of graph creation. Handling sequencing error, however, is one of the most important tasks a fully realized assembler must achieve.

We assume when handling error that the data is reasonably error free; a 1% error rate is

Algorithm 20 : ExtractChains

$\langle ch, e, p_F, p_R, c_F, c_R \rangle \quad \overleftarrow{\textbf{ExtractChains}} \quad \mathcal{A} : \langle u, e, d_u, c_u, \textbf{ch}, r_1, r_2 \rangle$

if $|\mathcal{A}| = 2$ **then**
  $\overleftarrow{\textbf{Emit:}}$ $\langle \mathcal{A}[1].ch, \mathcal{A}[1].e, \mathcal{A}[1].r_1, \mathcal{A}[1].r_2, \mathcal{A}[1].c_u, \mathcal{A}[2].c_u \rangle$
else
  $last \leftarrow$ **if** $\mathcal{A}[1].u = \mathcal{A}[3].u$ or $\mathcal{A}[1].u = \mathcal{A}[4].u$ **then** $\mathcal{A}[2]$ **else** $\mathcal{A}[1]$
  for $i = 1$ to $|\mathcal{A}|$ step 2 **do**
    $last \leftarrow$ **if** $\mathcal{A}[i].u = last.u$ **then** $\mathcal{A}[i]$ **else** $\mathcal{A}[i+1]$
    $next \leftarrow$ **if** $\mathcal{A}[i].u = last.u$ **then** $\mathcal{A}[i+1]$ **else** $\mathcal{A}[i]$
    $\overleftarrow{\textbf{Emit:}}$ $\langle last.ch, last.e, last.r_1, last.r_2, next.c_u, last.c_u \rangle$
    $last \leftarrow next$
  end for
end if

Algorithm 21 : GenerateStringGraph

$\mathcal{A}$ $\overleftarrow{\textbf{EdgesToAdjacencies}}$ $\mathcal{D}$
$\mathcal{L}$ $\overleftarrow{\textbf{AdjacenciesToListRanking}}$ $\mathcal{A}$
ListRank$(\mathcal{L})$
$\mathcal{R}$ $\overleftarrow{\textbf{SetRank}}$ $\mathcal{L}$
$\mathcal{T}$ $\overleftarrow{\textbf{ExtractTopology}}$ $\mathcal{R}$
$\overleftarrow{\textbf{AssignAdjacencyInfo}}$ $\mathcal{T}$
$\overleftarrow{\textbf{ExchangeAdjacencyInfo}}$ $\mathcal{T}$
$\mathcal{C}$ $\overleftarrow{\textbf{ExtractChains}}$ $\mathcal{R}$

assumed. Unlike overlap-layout-consensus methods, in which error handling is delegated to the final consensus phase, any method that models assembly as finding a tour in the graph requires that sequencing error be corrected before the even more complicated task of reconstructing the repeat organization is tackled. This is because errors manifest themselves as false edges in the graph, producing tangles. High coverage, randomness in error location, and a low error rate are helpful when dealing with errors.

When assembling larger reads, Pevzner *et al.* proposed correcting errors in reads using what they termed the spectral alignment[50] problem, for which one tries to edit each k-mer in the data such that it is part of the spectrum of k-mers that occur in the genome. Batzglou *et al.* [2] and Sundquist *et al.* [62] describe a method for solving this problem that relies on multiple sequence alignment which is computationally expensive, while Chaisson *et al.* [7] introduced a faster dynamic programming solution. Butler *et al.* solve the spectral alignment problem over three spectra as a preprocessing phase for the ALLPATHs assembler.

While we think that solving the spectral alignment problem can be useful,[2] we have not as a part of this work explored solving this problem for large genomes. On the other hand, in solving the error removal problem, we have given a method for discovering the $(k+1)$-spectrum of the genome to be assembled, which is an important first step in any read editing method.

We will describe two ways of looking at error in short read data. The first is a simple thresholding vision, and is presented primarily for theoretical interest. In practice, coverage is not high enough or uniform enough to use such a method, and it ignores contextual information that the second method presented, a graph editing method, takes into account. It does, however, provide useful context for thinking about the frequencies of k-mers in the data.

2.7.1 Thresholding

We wish to find a threshold such that, with high probability, all k-molecules with frequency below this threshold are artifacts due to sequencing errors. Correspondingly, all real

[2]Its usefulness is diminished when assembling very short reads, because k is very close to the read length l and coverage is high, making any additional information gained after editing reads to likely be redundant. This information is useful in resolving repeats of length greater than k.

k-molecules should occur at a rate above this threshold. We make the simplifying assumption that the sampling of the genome and the sequencing error are independent, uniform, random processes. For the initial analysis, we ignore the increased frequencies of certain k-molecules due to the presence of repeats, but address this subsequently.

Let g be the length of the genome, l be the length of each read, r be the substitution error rate per base, and c be the coverage rate per base. We describe a k-molecule as a tuple $s = \langle p_1, p_2, ..., p_k \rangle$. An *edit profile* is the corresponding tuple $\langle c_1, c_2, ..., c_k \rangle$ with $c_i \in [0, 3]$. The probability $P(c_i = 0)$ is $1 - r$ (i.e., base called correctly), while the probabilities $P(c_i = 1) = P(c_i = 2) = P(c_i = 3) = \frac{r}{3}$ (corresponding to each of the three incorrect base call possibilities).

We are interested in two classes of edit profiles: the identity profile, which has probability $(1 - r)^k$; and the profiles corresponding to a single edit, each with probability $\frac{r}{3}(1 - r)^{(k-1)}$. We ignore other profiles given their low probabilities for practical values of r. The expected rate at which the identity profile occurs at some location in the genome is:

$$\lambda_p = \left(\frac{c(l - k)}{l} \right) (1 - r)^k$$

We say call $\frac{c(l-k)}{l}$ the *effective coverage* of the genome. The expected rate at which a single error edit profile occurs is:

$$\lambda_d = \left(\frac{c(l - k)}{l} \right) \left(\frac{r}{3} \right) (1 - r)^{k-1}$$

The number of times a particular k-molecule (the identity profile at a particular position) is seen in the data is a Poisson process, with the expected number of k-molecules seen exactly t times given by the equation:

$$\mathcal{C}_t = \left(\frac{{\lambda_p}^t e^{-\lambda_p}}{t!} \right) g$$

The expected number of times a single base k-mer edit is seen exactly t times is defined similarly:

$$\mathcal{E}_t = \left(\frac{{\lambda_d}^t e^{-\lambda_d}}{t!} \right) 3kg$$

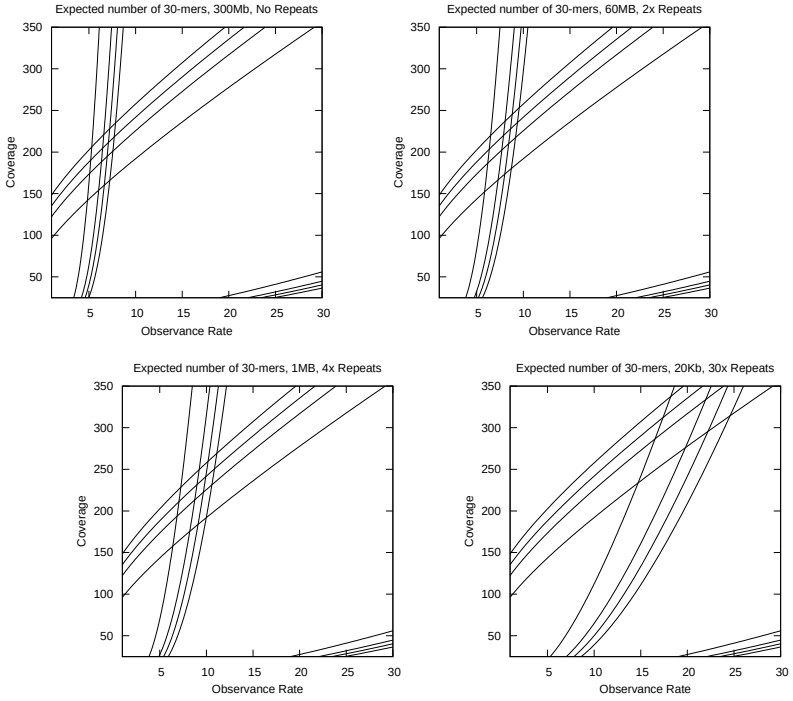

Figure 2.4 Contour lines for $\mathcal{C}_t$ and $\mathcal{E}_t$ when plotted against c and t. We plot $\log_{10}\mathcal{C}_t = \{-2, -1, 0, \text{ and } 2\}$ for genome length 300Mb, read length of 40bp, 1% error, and k=30. We also plot $\log_{10}\mathcal{E}_t = \{-2, -1, 0, \text{ and } 2\}$ for a hypothetical genome repeat decomposition, superimposed against $\mathcal{C}_t$. We show in the upper left a plot of $\mathcal{C}_t$ for 300Mb of unique sequence. We show in the upper right a plot for 60Mb of sequence repeated twice. We show in the lower left a plot for 1Mb of sequence repeated 4 times. Finally, we show in the lower right a plot for 20Kb repeated 30 times. These plots indicate that with 1% sequencing error rate, 30-mers can be differentiated using a simple threshold method at 250-fold to 300-fold coverage.

Given a genome of length g and an error rate r, we wish to find coverage c such that some threshold τ separates good k-molecules from bad k-molecules with high probability. We create a 3-dimensional plot of $\mathcal{C}_t$ and $\mathcal{E}_t$ given c and τ, as shown in the upper left quadrant of *Fig. 2.4*. Observe that if the genome is unique, 200-fold coverage of length 40bp reads with 1% error gives good separation of 30-mers.

We can update this analysis to consider sampling bias and repeats by modeling both as non-uniform coverage of some genome with only unique k-molecules. We analyze the k-molecules by separating this genome into sets of k-molecules with similar coverage. Our task is to find a single threshold that separates real k-mers from errors in all sets simultaneously. As shown in *Fig. 2.4*, for a genome of length 300Mb, a 1% error rate, and an average read length of 40, we expect that 300-fold coverage will separate good and bad 30-mers for many repeats.

2.7.2 Graph Editing

As mentioned previously, the thresholding approach is far too simplistic to work well on experimental short read data, although we have demonstrated that the theoretical analysis described above is accurate for synthetic data with uniform random sampling of the genome [30]. Still, in this work we are interested in a functional assembler, and therefore we present a second method of error discovery, a graph editing method.

Graph editing was proposed by Pevzner *et al.* [49] to achieve the following goals: removal if any errors left over after spectral alignment, editing the graph such that nearly identical copies of repeats become a single edge, and dealing with tandem repeats. The achieved this by removing what they termed *bulges* and *whirls* from the graph. Bulges were defined as short undirected cycles in the graph and whirls as short directed cycles. They considered a cycle short if its girth was less than some parameter g. They formally defined a combinatorial problem of finding a maximum subgraph with large girth, where all short cycles have been removed. In addition, they described the process of *erosion* corresponding to tip removal below, and the removal of low coverage edges, which we call *spurious links*.

In the Velvet method for short sequence assembly [71], the authors construct the de Bruijn

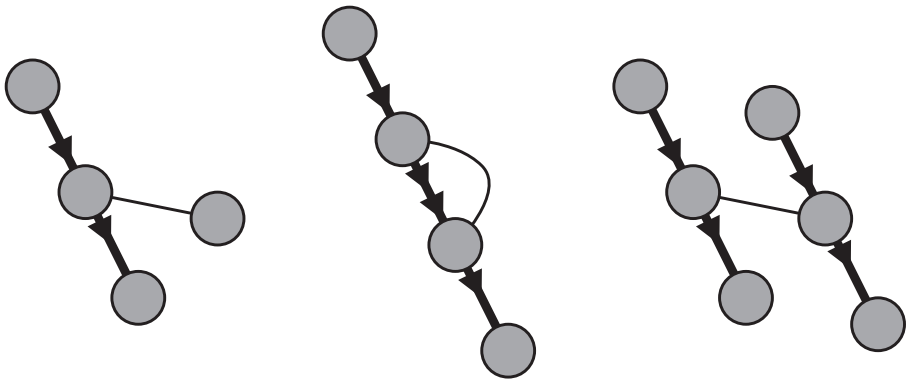

Figure 2.5 Motifs used to identify errors with coverage indicated by line weight. From left to right: a tip, a bubble, and a spurious link.

graph from the uncorrected sequence data and then correct errors using a similar process of finding and editing motifs in this graph. The complexity of the graph for large organisms and high coverage makes this problematic as a stand alone approach, especially in the context of the difficulty of finding parallel graph algorithms. At the same time, the thresholding approach described above relies on very high coverage, while the graph editing approach combines coverage information with contextual information in the k-string graph. This allows lowering the threshold presented in the previous section if the coverage is not adequate to produce a separation of real k-molecules and artifacts. It also allows a more robust method if the sequence sampling of the genome is not uniform and at random.

We give a more simplistic approach to graph editing than given by the methods above, but slightly more complicated than the method presented in the ABySS short sequence assembly method. The three classes of error motifs: cycles, tips, and spurious links, remain in all four error identification methods mentioned. The simplification we propose is necessary due to the difficulty of path finding on a distributed graph or, alternatively, using our computational framework.

When editing the graph, we wish to update the graph representation (collections $\mathcal{T}$ and $\mathcal{C}$). We use graph editing algorithms presented in *Chapter 3* to this end. Their functionality should be clear from their names, but the interested reader should return to this section after reading *Chapter 3* for a complete understanding of the following methods.

A tip (as shown in *Fig. 2.5*) is identified by a node u in the graph that has a single adjacent edge e with low coverage, and in which the far endpoint of the single edge (v) has an alternate path. Given the topology tuple, this means that if $d_v = in$ then $i_v > 1$ and $o_v \geq 1$, or if $d_v = out$ then $o_v > 1$ and $i_v \geq 1$. An algorithm for removing a tip from an adjacency list $\mathcal{A}_u$ is given as Algorithm 22. It uses the *Delete* algorithm defined in *Chapter 3*.

Algorithm 22 : RemoveTips($\mathcal{A}_u$)

if $|\mathcal{A}| = 1$ and $\mathcal{A}[1].l < T$ and $\mathcal{A}[1].cov_v < C$ and ($\mathcal{A}[1].d_v = in$ and $\mathcal{A}[1].i_v > 1$) or ($\mathcal{A}[1].d_v = out$ and $\mathcal{A}[1].o_v > 1$) **then**
  Delete($\mathcal{A}[1]$)
end if

A singleton is identified by a node u in the graph that has a single adjacent edge e with low coverage, and in which the far endpoint of the single edge (v) is only adjacent to e. Singleton edges are likely formed by read errors that result in a read forming its own connected component in the graph. We give an algorithm for removing a singleton adjacency list $\mathcal{A}_u$ in *Algorithm 23*.

Algorithm 23 : RemoveSingletons($\mathcal{A}_u$)

if $|\mathcal{A}| = 1$ and $\mathcal{A}[1].l < T$ and $\mathcal{A}[1].cov_v < C$ and $\mathcal{A}[1].i_v + \mathcal{A}[1].o_v = 1$ **then**
  Delete($\mathcal{A}[1]$)
end if

As shown in *Fig. 2.5*, a bubble forms when a redundant edge connects two nodes. We define a redundant edge as an edge that is the same length as some higher coverage edge (the difference between the lengths must be less than some parameter W) and has coverage below some threshold T. We give an algorithm for removing a redundant edge in an adjacency list $\mathcal{A}_u$ in *Algorithm 24*.

As shown in *Fig. 2.5*, a spurious link forms when an unneeded edge connects two nodes. We define an unneeded edge as an edge that is shorter than some threshold L, has coverage below some threshold T, and connects two nodes lying on some alternate paths. We give an algorithm for removing an unneeded edge in an adjacency list $\mathcal{A}_u$ in *Algorithm 25*.

We give an algorithm that emits the graph manipulation tuples (see *Chapter 3*) for the

Algorithm 24 : RemoveBubbles($\mathcal{A}_u$)

SortByV($\mathcal{A}$)
$i \leftarrow 1$
repeat
  $j \leftarrow i$
  $max \leftarrow \mathcal{A}[j].cov_u$
  $max_j \leftarrow j$
  repeat
    if $max < \mathcal{A}[j].cov_u$ **then**
      $max \leftarrow \mathcal{A}[j].cov_u$
      $max_j \leftarrow j$
    end if
    $j \leftarrow j + 1$
  until $j = |\mathcal{A}|$ or $\mathcal{A}[i].v \neq \mathcal{A}[j].v$
  $j \leftarrow i$
  repeat
    if $j \neq max_j$ and $|\mathcal{A}[j].l - \mathcal{A}[max_j].l| < W$ and $\mathcal{A}[j].cov_v < C$ **then**
      Delete($\mathcal{A}[j]$)
      return
    end if
    $j \leftarrow j + 1$
  until $j = |\mathcal{A}|$ or $\mathcal{A}[i].v \neq \mathcal{A}[j].v$
  $i \leftarrow j$
until $i = |\mathcal{A}|$

Algorithm 25 : RemoveSpuriousLinks($\mathcal{A}_u$)

for all $e = \langle ch, dir, l, u, d_u, i_u, o_u, cov_u, v, d_v, i_v, o_v, cov_v \rangle$ in $\mathcal{A}_u$ **do**
  if $l < T$ and $cov_v < C$ and ($d_v = in$ and $i_v > 1$ or $d_v = out$ and $o_v > 1$) and ($d_u = in$ and $i_u > 1$ or $d_u = out$ and $o_u > 1$)) **then**
    Delete(e)
    return
  end if
end for

edges that we delete. It allows a single node to select one edge to remove (this prevents unintentional breaking apart of the graph during the editing process), as shown in *Algorithm 26*.

Algorithm 26 : GetDeletions

$\langle manip \rangle \quad \overleftarrow{\textbf{GetDeletions}} \quad \mathcal{A} : \langle ch, dir, l, \mathbf{u}, d_u, i_u, o_u, cov_u, v, d_v, i_v, o_v, cov_v \rangle$

if ¬ **RemoveTips**($\mathcal{A}$)
else if ¬ **RemoveSingletons**($\mathcal{A}$)
else if ¬ **RemoveBubbles**($\mathcal{A}$)
else RemoveSpuriousLinks($\mathcal{A}$)

2.7.3 An Iterative Algorithm

After removing these edges, we might have created new chains in the graph. For this reason, we wish to merge adjacent edges in the graph and update the graph representation. This process involves identifying nodes that wish to center such operations, finding an independent set of such operations, and then merging edges adjacent to this set of nodes. Again, this process is described in detail in *Chapter 3*.

In the original Velvet paper, each type of correction, removing tips, removing bubbles, and removing spurious links – were applied in a single pass. Instead, we propose an iterative application of the above rules, as removing some erroneous edges that fit the above pattern might uncover additional such edges. We find in the observed execution of the algorithm on experimental data, four iterations are required before the algorithm exits. The iterative algorithm is given in *Algorithm 27*.

Algorithm 27 : CleanErrors

repeat
  $s \leftarrow |\mathcal{T}|$
  $\mathcal{M}_\mathcal{T}$ $\overleftarrow{\textbf{GetDeletions}}(\mathcal{T})$
  $\mathcal{M}_\mathcal{C}$ $\overleftarrow{\textbf{GetDeletions}}(\mathcal{T})$
  $\mathcal{T}$ $\overleftarrow{\textbf{UpdateTopology}}(\mathcal{T}, \mathcal{M}_\mathcal{T})$
  $\mathcal{C}$ $\overleftarrow{\textbf{UpdateChains}}(\mathcal{C}, \mathcal{M}_\mathcal{C})$
  ReduceGraph<I>
until $s = |\mathcal{T}|$

2.8 Endpoint Merging

The final graph editing operation described here is endpoint merging, for which we wish find pairs of endpoints (nodes u and v with $|\mathcal{A}_u| = 1$ and $|\mathcal{A}_v| = 1$) whose ends uniquely overlap by $ov_{u,v} \geq h$ bases.[3] We can think of the graph $G_m = \{V_m, E_m\}$ where $u \in V_m$ if and only if $|\mathcal{A}_u| = 1$ in the k-string graph, and $(u, v) \in E_m$ if and only if u and v have an overlap as defined previously. We merge nodes u and v and their corresponding edges in the k-string graph if and only if they are nodes in a connected component with cardinality two in G.

When concatenating chains on the edges connecting u and v, we insert temporary padding characters. As shown in *Fig. 2.6*, we replace these placeholders after the chains have been fully merged.

This merging of overlapping endpoints increases the *effective coverage* of the data from $\frac{(l-k)c}{l}$ to $\frac{(l-h)c}{l}$ for most of the genome. If the low coverage region in the data happens to corresponds to repeats of length less than $k-1$ but greater than h that happen to occur in multiple low coverage regions (thus on multiple endpoints of the graph), the size of the connected component in the graph G_m is greater than two and we cannot merge.

We identify endpoints with *Algorithm 28*.

Algorithm 28 : GetEndpoint

$\langle ch, u, dir \rangle$ $\overleftarrow{\textbf{GetEndpoint}}$ $\mathcal{A} : \langle ch, dir, l, \mathbf{u}, d_u, i_u, o_u, cov_u, v, d_v, i_v, o_v, cov_v \rangle$

if $|\mathcal{A}| = 1$ and $(\mathcal{A}[1].i_v + \mathcal{A}[1].o_v > 1$ or $\mathcal{A}[1].u < \mathcal{A}[1].v)$ **then**
  $\overleftarrow{\textbf{Emit:}}$ $\langle u, ch, dir \rangle$
end if

Once endpoints have been identified, we wish to find the last $(k+1)$-molecule of the chain. We proceed in two steps, first finding the molecule's identifier, and then finding the molecule using the mapping saved when applying the **Assign** function to the original list of $(k+1)$-molecules. For each $(k+1)$-molecule, we generate tuples for each of the $(k+1-h)$ h-molecules. We give this process in *Algorithms 29* and *30*.

We next identify unique endpoint pairs. First, we generate all pairs sharing some h-molecule

[3] Where $\log_4(g) < h < k-1$, h a lower bound on the size of overlap we deem significant.

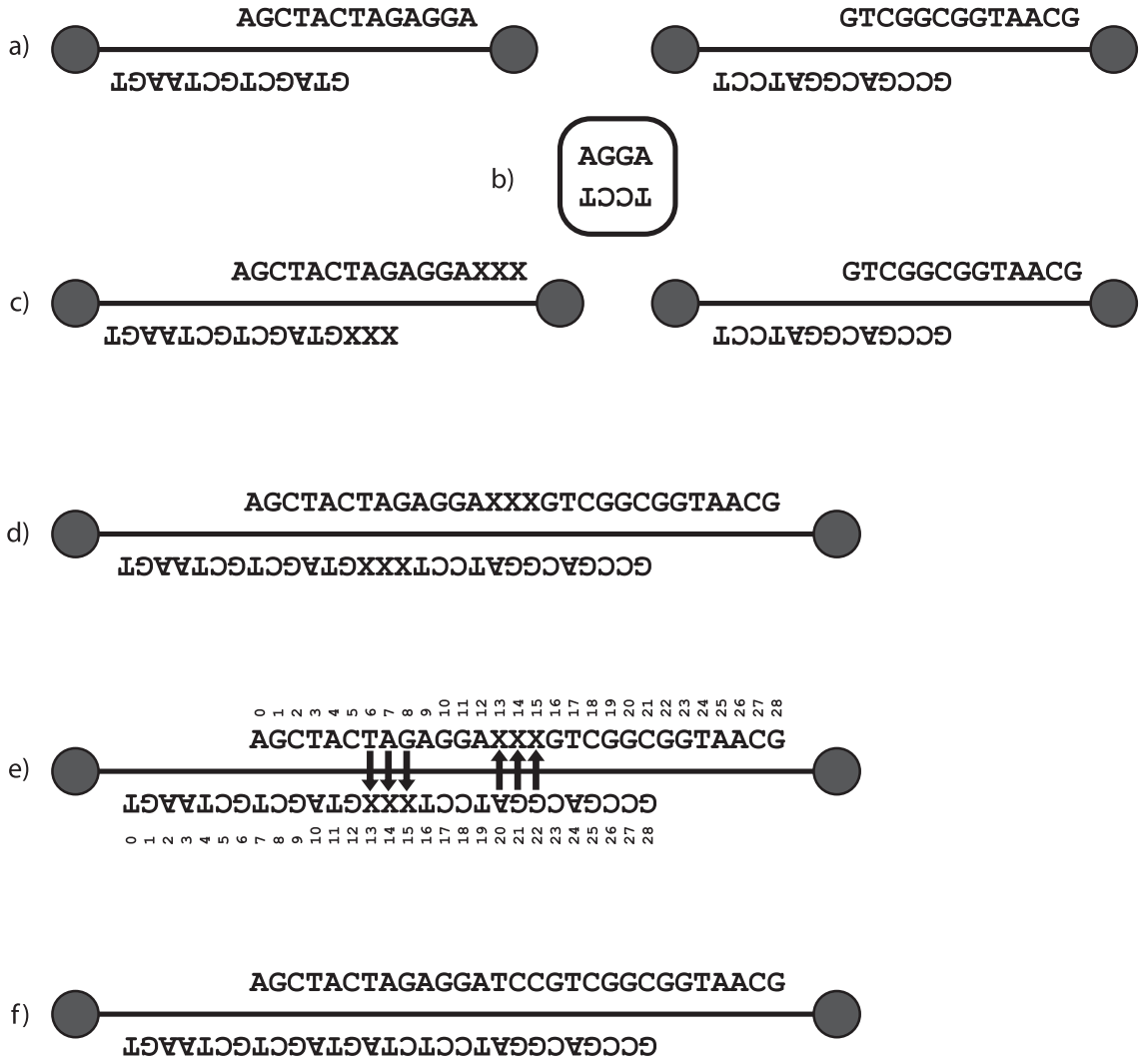

Figure 2.6 The process of merging endpoints in the graph that uniquely overlap by less than $k-1$ characters for $k = 8$. a) Two endpoints to be merged. b) The suffix-prefix overlap. c) We pad one of the two edges with X's. d) We merge edges. e) The ranks of characters used to clean the chain. f) The result of the merger.

Algorithm 29 : GetEndpointMoleculeID

$\langle u, e \rangle$ $\overleftarrow{\textbf{GetEndpointMoleculeID}}$ $\mathcal{A} : \langle \textbf{ch}, e, p_F, p_R, c_F, c_R \rangle$
$\mathcal{B} : \langle \textbf{ch}, u, dir \rangle$

for all $\langle ch, u, dir \rangle$ in $\mathcal{B}$ **do**
  $e \leftarrow$ **if** $dir = F$ **then** $\mathcal{B}[1].e$ **else** $\mathcal{B}[|\mathcal{B}|].e$
  $\overleftarrow{\textbf{Emit:}}$ $\langle u, e \rangle$
end for

Algorithm 30 : GetEndpointMolecule

$\langle u, (k+1)\text{-molecule}, h\text{-molecule}\rangle$ $\overleftarrow{\textbf{GetEndpointMolecule}}$ $\mathcal{A} : \langle \mathbf{e}, (k+1)\text{-molecule}\rangle$
$\mathcal{B} : \langle u, \mathbf{e}\rangle$

$count \leftarrow k - h + 1$
for all $\langle u, e\rangle$ in $\mathcal{B}$ **do**
  $M \leftarrow (k+1)$-molecule
  for $i = 1$ to $count$ **do**
    $\overleftarrow{\textbf{Emit:}}$ $\langle u, M, M[i, i+h-1]\rangle$
  end for
end for

as possible pairs. Then, for all pairs so identified, we compute the exact suffix-prefix overlap length, and keep a pair if it has an exact overlap of length at least h (*Algorithm 31*). Finally, we keep only pairs that are unique, as previously defined (*Algorithms 32* and *33*).

Algorithm 31 : GenerateEndpointPairs

$\langle u, v, overlap\rangle$ $\overleftarrow{\textbf{GenerateEndpointPairs}}$ $\mathcal{A} : \langle u, (k+1)\text{-molecule}, h\textbf{-molecule}\rangle$

for $i = 1$ to $|\mathcal{A}|$ **do**
  for $j = i$ to $|\mathcal{A}|$ **do**
    if $\mathcal{A}[i].u \neq \mathcal{A}[j].u$ **then**
      $overlap \leftarrow GetOverlap\,(\mathcal{A}[i].(k+1)\text{-molecule}, \mathcal{A}[j].(k+1)\text{-molecule})$
      if $overlap \geq h$ **then**
        $\overleftarrow{\textbf{Emit:}}$ $\langle \mathcal{A}[i].u, \mathcal{A}[j].u, overlap\rangle$
      end if
    end if
  end for
end for

We must bring together the tuples from $\mathcal{T}$ that represent the edges to be concatenated. We would be able to merge all endpoints in parallel, except for the rare case when a single edge wishes to be scaffolded on two ends. Thus we must take care when merging endpoints to identify any far endpoints of the corresponding edges that are merge candidates and resolve the conflicts (*Algorithms 34*). We choose between two candidate nodes u and v connected by an edge in the k-string graph by selecting the node with the smaller identifier (*Algorithm 35*). We merge two endpoints only if they both pass the conflict resolution process. To merge the

Algorithm 32 : CheckEndpointU

$\langle u, v, overlap, del\rangle \quad \overleftarrow{\textbf{CheckEndpointU}} \quad \mathcal{A} : \langle \mathbf{u}, v, overlap\rangle$

$del \leftarrow false$
for all $\langle u, v, overlap\rangle$ **do**
  $del \leftarrow$ **if** $\mathcal{A}[1].v \neq v$ **then** true **else** del
end for
for all $\langle u, v, overlap\rangle$ **do**
  $\overleftarrow{\textbf{Emit:}}$ $\langle u, v, overlap, del\rangle$
end for

Algorithm 33 : CheckEndpointV

$\langle u, v, overlap\rangle \quad \overleftarrow{\textbf{CheckEndpointV}} \quad \mathcal{A} : \langle u, \mathbf{v}, overlap, del\rangle$

$del2 \leftarrow false$
for all $\langle u, v, overlap, del\rangle$ **do**
  $del2 \leftarrow$ **if** $\mathcal{A}[1].v \neq v$ **then** true **else** $del2$
end for
if $\neg del2$ **then**
  for all $\langle u, v, overlap, del\rangle$ **do**
    if $\neg del$ **then**
      $\overleftarrow{\textbf{Emit:}}$ $\langle u, v, overlap\rangle$
      $\overleftarrow{\textbf{Emit:}}$ $\langle v, u, overlap\rangle$
    end if
  end for
end if

nodes, we use the **Connect** function described in *Chapter 3* (*Algorithm 36*).

Algorithm 34 : QueryNeighborhood

$\langle u, min, E_u, overlap\rangle$ $\overleftarrow{\textbf{QueryNeighborhood}}$ $\mathcal{A} : \langle ch, dir, l, \mathbf{u}, d_u, i_u, o_u, cov_u, v, d_v, i_v, o_v, cov_v\rangle$
$\mathcal{B} : \langle \mathbf{u}, v, overlap\rangle$

$min \leftarrow$**Min**$(\mathcal{B}[1].u, \mathcal{B}[1].v)$
$\overleftarrow{\textbf{Emit:}}$ $\langle \mathcal{A}[1].u, min, \mathcal{A}[1], \mathcal{A}[1].overlap\rangle$
$\overleftarrow{\textbf{Emit:}}$ $\langle \mathcal{A}[1].v, min, \mathcal{A}[1], \mathcal{A}[1].overlap\rangle$

Algorithm 35 : KeepMinimum

$\langle u, min, E_u, overlap\rangle$ $\overleftarrow{\textbf{KeepMinimum}}$ $\mathcal{A} : \langle \mathbf{u}, min, E_u, overlap\rangle$

if $|\mathcal{A}| = 1$ **then**
  $\overleftarrow{\textbf{Emit:}}$ $\mathcal{A}[1]$
else
  if $\mathcal{A}[1].min < \mathcal{A}[2].min$ **then**
    $\overleftarrow{\textbf{Emit:}}$ $\mathcal{A}[1]$
  else
    $\overleftarrow{\textbf{Emit:}}$ $\mathcal{A}[2]$
  end if
end if

Algorithm 36 : MergeEndpointPair

$manip$ $\overleftarrow{\textbf{MergeEndpointPair}}$ $\mathcal{A} : \langle u, \mathbf{min}, E_u, overlap\rangle$

if $|\mathcal{A}| = 4$ **then**
  $E_1 \leftarrow \mathcal{A}[1].E_u$
  $E_2 \leftarrow$ **if** $\mathcal{A}[2].E_u \neq E_1$ **then** $\mathcal{A}[2].E_u$ **else** $\mathcal{A}[3].E_u$
  Connect$(E_1, E_2, k - 1 - \mathcal{A}[1].overlap)$
end if

When connecting two nodes, we add $(k - 1 - overlap)$ bases of padding between the concatenated edges as the character 'X', as shown in *Fig. 2.6*. We then update the chains after merging, replacing the 'X' placeholder with the appropriate character, as given by *Algorithm 37*.

Finally, we give in *Algorithm 38* a full method for merging endpoints. We use the graph editing support described in Chapter 3. The algorithm iterates until no additional merges have

Algorithm 37 : CleanChains

$\overleftarrow{\textbf{CleanChains}}\quad \mathcal{A} : \langle \textbf{ch}, e, p_F, p_R, c_F, c_R \rangle$

SortByP$_F(\mathcal{A})$
for $i = 1$ to $|\mathcal{A}|$ **do**
  $\mathcal{A}[i].c_F \leftarrow$ **if** $\mathcal{A}[i].c_F = X$ **then Complement**$(\mathcal{A}[i+k-1].c_R)$ **else** $\mathcal{A}[i].c_F$
  $\mathcal{A}[i].c_R \leftarrow$ **if** $\mathcal{A}[i].c_R = X$ **then Complement**$(\mathcal{A}[i-k+1].c_F)$ **else** $\mathcal{A}[i].c_R$
end for

been identified, in order to successfully handle the rare case of conflicting candidate mergers described previously.

Algorithm 38 : MergeEndpoints

repeat
  $s \leftarrow |\mathcal{T}|$
  $\mathcal{E}$ $\overleftarrow{\textbf{GetEndpoint}}(\mathcal{T})$
  $\mathcal{ID}$ $\overleftarrow{\textbf{GetEndpointMoleculeID}}(\mathcal{C}, \mathcal{E})$
  $\mathcal{K}$ $\overleftarrow{\textbf{GetEndpointMolecule}}(\mathcal{K}, \mathcal{ID})$
  $\mathcal{P}$ $\overleftarrow{\textbf{GenerateEndpointPairs}}(\mathcal{K})$
  $\mathcal{P}'$ $\overleftarrow{\textbf{CheckEndpointU}}(\mathcal{P})$
  $\mathcal{P}$ $\overleftarrow{\textbf{CheckEndpointV}}(\mathcal{P}')$
  $\mathcal{O}$ $\overleftarrow{\textbf{QueryNeighborhood}}(\mathcal{T}, \mathcal{P})$
  $\mathcal{O}'$ $\overleftarrow{\textbf{KeepMinimum}}(\mathcal{O})$
  $\mathcal{M}_\mathcal{T}$ $\overleftarrow{\textbf{MergeEndpointPair}}(\mathcal{O}')$
  $\mathcal{M}_\mathcal{C}$ $\overleftarrow{\textbf{MergeEndpointPair}}(\mathcal{O}')$
  $\mathcal{T}$ $\overleftarrow{\textbf{UpdateTopology}}(\mathcal{T}, \mathcal{M}_\mathcal{T})$
  $\mathcal{C}$ $\overleftarrow{\textbf{UpdateChains}}(\mathcal{C}, \mathcal{M}_\mathcal{C})$
  $\overleftarrow{\textbf{CleanChains}}(\mathcal{C})$
until $s = |\mathcal{T}|$

2.9 Contigs as Edges

Once we have constructed the graph, compacted the chains, finished error identification, and merged endpoints, we can output contigs as edge labels. In that sense we have already achieved a limited assembly. However, we wish to mention a detail about the process of reporting the contig in our graph representation, as the process is slightly complicated by the fact that the chains record DNA strands offset by $k-1$ characters. From each chain of length l, we can reconstruct a DNA molecule with $(l+k-1)$ nucleotides, where l is the length of

the chain. We do this by reading l nucleotides from the chain in one direction as s, reading $(k-1)$ nucleotides in the opposite direction as e, taking the reverse complement of e as e', and reporting the sequence as se'.

CHAPTER 3. GRAPH SIMPLIFICATION AND TRAVERSAL

In this chapter, we describe two methods for assembly (one for the assembly of transcriptomes, and the other for the assembly of genomes given paired reads) that demonstrate two broad approaches for reconstructing a genome using an assembly graph. The first is *graph simplification*, in which the number of nodes in the graph is reduced through the application of a set of heuristic edit operations. This approach is explored by Pevzner in his EULER-DB assembler [48] and in the ABySS short sequence assembler [60].[1] After path simplification, edges in the graph correspond to assembled contigs. The second is *path traversal*, in which a path in the graph is found. We can traverse the graph using a whole graph approach, such as the maximum flow/Chinese postman tour formulations described by Myers [44] and Medvedev *et. al.* [41] and the Eulerian superpath problem proposed by Pevzner *et al.* [50], or through a heuristic path discovery algorithm like the ones presented in the Velvet [71] paper and the genome assembler described here. Using path traversal, an assembled contig is some combination of graph labels on the path, in our case the concatenation of edge labels.

The primary strength of graph simplification is that it allows simple rules to be used to process the graph. For example, reasonable rules like the loop reduction rule presented in the first section of this chapter might, after application, unmask motifs in the graph that can then be further simplified. In this way, by repeatedly applying simple transformations to the graph we can ultimately process complex structures. Thus it is easier to design a decision algorithm than in the case where we are processing the original graph *as is*, as done in traversal methods. On the other hand, when we perform a graph simplification, we run the risk of oversimplification, or applying a rule too liberally. This might introduce error in the

[1] Sometimes the lines are blurred. For example, the ABySS assembler finds and merges edges with strong clone pair links, and then continues with a path-walking approach.

assembly. For example, the loop reduction rule is valid as long as the loop is taken every time that motif is is visited in a graph traversal; if there exists a traversal of the motif that skips the loop, then the reduction is invalid.

The primary strength of a path traversal algorithm is that it often uses a more complete picture of the data when making a decision about which edge to include next in a particular contig, when compared to a simple reduction rule. On the other hand, it also has a more complicated decision to make. As we show in the *Chapter 4*, for some relatively simple datasets, the graph simplification and path traversal methods produce similar results, making this difference partially academic. One benefit of the path traversal algorithm that we have developed is that it easily incorporates any number of insertion lengths in the data set, and is in fact the first sequence assembler designed with this feature in mind. As we show in *Chapter 4*, this feature allows for a better assembly of highly repetitive genomes.

3.1 Transcriptome Assembly using Graph Simplification

The first method we describe is a method for the assembly of transcriptomes, using unpaired sequence data. The transcriptome of an organism is the expressed *messenger RNA* (or mRNA) present in the cell. An mRNA is a copy of a gene, often translated into protein via a molecular machine called a ribosome.

In the cell, RNA polymerase (a molecular machine that synthesizes an RNA polymer) transcribes mRNA from the DNA in the genome (hence the name *transcriptome*). The transcribed RNA is then polyadenylated; a chain of deoxyadenine triphosphate (dATP) called the poly-A tail is added to the 3' end of the transcript. Each time a ribosome translates a protein from the mRNA, this poly-A tail shortens, and after it has shortened substantially, the RNA is destroyed.

To assemble a transcriptome, we do not directly assemble the RNA, but rather create a complementary DNA copy of the RNA, called a cDNA. The cDNA is created by attaching a primer to the poly-A tail of the mRNA and then an enzyme called reverse transcriptase synthesizes a DNA molecule adjacent to the RNA. An enzyme called Rnase then removes the

RNA. Finally, using a loop in one end of the DNA as a primer, a DNA polymerase synthesizes a second strand of DNA. This DNA is cloned into bacteria, where it is replicated when the bacteria reproduces.

In a transcriptome, the different genes of an organism can have vastly different expression levels, or copy counts. We can normalize a cDNA library [61], causing the varying expression levels among the genes to be equalized in the library. In our method, we make use of varying expression levels; thus we wish the cDNA library to be unnormalized or only partially normalized. Biologists can use unnormalized cDNA libraries as an alternative to microarrays for measuring gene expression, although currently their cost is much higher.

We assume that we have constructed a k-string graph from a cDNA data set, using the approach in *Chapter 2*. In the absence of repeats, each gene from the library would be in a separate connected component of the k-string graph. However, the repeats in the library cause genes to be glued together. Much of the repeat structure of the genome is hidden in the graph, and as a result the length of all chains is less than the transcriptome length g. We wish to perform a sequence of reductions that simultaneously simplifies the graph while expanding the length of all chains to approach the size of g. We achieve this by manipulating the graph with operations centered at nodes.

Consider the adjacency list for a node u, $\mathcal{A}_u$. We can partition $\mathcal{A}_u$ into two sets $\mathcal{I}_u$ and $\mathcal{O}_u$, where $e_i \in \mathcal{I}_u$ if and only if the direction of e_i at u is pointing into u, and $e_i \in \mathcal{O}_u$ if and only if the direction of e_i at u is pointing out of the u. When traversing the graph, if we enter the node u along an edge that is in $\mathcal{I}_u$, we must exit the node in an edge in $\mathcal{O}_u$, and vice versa. This means that for each $e_i \in \mathcal{I}_u$ there are $|\mathcal{O}_u|$ possible continuations, and for each $e_j \in \mathcal{O}_u$ there are $|\mathcal{I}_u|$ possible continuations. Our goal in assembly by graph simplification is to reduce these possibilities by pairing edges in $\mathcal{I}_u$ with edges in $\mathcal{O}_u$.

Each operation used for simplification defines a set of *deleted edges* D_u as a subset of either $\mathcal{I}_u$ or $\mathcal{O}_u$, and a set of *modified edges* M_u as a subset of the other partition. There is a onto function from modified edges to deleted edges: $Connect(M_u) \rightarrow D_u$. In other words, we associate each deleted edge with one or more modified edges, while we associate each modified

edge with exactly one deleted edge. After *graph simplification*, we will have merged each modified edge with its corresponding deleted edge. As a result, both the deleted edges and modified edges are no longer in the adjacency list of u. The connection process involves the following updates to the k-string graph representation: $\{\mathcal{T}, \mathcal{C}\}$:

1. We remove the tuples in $\mathcal{T}$ corresponding to the deleted edges.
2. We update the tuples in $\mathcal{T}$ corresponding to the modified edges with new endpoint and length information, fields $v, d_v, cov_v, o_v, i_v, len$, and dir.
3. We duplicate the chain tuples in $\mathcal{C}$ corresponding to the deleted edges, modifying the fields id, r_1, and r_2 such that each new chain is concatenated to a chain corresponding to a modified edge.
4. We update the chain tuples in $\mathcal{C}$ corresponding to the modified edges with new rank information such that they are properly concatenated with the chains corresponding to a deleted edge.

We will discuss the specific process by which we perform these operations within our computational framework after we describe specific operations used by our graph simplification method.

3.1.1 The Conflict Graph

We term the first operation *Y-to-V* reduction. Its repeated application results in a conflict graph – a graph in which each node with multiple incident edges has multiple in and multiple out edges. Medvedev *et al.* first named this graph in an assembly paper that used the overlap string graph as the base graph [40], but they included the loop reduction in the rules used in its generation. We feel that it is better to describe the loop reduction separately with other heuristic transformations, as it can break the string graph property – that the concatenation of the edges along some path in the graph be the genomic sequence.

Y-to-V Reduction: We show an example of this operation's application in *Fig. 3.1.* We define a Y-node as a node in which $|\mathcal{I}_u| = 1$ and $|\mathcal{O}_u| > 1$ (or vice versa). We consider the

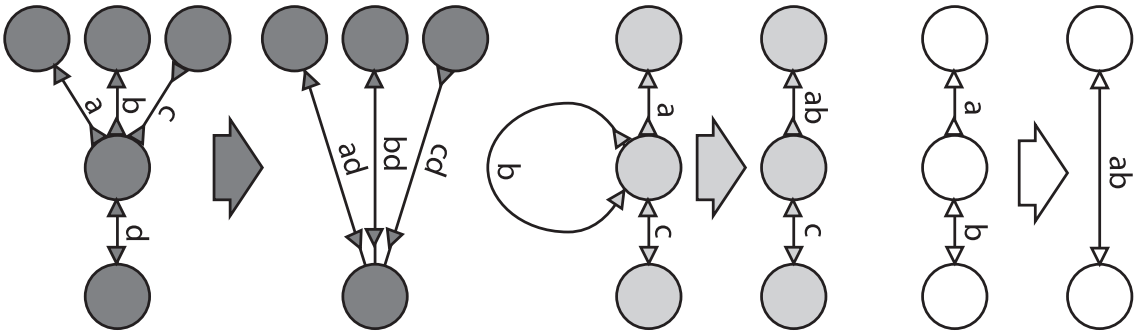

Figure 3.1 Three operations used in sequence assembly by graph simplification. We show the edge labels as characters in the initial motif, and the resulting edge labels as a concatenation of these characters. We show a Y-to-V reduction on the left. We show a loop reduction in the center. We show an I reduction on the right.

case where $|\mathcal{I}_u| = 1$ and $|\mathcal{O}_u| > 1$. We set $\mathcal{D}_u \leftarrow \mathcal{I}_u$ and $\mathcal{M}_u \leftarrow \mathcal{O}_u$. The operation empties the adjacency list for u, and we consider u removed from the graph.

The Y-to-V reduction duplicates repeated elements from the transcriptome in the graph. With each application, the total length of all edge labels in the graph approaches the actual length of the transcriptome but remains bounded by said length. Repeated application of this transformation results in a conflict graph. We show this by contradiction: Consider a graph that can not be operated on by a Y-to-V operation that is not a conflict graph. Then there exists a node u with multiple incident edges and a single in or out edge. u can center a Y-to-V reduction.

Interestingly, we can answer some questions about the graph more easily before transforming the graph to the conflict graph. Inspecting only the topology of the k-string graph, we could identify which sequences were adjacent to the same genomic repeat. When we remove this information from the conflict graph topology, we force inspection of edge labels to discover this information. In another complicating side effect, molecules of length $k+1$ no longer map to unique locations in the graph. A $k+1$ length molecule from a repeated region maps to multiple locations after the transformation.

Importantly, the k-string graph property holds for the conflict graph: some concatenation

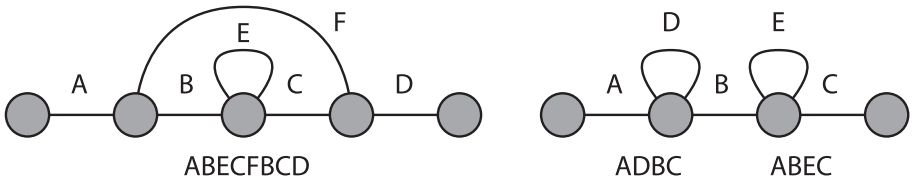

Figure 3.2 Two examples of situations in which the application of loop reduction produces a misassembly. On the left, we show an example genome, with edges labeled with strings. On the right, we show an example transcriptome with two alternative splicings of genes.

of edge labels along a path in the graph corresponds to the genome. We see this by noticing that every path in the k-string graph maps to a path in the conflict graph, and vice versa. This holds because any two edges that are merged by the Y-to-V reduction rule must be traversed together in any traversal of the k-string graph. It also follows from this observation that constructing the conflict graph does not assist a path walking approach to assembly.

Still, constructing the conflict graph allows additional node-centered operations, such as coverage matching described bellow, and our assembly by graph simplification approach relies on it.

3.1.2 Heuristic Graph Simplification

Next we describe a number of rules that are heuristic in nature. Their application could possibly break the k-string graph property, but they are reasonable rules in that we see that it is likely they are correct much of the time. This is shown through experimental validation.

Loop Reduction: The loop reduction operation, as shown in *Fig. 3.1*, was described by Medvedev as a part of the conflict graph generation. As shown in the figure, a loop node has exactly four edges in its adjacency list with two incoming edges and two outgoing edges. One of the incoming and outgoing edges is the same edge; it is a self loop. Exactly one traversal of the loop motif visits all edges: enter the node u, take the loop, and then exit u.

We give an example of a simple genome in which applying the loop rule breaks the k-string graph property in *Fig. 3.2*. The counter example demonstrates that an assumption made

when applying the loop reduction rule does not have to be true; we do not necessarily want to traverse the loop *every* time we enter node u, only at least one of these times. If we enter node u multiple times during graph traversal, we might wish to bypass the loop during some part of the traversal.

This is especially true when processing a transcriptome with alternative splicing. In many eukaryotes, the mRNA goes through an additional phase after translation in which certain regions of the translated mRNA can be thought of as being excised. This process is not deterministic; in some mRNA, the pieces removed might be different than in other mRNA molecules. Ultimately this process allows a smaller number of genes to encode for a larger number of proteins. In humans, perhaps 80% of mRNAs undergo alternative splicing [39]. This process results in a loop motif in the assembly graph. During the assembly of one transcript, we might include the loop, while in the assembly of a second transcript, we might skip it.

In light of these observations, we improve the loop reduction rule considering coverage information. Before applying the loop reduction rule, we perform a check on the coverage of the three edges in the motif. If the loop edge has similar coverage to the other two edges, we apply the rule. If it does not, we do not apply the reduction. We could implement *similar* in many ways, but we choose to check that ratio of coverage on the loop to the coverage on the entry edge is greater than some threshold.

Coverage Matching. We show the application of coverage matching in *Fig. 3.3.* We developed this rule in [31] to use a characteristic of transcriptome data – that expressed genes have varying coverage levels. In it, we attempt to match edges belonging to the same expressed gene.

Consider incoming edge e_i and outgoing edge e_j both adjacent to node u. If $|e_i.cov_u - e_j.cov_u| < T$, where T is some threshold, then we term e_i and e_j *compatible*. If e_i is only compatible with e_j and e_j is only compatible with e_i, then we term them *uniquely compatible*. We now define $\mathcal{D}_u$ to be that set of all tuples e_i in $\mathcal{I}_u$ that are uniquely compatible with some edge e_j in $\mathcal{O}_u$ and define $\mathcal{M}_u$ to be the set of all tuples e_j in $\mathcal{O}_u$ that are uniquely compatible with some edge e_i in $\mathcal{I}_u$. In this case, the function $Connect(\mathcal{M}) \rightarrow \mathcal{D}$ is both one-to-one and

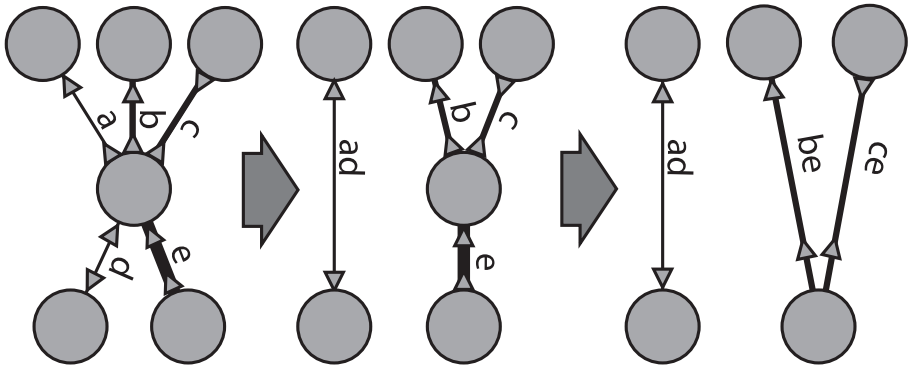

Figure 3.3 We show the coverage matching graph simplification operation, followed by the Y-to-V operation, to demonstrate the iterative nature of graph simplification. Line thickness corresponds to coverage.

onto.

I Reduction: We show the application of this operation in *Fig. 3.1.* We define an I-node as a node u with $|\mathcal{I}_u| = 1$ and $|\mathcal{O}_u| = 1$. This motif does not exist in the original k-string graph after all chains of edges have been compacted, but might arise after other modifications of the graph, such as the loop operation or edge removal during error correction. For the I operation, we set $\mathcal{D}_u \leftarrow \mathcal{I}_u$ and $\mathcal{M}_u \leftarrow \mathcal{O}_u \leftarrow \mathcal{O}_u$. Because the adjacency list for u becomes empty, we consider u removed from the graph.

3.1.3 Graph Reduction on Independent Sets

We apply the described operations to the graph representation $(\mathcal{T}, \mathcal{C})$ using our computational framework. We proceed in a series of iterations. In each iteration we identify nodes that center reductions and carry out those reductions on an independent set of such nodes.

As a first step we find nodes that center operations. We can identify all nodes matching one or more of our reduction operations easily because the motifs are defined by the adjacency information $\mathcal{A}_u$, which we can bring into a single bucket of our topology tuple array $\mathcal{T}$ if we choose the node identifier u as the key. However, we cannot concurrently apply the operations on all of these nodes. If nodes u and v both could center operations, and u and v are directly

connected by an edge in the graph, the operations they center are incompatible. This is because we might wish to remove u from the graph for the operation centered at u, while we might want to make a new edge with u as an endpoint for the operation centered at v.

For this reason during each iteration we choose an independent set of the nodes identified as centering valid operations. An independent set of nodes is a set of nodes such that the induced graph has an empty edge set. Finding a maximum independent set is NP-hard [34] (it is equivalent to finding the maximum sized clique in the complement graph). While a randomized algorithm for finding a *maximal* independent set [42] could be adapted for our purposes, we instead describe a heuristic method that chooses a good independent set assuming that the nodes of the graph have similar degree and the node identifiers are randomly permuted. This method is very simple; we choose a node that wishes to participate as a member of the independent set only if its identifier is smaller than the identifiers of all of its neighbors wishing to participate.

The modifications of the graph are handled through the creation of what we call *topology manipulation tuples* and *chain manipulation tuples*. These tuples carry the information needed to apply the operations to the graph tuple collections $\mathcal{T}$ and $\mathcal{C}$ respectively, as will become clear when we present the psuedocode for the full graph manipulation method. A topology manipulation tuple has the form $\langle ch,\ u,\ delete,\ l_{new},\ d_{new},\ v_{old},\ v_{new},\ d_{vnew},\ i_{vnew},\ o_{vnew},\ cov_{vnew}\rangle$, where:

- ch and u uniquely identify the tuple to be modified.
- $delete$ is a boolean field used to declare if the edge is to be deleted
- l_{new} specifies the new length of the modified edge
- d_{new} specifies the direction to read the corresponding chain of the modified edge
- v_{old} indicates which of the two endpoints to update
- $v_{new}, d_{vnew}, i_{vnew}, o_{vnew}$, and cov_{vnew} specify the new information for that endpoint.

A chain manipulation must carry information on how to update chains in $\mathcal{C}$, and has the form $\langle ch,\ delete,\ reverse,\ ch_{new},\ offset,\ offset2,\ pad \rangle$, where:

- *ch* uniquely identifies the chain to be modified.
- *delete* is a boolean field used to declare if the chain is to be deleted.
- *reverse* is a boolean field used to declare if the chain is to be reversed.
- *offset* is an integer offset to be added to all ranks in the tuples in the chain.
- *offset2* is an integer offset to be added to all ranks in the chain when traversing the chain in the second direction.
- *pad* specifies the number of dummy characters to be added to the end of this chain as padding. For the discussion in this section, this value will always be 0. It only comes into use during endpoint merging as described in *Chapter 2*.

To assist with generating the manipulation tuples described above, we use functions called **Delete** and **Connect**, which take as input graph topology tuples from $\mathcal{T}$ and emit either topology manipulation tuples or chain manipulation tuples, depending on the context in which they were called. Thus we give two functions for Delete and Connect; the version used will be clear from context. They are given as *Algorithms 39, 40, 41*, and *42*.

Algorithm 39 : Delete(edge e)

Emit: $\langle e.ch, e.u, true, 0, 0, 0, 0, 0, 0, 0, 0 \rangle$
Emit: $\langle e.ch, e.v, true, 0, 0, 0, 0, 0, 0, 0, 0 \rangle$

Algorithm 40 : Delete(edge e)

Emit: $\langle e.ch, true, false, 0, 0, 0 \rangle$

Algorithm 41 : Connect(edge m, edge d, pad p)

Emit: $\langle m.ch, m.v, false, m.len + d.len + p, F, m.u, d.v, d.d_i, d.v_i, d.o_i, d.c_i \rangle$
Emit: $\langle m.ch, d.v, false, m.len + d.len + p, R, m.u, d.v, d.d_i, d.v_i, d.o_i, d.c_i \rangle$
Delete(d)

Algorithm 42 : Connect(edge m, edge d, pad p)

$rev_m \leftarrow$ **if** $m.d = F$ **then** false **else** true
$rev_d \leftarrow$ **if** $d.d = R$ **then** false **else** true
$\overleftarrow{\textbf{Emit:}}$ $\langle m.ch, false, rev_m, 0, d.l, 0\rangle$
$\overleftarrow{\textbf{Emit:}}$ $\langle d.ch, false, rev_d, m.l, 0, p\rangle$

We also describe algorithms for updating $\mathcal{T}$ and $\mathcal{C}$ given collections of manipulation tuples. We allow at most one topology manipulation tuple for each topology tuple in $\mathcal{T}$ (*Algorithm 43*). On the other hand we allow multiple chain manipulation tuples for each chain. We produce one (possibly modified) copy of a chain for each chain manipulation tuple not marked as delete. A copy is not made for a chain manipulation tuple marked as delete, however a tuple marked as delete does not affect the copies requested by other manipulation tuples; the other tuples take precedence. At the same time, if all manipulation tuples for a particular chain are marked as delete, then that chain is not copied into the new set of chains (*Algorithm 44.*).

Algorithm 43 : UpdateTopology

$$\langle ch, dir, l, u, d_u, i_u, o_u, cov_u, v, d_v, i_v, o_v, cov_v\rangle$$
$$\overleftarrow{\textbf{UpdateTopology}}$$
$$\mathcal{A} : \langle \mathbf{ch}, dir, l, \mathbf{u}, d_u, i_u, o_u, cov_u, v, d_v, i_v, o_v, cov_v\rangle$$
$$\mathcal{B} : \langle \mathbf{ch}, \mathbf{u}, delete, l, dir, v_{old}, v_{new}, d_{v_{new}}, i_{v_{new}}, o_{v_{new}}, cov_{v_{new}}\rangle$$

if $|\mathcal{B}| = 0$ **then**
  $\overleftarrow{\textbf{Emit:}}$ $\mathcal{A}[1]$
else if $\neg\mathcal{B}[1].delete$ **then**
  $a \leftarrow \mathcal{A}[1]$
  $b \leftarrow \mathcal{B}[1]$
  if $a.u = b.v_{old}$ **then**
    $\overleftarrow{\textbf{Emit:}}$ $\langle b.ch, b.dir, b.l, b.v_{new}, b.d_{v_{new}}, b.i_{v_{new}}, b.o_{v_{new}}, b.cov_{v_{new}}, a.v, a.d_v, a.i_v, a.o_v, a.cov_v\rangle$
  else
    $\overleftarrow{\textbf{Emit:}}$ $\langle b.ch, b.dir, b.l, a.u, a.d_u, a.i_u, a.o_u, a.cov_u, b.v_{new}, b.d_{v_{new}}, b.i_{v_{new}}, b.o_{v_{new}}, b.cov_{v_{new}}\rangle$
  end if
end if

In general, we use the same algorithms when processing different graph manipulation rules above. For this reason, we describe the algorithms using a notation for polymorphism similar to that used for defining templates in the programming language C++. We begin by sending messages to neighbors for use in identifying which nodes should center operations. We emit

Algorithm 44 : UpdateChains

$\langle ch, c_f, c_r, r_1, r_2 \rangle$ $\overleftarrow{\textbf{UpdateChains}}$ $\mathcal{A} : \langle \textbf{ch}, c_f, c_r, r_1, r_2 \rangle$
$\mathcal{B} : \langle \textbf{ch}, delete, reverse, ch_{new}, offset_1, offset_2, pad \rangle$

if $|\mathcal{B}| = 0$ **then**
  for all c in $\mathcal{A}$ **do**
    $\overleftarrow{\textbf{Emit:}}$ c
  end for
else
  for all $\langle ch, delete, reverse, ch_{new}, offset_1, offset_2, pad \rangle$ in $\mathcal{B}$ **do**
    if $delete =$ **false then**
      for all $\langle ch, r_1, r_2, c_1, c_2 \rangle$ in $\mathcal{A}$ **do**
        if $reverse =$ **false then**
          $\overleftarrow{\textbf{Emit:}}$ $\langle ch, r_1 + offset, r_2 + offset2 + pad, c_F, c_R \rangle$
        else
          $\overleftarrow{\textbf{Emit:}}$ $\langle ch, r_2 + offset2, r_1 + offset1, c_R, c_F \rangle$
        end if
      end for
      for $i = 1$ to pad **do**
        if $reverse =$ **false then**
          $\overleftarrow{\textbf{Emit:}}$ $\langle ch, |\mathcal{A}| + i - 1, offset2 + pad - i, 'X', 'X' \rangle$
        else
          $\overleftarrow{\textbf{Emit:}}$ $\langle ch, offset2 + pad - i, |\mathcal{A}| + i - 1, 'X', 'X' \rangle$
        end if
      end for
    end if
  end for
end if

pairs of node ids in *Algorithm 45*.

Algorithm 45 : AddressNeighbors<Op>

$\langle u_1, u_2 \rangle$ $\overleftarrow{\textbf{AddressNeighbors} < \textbf{Op} >}$ $\mathcal{A} : \langle ch, dir, l, \mathbf{u}, d_u, i_u, o_u, cov_u, v, d_v, i_v, o_v, cov_v \rangle$

if Check<Op>$(\mathcal{A}_u)$ **then**
  $\overleftarrow{\textbf{Emit:}}$ $\langle u, u \rangle$
  for all $\langle ch, dir, l, \mathbf{u}, d_u, i_u, o_u, cov_u, v, d_v, i_v, o_v, cov_v \rangle$ **do**
    $\overleftarrow{\textbf{Emit:}}$ $\langle v, u \rangle$
  end for
end if

Before we give the formal algorithms for identifying and applying the four graph operations – YtoV, Loop, I, and Coverage – we give templatized algorithms for modifying the graph. We give a single algorithm for both types of manipulation tuples, making use of the delete and connect polymorphism defined previously. The type of tuple emitted by a particular call to GetManipulation should be clear from context.

Algorithm 46 : GetManipulation<Op>

$\langle manip \rangle$ $\overleftarrow{\textbf{GetManipulation} < \textbf{Op} >}$ $\mathcal{A} : \langle u1, u2 \rangle$
$\mathcal{B} : \langle ch, dir, l, \mathbf{u}, d_u, i_u, o_u, cov_u, v, d_v, i_v, o_v, cov_v \rangle$

if $|\mathcal{A}| > 0$ **then**
  for all $\langle u_1, u_2 \rangle$ in $\mathcal{A}$ **do**
    if $u_2 < u_1$ **then**
      return
    end if
  end for
  Process<Op>$(\mathcal{B})$
end if

With the framework is in place, we give the specialized algorithms for identifying nodes centering operations as *Algorithms 49, 50, 51,* and *52.* We give the specialized algorithms for processing the nodes centering operations as *Algorithms 53, 54, 55,* and *56.*

Algorithm 47 : ModifyGraph<Op>

repeat
  $\mathcal{A} \overleftarrow{\textbf{AddressNeighbors} < \textbf{Op} >}(\mathcal{T})$
  $\mathcal{M}_{\mathcal{T}} \overleftarrow{\textbf{GetManipulation} < \textbf{Op} >}(\mathcal{A}, \mathcal{T})$
  $\mathcal{M}_{\mathcal{C}} \overleftarrow{\textbf{GetManipulation} < \textbf{Op} >}(\mathcal{A}, \mathcal{T})$
  $\mathcal{T} \overleftarrow{\textbf{UpdateTopology}}(\mathcal{T}, \mathcal{M}_{\mathcal{T}})$
  $\mathcal{C} \overleftarrow{\textbf{UpdateChains}}(\mathcal{C}, \mathcal{M}_{\mathcal{C}})$
until $|\mathcal{A}| = 0$

Algorithm 48 : ReduceGraph

repeat
  $s \leftarrow |\mathcal{T}|$
  ModifiyGraph<YtoV>
  ModifiyGraph<Loop>
  ModifiyGraph<I>
  ModifiyGraph<Coverage>
until $|\mathcal{T}| = s$

Algorithm 49 : Check<YtoV>($\mathcal{A}_u$)

$i \leftarrow 0$
$o \leftarrow 0$
for all $\langle ch, dir, l, \mathbf{u}, d_u, i_u, o_u, cov_u, v, d_v, i_v, o_v, cov_v \rangle$ in $\mathcal{A}$ **do**
  $i \leftarrow$ **if** $d_u = in$ **then** $i + 1$ **else** i
  $o \leftarrow$ **if** $d_u = out$ **then** $o + 1$ **else** o
end for
return $o = 1$ and $i > 1$ or $o > 1$ and $i = 1$

Algorithm 50 : Check<Loop>($\mathcal{A}_u$)

$i \leftarrow 0$
$o \leftarrow 0$
$l \leftarrow -1$
for all $\langle ch, dir, l, \mathbf{u}, d_u, i_u, o_u, cov_u, v, d_v, i_v, o_v, cov_v \rangle$ in $\mathcal{A}$ **do**
  $i \leftarrow$ **if** $d_u = in$ **then** $i + 1$ **else** i
  $o \leftarrow$ **if** $d_u = out$ **then** $o + 1$ **else** o
  $l \leftarrow$ **if** $u = v$ **then** u **else** l
end for
return $o = 2$ and $i = 2$ and $l \neq -1$

Algorithm 51 : Check<I>($\mathcal{A}_u$)

return $|\mathcal{A}| = 2$ and $\mathcal{A}[1].d_u \neq \mathcal{A}[2].d_u$

Algorithm 52 : Check<Coverage>($\mathcal{A}_u$)

declare I, O as multimaps
for $i = 1$ to $|\mathcal{A}|$ **do**
  for $j = 1$ to $|\mathcal{A}|$ **do**
    if $i \neq j$ and $\mathcal{A}[i].d_u \neq \mathcal{A}[j].d_u$ **then**
      if $\mathcal{A}[i].d_u = in$ **then**
        $I[\mathcal{A}[i].u].add(\mathcal{A}[j].u)$
        $O[\mathcal{A}[j].u].add(\mathcal{A}[i].u)$
      else
        $O[\mathcal{A}[i].u].add(\mathcal{A}[j].u)$
        $I[\mathcal{A}[j].u].add(\mathcal{A}[i].u)$
      end if
    end if
  end for
end for
for all $\langle ch, dir, l, \mathbf{u}, d_u, i_u, o_u, cov_u, v, d_v, i_v, o_v, cov_v \rangle$ in $\mathcal{A}$ **do**
  if $|I[u]| = 1$ and $|O[I[u][1]]| = 1$ **then**
    return true
  end if
end for
return false

Algorithm 53 : Process<YtoV>($\mathcal{A}_u$)

$i \leftarrow 0$
$o \leftarrow 0$
for all $\langle ch, dir, l, \mathbf{u}, d_u, i_u, o_u, cov_u, v, d_v, i_v, o_v, cov_v \rangle$ in $\mathcal{A}$ **do**
  $i \leftarrow$ **if** $d_u = in$ **then** $i + 1$ **else** i
  $o \leftarrow$ **if** $d_u = out$ **then** $o + 1$ **else** o
end for
$s \leftarrow nil$
for all e in $\mathcal{A}$ **do**
  $s \leftarrow$ **if** $(i = 1$ and $d_u = in)$ or $(o = 1$ and $d_u = out)$ **then** e **else** s
end for
for all e in $\mathcal{A}$ **do**
  if $(i = 1$ and $d_u = out)$ or $(o = 1$ and $d_u = in)$ **then**
    Connect$(s, e, 0)$
  end if
end for

Algorithm 54 : Process<Loop>($\mathcal{A}_u$)

```
l ← nil
for all e in A do
  l ← if e.u = e.v then e else l
end for
for all e in A do
  if e.d_u ≠ l.d_u then
    Connect(e, l, 0)
    return
  end if
end for
```

Algorithm 55 : Process<I>($\mathcal{A}_u$)

```
Connect(A[1], A[2], 0)
```

Algorithm 56 : Process<Coverage>($\mathcal{A}_u$)

```
declare I, O as multimaps
for i = 1 to |A| do
  for j = 1 to |A| do
    if i ≠ j and A[i].d_u ≠ A[j].d_u then
      if A[i].d_u = in then
        I[A[i]].add(A[j])
        O[A[j]].add(A[i])
      else
        O[A[i]].add(A[j])
        I[A[j]].add(A[i])
      end if
    end if
  end for
end for
for all e in A do
  if |I[e]| = 1 and |O[I[e][1]]| = 1 then
    Connect(e, O[I[e][1]][1], 0)
  end if
end for
```

3.1.4 Graph Simplificaiton in EULER-DB

Before continuing with the next section, in which we describe our method for assembly by graph traversal, we would like to spend a moment discussing an alternative approach to assembling paired data presented by Pevzner *et al.* in the EULER-DB assembler, which was the basis of the EULER-SR short read assembler. This approach has been shown to work very well for the assembly of short genomes, and so we explore its merits and our reasons for choosing a different approach.

The EULER-DB approach to processing reads and paired reads is as follows. For reads, map the read to a path in the graph, and then peel that path out of the graph as a single edge. For read pairs, the process works similarly, but takes into account the following. While a read maps uniquely to a path in the graph, a read pair might map to multiple paths in the graph that satisfy the separation distance expected by the reads. Thus, one peels a path from the graph only if it is the only path separating two reads.

Path peeling is applied, a single path at a time, until there exists no path to be peeled. At this point, assembled contigs correspond to edges in the graph.

We have chosen to use a graph traversal approach to assembly over a graph simplification approach for processing clone pair information. The graph traversal approach gives a more natural way to consider multiple sources of information, such as multiple paired reads with different insertion lengths, while the simplification approach uses only a single piece of information. Instead of relying on a single piece of information when making a decision, we rely on multiple corroborating sources of information, and compare the support we have for a particular path with the support we would expect to see given the data characteristics.

3.2 Genome Assembly using Graph Traversal

In our assembly by graph traversal approach, we find a path through the graph a single edge at a time, greedily extending a path with an edge that best fits the distance constraints in the data. We calculate an extension score for each candidate extension edge, by comparing the *observed support* for that extension with the *expected support* for that extension. We then

choose a candidate edge as the extension if its extension score is substantially better than other candidates.

This method relies on the calculation of distant constraint features associated with pairs of edges in the graph. These features come in two classes. The first, which we term exact traversal constraints, are identified by processing reads in the data. By mapping a read into the graph, we can identify that edge u and edge v should be separated by x bases in the graph traversal. These constraints are roughly equivalent to the read mapping done in the EULER assembler. The second class of feature we term $(k+1)$-pair clusters. The clusters characterize the observed distances between two edges in the graph. We compare these observed distances to what is expected given the characteristics of the experimental data, giving a measure of support for a particular edge. The extension score for a particular edge is calculated by combining information from the features associated with the pairing of that edge and each edge on the current path.

A prior approach most similar to the approach described here, although much simpler as it is intended for data with a single insert length, is the breadcrumb algorithm proposed for the Velvet assembler. When evaluating possible extensions, an edge is chosen if and only if it is the only edge such that a pair in the data maps to that edge and the visited path.

3.2.1 Exact Traversal Constraints

The reads in the sequence data map to paths in the graph, and we translate this mapping into a set of traversal constraints, of the the form $\langle e_i, e_j, d \rangle$, where e_i and e_j are edges in the graph, and d is an integer in the range $[0, l-k]$, where l is the maximum read length,k is the parameter k chosen during graph construction, and d indicates the distance between the two edges as seen in some read. We lose some information in forming the traversal constraints above, as apposed to checking a particular path base-by-base, as is done in the EULER assembler.

A nice side affect of reducing the read mapping to distance constraints is that many reads give the same distance constraint $\langle e_i, e_2, d \rangle$. In this case, we record only a single tuple for that constraint, in effect removing much of the redundant information in the read data. We follow

this pattern of removing redundant information when processing the paired reads in the next section.

We describe how we form the exact constraint tuples from the read data and precisely how we use the exact constraints within our traversal method, after first describing the second class of distance constraints used by our traversal algorithm.

3.2.2 Paired Read Constraints

The reads in the sequence data come in pairs, each read pair coming from the two ends of a sheared DNA fragment. As described in the introduction, fragment lengths fall into a specified range, controlled during the creation of a DNA library. Our method allows for the consideration of multiple such fragment ranges, which we call fragment types, and assumes that pairs of reads are classified according to fragment type.

We allow in our assembler from zero to many fragment types, identified using integers in the range $[0, n_z - 1]$, where n_z is the number of fragment types. Each additional fragment type gives more information when deciding among path extensions. We are not the first method to describe using multiple fragment types in assembly. Chaison *et al.* [7] describe using two fragment types with sizes of 2.5kb and 10kb respectively. The ALLPATHS assembler requires that the reads come in three fragment types; a short type with length around 300 bases with very minimal deviation,[2] a medium fragment type with length 2kb, and a long fragment type with length 10kb. Our assembler is the first to be built with the ability to make full use of a fragment library with one to many fragment types, considering the multiple information provided by the different types in concert when making assembly decisions.

In the bidirected string graph $G = \{E, V\}$, edges e_i and e_j correspond to genomic sequences s_i and s_j, each sequence occurring one or more times in the genome. If the genomic distance between these two sequences falls within some fragment range, there is evidence of these two sequences' relative location in the sequence data. We summarize this information as a set of features we term *partial* $(k+1)$*-pair clusters*, and use these features to finish assembly.

[2]The ALLPATHS assembler, which finds all paths between edges connected by short insertion lengths, requires very little error deviation (they demonstrate 1%), which might be unachievable in experimental data.

Definition 3.2.1. *A* **position** *in the bidirected graph G is a tuple of the form $p = \langle e, f \rangle$, with $e \in E$, $f \in \mathbb{N}$, $0 \leq f < \|e\|$. The field f corresponds to a position along the edge in its forward direction and $\|e\|$ is the length of the edge.*

Observation 3.2.2. *By construction of the string graph, there is a bijection between valid $(k+1)$-molecules in the input and the set of all positions in the graph. Therefore we use $p(m)$ to denote the position corresponding to $(k+1)$-molecule m, and $p(m).e$ and $p(m).f$ to denote the corresponding fields.*

Definition 3.2.3. *A* **read pair** *is a tuple of the form $\langle R_1, R_2, z \rangle$, where R_1 and R_2 are the reads and z is the fragment type.*

Definition 3.2.4. *A* **$(k+1)$-pair** *is a tuple of the form $\pi = \langle m_1, m_2, z \rangle$, where m_1 and m_2 are molecules.*

Observation 3.2.5. *We use $\lceil z \rceil$ to denote the largest possible distance between observed $(k+1)$-molecules when reading the ends of a fragment of type z, and $\lfloor z \rfloor$ to denote the smallest possible distance. If z_{min} is the minimum length of fragment type z, z_{max} the maximum fragment length, and l the maximum read length, $\lfloor z \rfloor = z_{min} - 2l + (k+1)$, and $\lceil z \rceil = z_{max} - (k+1)$.*

Definition 3.2.6. *The set of all $(k+1)$-pairs in the input is the set $\Pi = \{\pi_1, \pi_2, ..., \pi_M\}$, with $\langle m_1, m_2, z \rangle \in \Pi$ if and only if there exists some read pair $\langle R_1, R_2, z \rangle$ with m_1 a sub-molecule of R_1 and m_2 a sub-molecule of R_2.*

Observation 3.2.7. *When we allow duplicates, $M = O\left(N(l-k)^2\right)$, where N is the number of reads and l the maximum read length.*

Definition 3.2.8. *An* **edge traversal** *is a tuple of the form $t = \langle e, d \rangle$, with $e \in E$ and $d \in \{F, R\}$, with F corresponding to traversing the edge in the forward direction, and R in the reverse direction.*

Definition 3.2.9. *A* **path** *is a sequence of edge traversals: $T = \langle t_1, t_2, ..., t_l \rangle$.*

Observation 3.2.10. *In general, edges in the graph can be traversed multiple times, so there could exist t_i and t_j, $i \neq j$ and $e_i = e_j$. We always assume that paths being discussed are valid walks in the string graph, as described in the previous section.*

Consider some $\pi_x = \langle m_{1x}, m_{2x}, z_x \rangle$ and traversal T. Let $\mathcal{L}_x = \{t_i | e_i = p(m_{1x}).e\}$ be the set of edge traversals in T to which m_{1x} maps. Let $\mathcal{R}_x = \{t_j | e_j = p(m_{2x}).e\}$ be the set of all edge traversals to which m_{2x} maps.

Definition 3.2.11. *For each $\langle t_i, t_j \rangle \in \mathcal{L}_x \times \mathcal{R}_x$, $i < j$, the* **observed distance** *of π_x is:*

$$d(\pi_x, t_i, t_j) = \sum_{h=i+1}^{j-1} \|e_h\| + \sigma_i + \sigma_j$$

$$\sigma_i = \begin{cases} p(m_{1x}).f & \text{if } d_i = R \\ \|p(m_{1x}).e\| - p(m_{1x}).f & \text{if } d_i = F \end{cases}$$

$$\sigma_j = \begin{cases} p(m_{2x}).f & \text{if } d_j = F \\ \|p(m_{2x}).e\| - p(m_{2x}).f & \text{if } d_j = R \end{cases}$$

Definition 3.2.12. *π_x* **supports** *T using t_i and t_j if and only if $\lfloor z_x \rfloor \leq d(\pi_x, t_i, t_j)$ and $\lceil z_x \rceil \geq d(\pi_x, t_i, t_j)$.*

Definition 3.2.13. *t_i and t_j are* **supported by** *Π if and only if there exists some π_x that supports the path using t_i and t_j.*

Observation 3.2.14. *We call this support weak because the genomic distance between t_i and t_j can differ from the path distance by as much as $\lceil z_x \rceil - \lfloor z_x \rfloor$.*

Definition 3.2.15. *The* **maximum distance expectation** *for t_i and t_j and some fragment type z, denoted by $\lceil (t_i, t_j, z) \rceil$, is calculated as $\min\left(\lceil z \rceil, \sum_{h=i}^{j} \|e_h\|\right)$.*

Definition 3.2.16. *The* **minimum distance expectation** *for t_i and t_j and some fragment type z, denoted by $\lfloor (t_i, t_j, z) \rfloor$, is calculated as $\max\left(\lfloor z \rfloor, \sum_{h=i+1}^{j-1} \|e_h\|\right)$.*

In general, multiple $(k+1)$-pairs with the same fragment type can support a pair of edge traversals on a path. Moreover, if the path is correct, we would expect that, for all z_h, support for much of the range $[\ \lfloor (t_i, t_j, z_h) \rfloor, \lceil (t_i, t_j, z_h) \rceil\ ]$ to be found in the data, assuming the edges are not very short. We wish to formalize this support expectation.

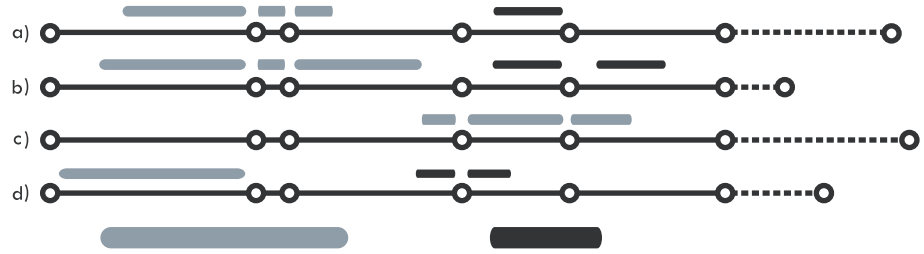

Figure 3.4 Some examples of path extension candidates, with $(k+1)$-pair clusters for two fragment types shown. In a), we show prototypical strong cluster support. In b), we show $(k+1)$-pair cluster support for the extension of a repeat that occurs twice in quick succession in the path. In c), we show an obvious example of lack of support. Finally, in d) we show an example of lack of strong support for the extension, even though the cluster overlaps with expected distance constraints.

Definition 3.2.17. *A* $(k+1)$**-pair cluster** *is a set of observed distances for* t_i, t_j*, and* z*. We summarize a cluster using the range* $\alpha(t_i, t_j, z) = [min, max]$*, with min being the minimum observed distance in the cluster and max the maximum observed distance.*

We construct the $(k+1)$-pair clusters starting from all single element sets taken from Π and proceeding in two phases of merging. In the first phase, we perform single linkage clustering, merging two sets $\alpha_x(t_i, t_j, z)$ and $\alpha_y(t_i, t_j, z)$ if and only if $(max_x + R > min_y) \wedge (min_x - R < max_y)$, for some parameter R. In the second stage, we order all clusters by min, and then, considering all consecutive pairs $(\alpha_x(t_i, t_j, z), \alpha_y(t_i, t_j, z))$ in this ordered set, merge if $max_y - min_x < \lceil z \rceil$.

Definition 3.2.18. t_i *and* t_j *are* **strongly supported** *by a* $(k+1)$*-pair cluster* $\alpha(t_i, t_j, z)$ *if* $\alpha_{min} < \lfloor (t_i, t_j, z) \rfloor + T$ *and* $\alpha_{max} > \lceil (t_i, t_j, z) \rceil - T$*, with* T *a sensitivity parameter.*

The preceding definition is carefully chosen to allow for an edge to be strongly supported even if, for example, t_i is a repeat and occurs at multiple distances from t_j in the genome. For a visual intuition behind the strongly supported definition, see *Fig. 3.4*.

In practice, we wish to be able to answer the question of whether t_i and t_j are strongly supported without having to consider the entire set Π when analyzing a particular path, but

instead preprocess the raw paired reads to extract the necessary features. This needs to be done without any *a priori* knowledge of the nature of the eventual traversal T. In other words, we do not know either the distance between pairs of edges or their relative orientations at the time of summarization.

We achieve this goal by calculating, for each tuple $\langle e_i, e_j, z\rangle$, the ranges of the partial sum $\sigma_i + \sigma_j$ corresponding to each $(k+1)$-pair cluster. As we don't know the relative orientation of the edges at the time of summation, we track the range of $\sigma_i+\sigma_j$ for all four possible orientations of edges, using ff_{min} and ff_{max} to denote this range when e_i and e_j are traversed forwards, with fr_{min}, fr_{max}, rf_{min}, rf_{max}, rr_{min}, and rr_{max} denoting the ranges of other orientations.

Definition 3.2.19. *A* **partial $(k+1)$-pair cluster** *is a summarization of a set of observed partial sums $\sigma_i+\sigma_j$ for edges e_i, e_j and fragment type z, denoted as $\hat{\alpha}(e_i, e_j, z) = \langle rf_{min}, rf_{max}, ff_{min}, ff_{max}\rangle$.*

Observation 3.2.20. *We can calculate the value of rr_{min} as $\|e_i\| + \|e_j\| - ff_{max}$, and rr_{max}, fr_{min}, and can calculate fr_{max} similarly.*

Given a traversal T with edges t_i and t_j, we can calculate the $(k+1)$-pair clusters $\alpha_x(t_i, t_j, z)$ corresponding to partial $(k+1)$-pair cluster $\hat{\alpha}_y(e_i, e_j, z)$. If $t_i.f = F$ and $t_j.f = F$, $min_x = \text{ff}_{miny} + \sum_{h=i+1}^{j-1} \|e_h\|$ and $max_x = \text{ff}_{maxy} + \sum_{h=i+1}^{j-1} \|e_h\|$. The other orientations of t_i and t_j are handled similarly.

3.2.3 Generating Exact and Approximate Distance Constraints

In describing method for computing exact constraints and partial $(k+1)$-pair clusters from Π, we again use our computational framework. We generate both types of constraints at the same time, storing the constraints in tuples of the form $\langle e_1, e_2, z, \text{ff}_{min}, \text{ff}_{max}, \text{fr}_{min}, \text{fr}_{max}\rangle$, where z is the fragment type in the case of paired constraints. In the case of exact traversal constraints, z is the exact distance constraint for the pair offset by the number of fragment types: $z = d + n_z - 1$, and $\text{ff}_{min} = \text{ff}_{max}$ and $\text{fr}_{min} = \text{fr}_{max}$.

Assume that we have, for each $(k+1)$-molecule m, a mapping to that molecule's corresponding identifier, originally given by the **Assign** function. We construct a constraint by

combining this information with information in the chains $\mathcal{C}$. First, we generate tuples of the form $\langle m_1, m_2, z\rangle$, where z is the fragment type in the case of paired reads, or the exact distance constraint d offset by $n_z - 1 : z = d + n_z - 1$. The generation algorithm is given as *Algorithm 57*.

Algorithm 57 : GeneratePairs

$\langle m_1, m_2, t\rangle \quad \overleftarrow{\textbf{GeneratePairs}} \quad \mathcal{A} : \langle r_1, r_2, z\rangle$

$c_1 \leftarrow |r_1| - k - 1$
$c_2 \leftarrow |r_1| - k - 1$
for $i = 1$ to c_1 **do**
  for $j = i$ to c_1 **do**
    $\overleftarrow{\textbf{Emit:}}$ $\langle r_1[i, i + k + 1], r_1[j, j + k + 1], i - j + n_z\rangle$
  end for
end for
for $i = 1$ to c_2 **do**
  for $j = i$ to c_2 **do**
    $\overleftarrow{\textbf{Emit:}}$ $\langle r_2[i, i + k + 1], r_2[j, j + k + 1], i - j + n_z\rangle$
  end for
end for
for $i = 1$ to c_1 **do**
  for $j = i$ to c_2 **do**
    $\overleftarrow{\textbf{Emit:}}$ $\langle r_1[i, i + k + 1], r_2[j, j + k + 1], z\rangle$
  end for
end for

Next, we wish to map $(k + 1)$-molecules to positions in the graph. To do this, we will first map the molecules to *ids* (*Algorithms 58* and *59*). We then find the first position in the graph using *Algorithm 60*.

Algorithm 58 : GetFirstID

$\langle e_1, m_2, z\rangle \quad \overleftarrow{\textbf{GetFirstID}} \quad \mathcal{A} : \langle \mathbf{m}, e\rangle$
$\mathcal{B} : \langle \mathbf{m_1}, m_2, z\rangle$

$\overleftarrow{\textbf{Emit:}}$ $\langle e, m_2, z\rangle$

We store the partial $(k + 1)$ pair clusters recording the smaller of the two chain identifiers in the field ch_1 and the larger of the two identifiers in the field ch_2. As mentioned above, we track ranges for two of the four possible orientations of pairs of edges, as shown in *Algorithm*

Algorithm 59 : GetSecondID

$\langle e_1, e_2, z\rangle$ $\overleftarrow{\textbf{GetSecondID}}$ $\mathcal{A} : \langle \mathbf{m}, e\rangle$, $\mathcal{B} : \langle e_1, \mathbf{m_2}, z\rangle$

$\overleftarrow{\textbf{Emit:}}$ $\langle e_1, e, z\rangle$

Algorithm 60 : GetFirstPosition

$\langle ch_1, r_a, r_b, e_2, z\rangle$ $\overleftarrow{\textbf{GetFirstPosition}}$ $\mathcal{A} : \langle ch, \mathbf{e}, r_1, r_2, c_F, c_R\rangle$, $\mathcal{B} : \langle \mathbf{e_1}, e_2, z\rangle$

$\overleftarrow{\textbf{Emit:}}$ $\langle ch, r_1, r_2, e_2, z\rangle$

61.

Algorithm 61 : GetConstraint

$\langle ch_1, ch_2, z, \text{ff}_{min}, \text{ff}_{max}, \text{fr}_{min}, \text{fr}_{max}\rangle$ $\overleftarrow{\textbf{GetConstraint}}$ $\mathcal{A} : \langle ch, \mathbf{e}, r_1, r_2, c_F, c_R\rangle$, $\mathcal{B} : \langle ch_1, r_a, r_b, \mathbf{e_2}, z\rangle$

if $ch_1 < ch$ **then**
  $\overleftarrow{\textbf{Emit:}}$ $\langle ch_1, ch, z, r_b + r_1, r_b + r_1, r_b + r_2, r_b + r_2\rangle$
else
  $\overleftarrow{\textbf{Emit:}}$ $\langle ch_1, ch, z, r_2 + r_a, r_2 + r_a, r_2 + r_b, r_2 + r_b\rangle$
end if

The first reduction algorithm finds a single linkage clustering of the paired constraint. Each cluster is uniquely identified by the triple $\langle ch_1, ch_2, z\rangle$. The clusters we find will be the same independent of the four possible orientations. Therefore, to find clusters that should be merged, we first sort all tuples by ff_{min} and then proceed with *Algorithm 62*.

After all stages have completed, we consider each bucket of partial $(k+1)$-pair clusters with equivalent (e_{ID_i}, e_{ID_j}, z), and merge partial $(k+1)$-pair clusters in accordance with the phase two merging rule described above, updating all minimums and maximums. As the clusters are sorted by ff_{min}, we can achieve this with a single pass through the array on each processor. This algorithm is given as *Algorithm 63*.

As when processing the reads the first time to construct the de Bruijn graph, we process the paired reads in stages to overcome memory limitations. For this reason our constraint generation algorithm, given as *Algorithm 64*, is structurally similar to *Algorithm 3*.

Algorithm 62 : ReduceConstraints

$$\langle ch_1, ch_2, z, \mathrm{ff}_{min}, \mathrm{ff}_{max}, \mathrm{fr}_{min}, \mathrm{fr}_{max}\rangle$$
$$\overleftarrow{\textbf{ReduceConstraints}}$$
$$\mathcal{A} : \langle \mathbf{ch_1}, \mathbf{ch_2}, \mathbf{z}, \mathrm{ff}_{min}, \mathrm{ff}_{max}, \mathrm{fr}_{min}, \mathrm{fr}_{max}\rangle$$
$$\mathcal{B} : \langle \mathbf{ch_1}, \mathbf{ch_2}, \mathbf{z}, \mathrm{ff}_{min}, \mathrm{ff}_{max}, \mathrm{fr}_{min}, \mathrm{fr}_{max}\rangle$$

SortByFFMin($\mathcal{A}$)
SortByFFMin($\mathcal{B}$)
$i \leftarrow 1$
$j \leftarrow 1$
repeat
  if $i \leq |\mathcal{A}|$ and $j \leq |\mathcal{B}|$ **then**
    if $\mathcal{A}[i].\mathrm{ff}_{min} < \mathcal{B}[j].\mathrm{ff}_{min}$ **then**
      $cl \leftarrow \mathcal{A}[i]$
      $i \leftarrow i + 1$
    else
      $cl \leftarrow \mathcal{B}[j]$
      $j \leftarrow j + 1$
    end if
  else if $i \leq |\mathcal{A}|$ **then**
    $cl \leftarrow \mathcal{A}[i]$
    $i \leftarrow i + 1$
  else
    $cl \leftarrow \mathcal{B}[i]$
    $j \leftarrow j + 1$
  end if
  repeat
    while $i \leq |\mathcal{A}|$ and $\mathcal{A}[i].\mathrm{ff}_{min} \leq cl.\mathrm{ff}_{max} + D$ **do**
      $cl.\mathrm{ff}_{max} \leftarrow \mathbf{Max}(cl.\mathrm{ff}_{max}, \mathcal{A}[i].\mathrm{ff}_{max})$
      $cl.\mathrm{fr}_{max} \leftarrow \mathbf{Max}(cl.\mathrm{fr}_{max}, \mathcal{A}[i].\mathrm{fr}_{max})$
      $cl.\mathrm{fr}_{min} \leftarrow \mathbf{Min}(cl.\mathrm{fr}_{min}, \mathcal{A}[i].\mathrm{fr}_{min})$
      $i \leftarrow i + 1$
    end while
    while $j \leq |\mathcal{B}|$ and $\mathcal{B}[j].\mathrm{ff}_{min} \leq cl.\mathrm{ff}_{max} + D$ **do**
      $cl.\mathrm{ff}_{max} \leftarrow \mathbf{Max}(cl.\mathrm{ff}_{max}, \mathcal{B}[j].\mathrm{ff}_{max})$
      $cl.\mathrm{fr}_{max} \leftarrow \mathbf{Max}(cl.\mathrm{fr}_{max}, \mathcal{B}[j].\mathrm{fr}_{max})$
      $cl.\mathrm{fr}_{min} \leftarrow \mathbf{Min}(cl.\mathrm{fr}_{min}, \mathcal{B}[j].\mathrm{fr}_{min})$
      $j \leftarrow j + 1$
    end while
  until $(i > |\mathcal{A}|$ or $\mathcal{A}[i].\mathrm{ff}_{min} > cl.\mathrm{ff}_{max} + D)$ and $(j > |\mathcal{B}|$ or $\mathcal{B}[j].\mathrm{ff}_{min} > cl.\mathrm{ff}_{max} + D)$
  $\overleftarrow{\textbf{Emit:}}$ cl
until $i > |\mathcal{A}|$ and $j > |\mathcal{B}|$

Algorithm 63 : ReduceConstraintsII

$$\langle ch_1, ch_2, z, \mathrm{ff}_{min}, \mathrm{ff}_{max}, \mathrm{fr}_{min}, \mathrm{fr}_{max} \rangle$$
$$\overleftarrow{\textbf{ReduceConstraintsII}}$$
$$\mathcal{A} : \langle \mathbf{ch_1}, \mathbf{ch_2}, \mathbf{z}, \mathrm{ff}_{min}, \mathrm{ff}_{max}, \mathrm{fr}_{min}, \mathrm{fr}_{max} \rangle$$

SortByFFMin($\mathcal{A}$)
$i \leftarrow 1$
repeat
  $cl \leftarrow \mathcal{A}[i]$
  $i \leftarrow i + 1$
  while $i \leq |\mathcal{A}|$ and $\mathcal{A}[i].\mathrm{ff}_{max} - cl.\mathrm{ff}_{min} \leq (\lceil z \rceil - \lfloor z \rfloor)$ **do**
    $cl.\mathrm{ff}_{max} \leftarrow \mathbf{Max}(cl.\mathrm{ff}_{max}, \mathcal{A}[i].\mathrm{ff}_{max})$
    $cl.\mathrm{fr}_{max} \leftarrow \mathbf{Max}(cl.\mathrm{fr}_{max}, \mathcal{A}[i].\mathrm{fr}_{max})$
    $cl.\mathrm{fr}_{min} \leftarrow \mathbf{Min}(cl.\mathrm{fr}_{min}, \mathcal{A}[i].\mathrm{fr}_{min})$
    $i \leftarrow i + 1$
  end while
  $\overleftarrow{\textbf{Emit:}}$ cl
until $i > |\mathcal{A}|$ and $j > |\mathcal{B}|$

Algorithm 64 : ReadPairs

$\mathcal{B} \leftarrow \emptyset$
for s from 1 to S **do**
  $\mathcal{P} \leftarrow \overleftarrow{\textbf{GetPairs}}(s)$
  $\mathcal{I}$ $\overleftarrow{\textbf{GetFirstID}}(\mathcal{M}, \mathcal{P})$
  $\mathcal{I}'$ $\overleftarrow{\textbf{GetSecondID}}(\mathcal{M}, \mathcal{I})$
  $\mathcal{P}$ $\overleftarrow{\textbf{GetFirstPosition}}(\mathcal{C}, \mathcal{I}')$
  $\mathcal{B}'$ $\overleftarrow{\textbf{GetConstraint}}(\mathcal{C}, \mathcal{P})$
  $\mathcal{B}$ $\overleftarrow{\textbf{ReduceConstraints}}(\mathcal{B}, \mathcal{B}')$
end for
$\mathcal{B}$ $\overleftarrow{\textbf{ReduceConstraintsII}}(\mathcal{B})$

3.2.4 Graph Traversal

Given the exact traversal constraints from Section 3.2.1 and the partial $(k+1)$-pair clusters from Section 3.2.2, we wish to find valid walks through the bidirected string graph.

We assign to each edge e an expected traversal bound $b(e)$ by analyzing coverage. When traversing the graph, we keep track of the number of times an edge has been traversed as $c(e)$, and use this information in conjunction with the bound to choose between ambiguous options. Additionally, we restrict traversal to edges with $c(e) < 2b(e)$. Initially $c(e) = 0$ for all edges.

Our method for traversing the graph is one of path extension. Given a likely partial traversal of the graph as a path $T = \langle t_1, t_2, ...t_l \rangle$ with total length greater than the maximum fragment size, we can determine, by looking at the structure of the string graph, a set of possible edge traversals that can serve as extensions of this path: $E = \{t'_1, t'_2, ...t'_h\}, t'_j = \langle e_j, d_j \rangle$. We describe a heuristic method for choosing the best extension from E by choosing the candidate with the most support among the exact distance constraints and $(k+1)$-pair clusters.

First, we will describe the score we create for the simpler exact distance constraints. Specifically, consider some extension t'_j. Let T' be the path created by extending T with t'_j. We denote the distance between t_i and the end of the path as $d(t_i)$.

Definition 3.2.21. *The* **exact expected support** *for t_i and t'_j is:*

$$\gamma^e_{ij} = \begin{cases} 1 & \text{if } d(t_i) < l - k - B_e \\ 0 & \text{otherwise} \end{cases}$$

Where B_e is a parameter.

Definition 3.2.22. *The* **exact observed support** *for t_i and t'_j is:*

$$\omega^e_{ij} = \begin{cases} 1 & \text{if } d(t_i) = z + n_z \text{ for some } \alpha(t_i, t'_j, z) \\ 0 & \text{otherwise} \end{cases}$$

Definition 3.2.23. *The* **exact extension score** *of a candidate t'_j (for $\sum_i \gamma^e_{ij} > 0$) is defined as:*

$$0 \leq ES(t'_j) = \frac{\sum_i \omega^e_{ij}}{\sum_i \gamma^e_{ij}} \leq 1$$

The score derived from $(k+1)$-pair clusters is defined similarly. Again, consider some extension t'_j. Let T' be the path created by extending T with t'_j. We describe the process using $(k+1)$-pair clusters, as it is more natural to do so; these are derived from the partial $(k+1)$-pair clusters for any path T.

Definition 3.2.24. *The* **approximate expected support** *for t_i, t'_j, and fragment type z_v is:*

$$\gamma_{ijv} = \begin{cases} \hat{\gamma}^a_{ijv} & \text{if } \left(\lfloor (t_i, t'_j, z_v) \rfloor < (\lceil z_v \rceil - B_a)\right) \wedge \left(\lceil (t_i, t'_j, z_v) \rceil > (\lfloor z_v \rfloor + B_a)\right) \\ \hat{\gamma}^a_{ijv} & \text{if } t_i \text{ and } t'_j \text{ are strongly supported by some } \alpha(t_i, t'_j, z_v) \\ 0 & \text{otherwise} \end{cases}$$

$$\hat{\gamma}^a_{ijv} = \lceil (t_i, t'_j, z_v) \rceil - \lfloor (t_i, t'_j, z_v) \rfloor$$

Where B_a is a parameter.

Definition 3.2.25. *The* **approximate observed support** *for t_i, t'_j and fragment type z_v is:*

$$\omega^a_{ijv} = \begin{cases} \hat{\omega}^a_{ijv} & \text{if } t_i \text{ and } t'_j \text{ are strongly supported by some } \alpha(t_i, t'_j, z_v) \\ 0 & \text{otherwise} \end{cases}$$

$$\hat{\omega}^a_{ijv} = \min\left(\lceil (t'_i, t'_j, z_v) \rceil, \alpha.max\right) - \max\left(\lfloor (t'_i, t'_j, z_v) \rfloor, \alpha.min\right)$$

Definition 3.2.26. *The* **approximate extension score** *of a candidate t'_j (for $\sum_{i,v} \gamma^a_{ijv} > 0$) is defined as:*

$$0 \leq AS(t'_j) = \frac{\sum_{i,v} \omega^a_{ijv}}{\sum_{i,v} \gamma^a_{ijv}} \leq 1$$

We say that an extension t'_j is *unambiguous* if $AS(t'_j) > 0$ and $ES(t'_j) > 0$ and, for all w with $0 < w \leq h$ and $w \neq j$, $\frac{AS(t'_w)}{AS(t'_j)} < D_a$ and $\frac{AS(t'_w)}{AS(t'_j)} < D_e$ (for some specificity parameters D_a and D_e). If there exists an extension t'_j that is unambiguous, then we append t'_j to the path and continue with traversal. If t'_j does not exist, we consider only those edges with $b(t'_j) > c(t'_j)$,

and look for the existence of an unambiguous extension $\hat{t}'_j$. If neither t'_j nor $\hat{t}'_j$ exist, we stop extension.

We seed paths with some edge e with $\|e\|$ greater than the maximum fragment size and $c(e) = 0$. As it is not obvious, for example, that really long edges are better than moderately long edges as starting points, we simply choose from candidate starting points in increasing order of edge identifier, until none remain.

CHAPTER 4. EXPERIMENTAL RESULTS

In this chapter, we provide the results obtained from our assembler on both experimental and synthetic data. Synthetic data is generated from a known genome, attempting to mimic the properties of experimental data. Experimental data is generated by the Illumina sequencing platform. We discuss our software implementation, data preparation, running time, and assembly quality.

4.1 Software Implementation

We implemented our method using the programming language C++ and the Message Passing Interface (MPI) parallel programming library. We chose C++ because it has well maintained MPI implementations such as MPICH2 [64] and OpenMPI [16] and for fast execution speed after compilation. The software is comprised of approximately 51 files consisting of nearly 12,000 lines and 350,000 characters.

We make heavy use of C++ templates. A C++ template allows the specialization of a particular class or function when combined with different external components, as long as those components adhere to an interface contract. We choose templates over other options such as base classes and inheritance for three primary reasons. One, templates provide performance improvements at the cost of executable size due to a specialized version of the templated class being created for each new component with which it is used. Two, templates allow the choice of either functors (objects acting and functions) or function pointers. Three, templates allow seamless integration with the C++ Standard Template Library (STL).

Although untested, the software was designed such that the user can specify any STL random access container, as long as that container supports being used by an STL sort function.

We chose the standard STL *vector* for testing the assembler, although other possibilities such as the Standard Template Library for Extra Large Data Sets (STXXL) [10] might be substituted. Achieving a genome assembly in secondary storage by integrating with a library such as STXXL is an area of future research.

We find debugging a parallel software implementation challenging, and the organization of our method around a small set of computational concepts simplifies this task considerably, as there are only a few parallel entry points. Specifically, the parallelism in our software is limited to four functions. These functions are blocking and collective: execution halts until all processors call a particular function.

- **Connect:** Sorts a distributed array of tuples (C++ structs).
- **Distribute:** Redistributes the tuples in distributed array $\mathcal{B}$ according to the distribution of a second array $\mathcal{A}$.
- **Assign:** Using a prefix-sum[1], counts the number of unique keys in a distributed array as N and replaces the field ID in each tuple with a unique key in the range $[0, N)$.
- **GetGlobalCount:** Using a prefix-sum, counts the number of elements in a distributed array.

Finally, our implementation requires that all data in the distributed array be stored in a single memory block, without additional pointers to other locations in the heap. In other words, we do not allow a member of a struct used by our library to be a pointer to memory allocated at runtime. This choice was made to simplify the implementation as it allows us to skip the complex marshalling and unmarshalling that would otherwise be needed.[2] A side effect of this choice is that we choose k at compile time rather than run time.

[1] As described in *Section 1.3.2*

[2] Marshalling (also known as serialization) requires that multiple memory blocks composing a single logical object be copied into a single memory block in preparation for transfer across a network. Unmarshalling (deserialization) is the inverse process.

4.2 Experimental Data Acquisition and Preparation

We downloaded experimental Illumina data from the National Center for Biotechnology Information (NCBI) short read archive [68], publicly available from **ftp://ftp.ncbi.nih.gov**. A part of the International Nucleotide Sequence Database Collaboration (INSDC), the NCBI short read archive provides short read data using a data model, data transfer protocol, and accession space (project identifier) shared with other members of the INSDC. Because of the shear size of the short read data and its static nature (once uploaded into the archive, it tends not to change), the data is stored in flat files, in a directory structure aligned with various sequencing projects.

The short read archive provides data originating from Illumina sequencers in the FastaQ file format, an ASCII file format in which each read is given by four lines. The first line starts with a '@' character followed by the read name and other identifying information such as the lane and coordinates of the blot being read. The second line lists the read using the characters $\{A, G, C, T\}$. The third line starts with a '+' character followed by a second identifier. The last line lists the quality scores, with Q ranging from 0 to 40, given as ASCII characters with numerical values equal to $Q + 33$.

We seen in the archive multiple organizations of paired reads. In some cases, they are interleaved in the same file. In some cases, they are appended together as a single longer read. In some cases, they appear on the same line in two separate FastaQ files.

We convert the data from FastaQ format into a proprietary binary file format. As the IBM Blue Gene/L on which we test uses a PowerPC processor that expects a big endian byte order, and development and testing are done on Intel x86 processors that expect little endian, converting between the two representations is important. We store all data in little endian byte order in our binary format, and covert as necessary when reading from a big endian machine. We give the header format of our file using the notation $[NumBytes : Descriptor]$.

$$[4 : \text{HeaderSize}][8 : \text{FileSize}][4 : \text{RunSize}][4 : \text{1Min}][4 : \text{1Max}]...[4 : \text{kMin}][4 : \text{kMax}]$$

We record each paired read in a block of uniform size, known as the run size. We choose a uniform run size because this allows us to exactly decompose file reading between multiple processors through a calculated offset. We calculate the run size by first calculating the amount of space needed to record a particular sequence when two bytes are used per base. A 36 base read, for instance, would require 9 bytes. The run size for this example would be 24 bytes, in the following format:

$$[2 : \text{PairType}][2 : \text{ReadLength}][9 : \text{ReadData}][2 : \text{ReadLength}][9 : \text{ReadData}]$$

4.2.1 Data Trimming

We analyzed raw data composed of 36 base reads from a single Illumina run from the Michael Smith Genome Sciences Center to better understand data trimming. The quality score for each nucleotide in the analyzed data is a vector $\langle Q_A, Q_C, Q_G, Q_T \rangle$, where Q_N is the quality score for calling the nucleotide N, calculated using the following formula, where p is the probability of the nucleotide being N:[3]

$$Q = 10log_{10}\left(\frac{p}{1-p}\right)$$

The Q values are integers in the range $[-40, 40]$, with $Q = -40 \leftrightarrow p = 0$, $Q = 0 \leftrightarrow p = .5$ and $Q = 40 \leftrightarrow p = 1$. To measure the goodness of a base call, we look at the difference between the highest Q value and the second highest Q value. We want this difference to be significant to consider the call to be valid. For our analysis we chose to consider a difference greater than 10 between the maximum Q value and second highest Q value to be ambiguous. This corresponds to an underlying probability difference of between .4 and .5.

To adequately trim Illumina data, we are interested in three questions about the Illumina sequence quality. First, what is the probability that the base call is ambiguous at a particular position? Second, what is the probability that a base call is ambiguous at a particular position, given no ambiguous base calls in a previous position? Third, what is the probability that a

[3]The quality scores in the FastaQ files that range from 0 to 40 came later in the evolution of the Illumina system.

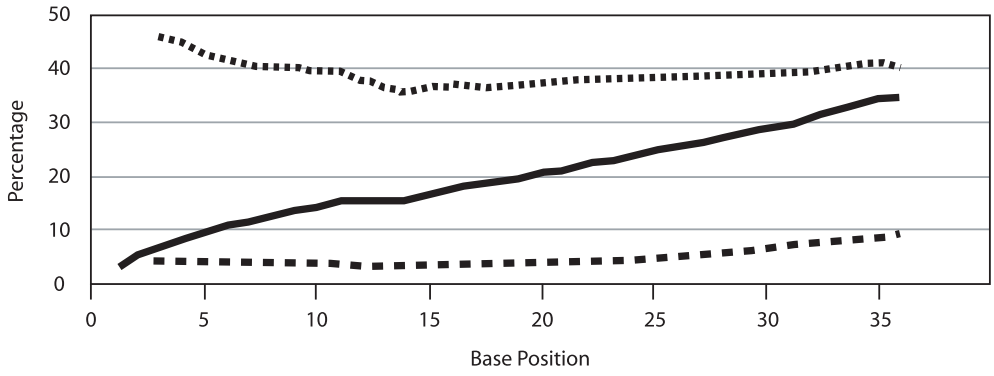

Figure 4.1 Measurement of errors in Illumina data from a single raw Illumina run. For each position in the read, we chart the observed error under three conditions. We chart the percentage of ambiguous calls under the condition there is an ambiguous call earlier in the sequence as the top line. We chart the percentage of ambiguous calls for all sequences as the middle line. We chart the percentage of ambiguous calls under the condition that there are no ambiguous calls in earlier positions as the bottom line.

base call is ambiguous at a particular position, given some ambiguous base call in a previous position?

As shown in *Fig. 4.1*, the conditional probability that a base is ambiguous if we have previously seen an ambiguous base is high in Illumina data. Conversely, the probability that a base is ambiguous given that all previous bases are unambiguous remains low across all positions. From this data, we can infer that once an ambiguous base call is made, whatever condition that initiated this state remains in effect for the remainder of the base calls, causing the rest of the sequence to be unreliable. At the same time, the sequence before this switchover point is of high quality. For this reason, we see the benefit of trimming Illumina data after the quality has degraded.

4.3 Transcriptome Assembly

We generated synthetic data from the genic regions of maize, predicted using FGENESH v.2.6 (using the monocots matrix) on the previously assembled maize genomic islands [14]. We used 61,428 gene structures to generate simulated high coverage transcriptome data. Each

k	Nodes	Edges	Compacted	Reduced
20	20,537,274	20,658,206	451,718	338,121
25	20,717,553	20,741,818	205,858	149,018
30	20,758,869	20,764,256	154,965	114,028

Table 4.1 The effect of varying k on graph size. We show the number of nodes in the initial de Bruijn graph, the number of edges in the initial de Bruijn graph, the number of compacted edges in the k-string graph, and the number of reduced edges in the k-string graph, after graph simplification.

gene was sampled at a random coverage between 50x and 1000x using read lengths of 30 to 50 base pairs, resulting in a data set of 925 million reads and 40 billion bases.

We analyzed the effect of varying k on the resulting compacted graph size and hence the quality of the resulting contigs, as shown in *Table 4.3*. As we increase k, we see a significant reduction in the number of final contigs produced by our algorithm, from 338,000 for $k = 20$ to 114,000 for $k = 30$. While the relative difference in the number of unique k-mers does not change much while varying k, the absolute difference in the number of unique k-mers is similar to the absolute difference in the output size, which is significant.

Producing long and correctly assembled contigs that cover most of the genome is the hallmark of a good assembler. When validating genome assemblies, we use the nX length measure, which is the maximum length l such that X percent of the genome is covered by contigs with length at least l. When describing a transcriptome reconstruction, we use a slightly different metric. Instead of the nX score, we measure how well contigs of length X or greater cover the reference genes, but only for genes of length X or longer. We make this modification because only a subset of the reference genes has lengths greater than X, so certainly we will fail to produce a contig of that length for that gene.

For $k = 30$ there were approximately two contigs per reference gene. For validation, we used the BLAST tool to align the assembled contigs to the reference. We post-processed the BLAST results to verify that each contig fully aligned to some predicted gene in the reference. Our

analysis showed that 92% of the contigs correctly aligned back to the reference. The remaining contigs are the result of over-collapsing edges during graph manipulation. Improving this result is an area of ongoing research. We found that approximately 38% of the applicable reference was covered by contigs with length greater than 500. The maximum length contig was 4017. The maximum length gene in the reference was 5704.

4.4 Genome Assembly

4.4.1 Synthetic Data

We experimentally evaluated the proposed method using synthetic data generated from previously assembled genomes, downloaded from the NCBI FTP server. For each genome, we cleaned any ambiguities from the data and concatenated any contigs from the same chromosome, in the order presented in the finished FASTA file. In this way, we generate a contiguous sequence for each chromosome even though the actual data may contain a large number of contigs, possibly scaffolded. From the input chromosomes, we then generated fragments and sampled 30bp to 50bp paired short reads from the ends of the fragments. We used a 0.9% substitution rate and 0.1% deletion rate, for a total error rate of 1%. Fragment sizes were based upon two hypothetical experimental protocols. Protocol I consisted of two fragment types, $\{900 \pm 100, 4300 \pm 600\}$. Protocol II consisted of five fragment types$\{330 \pm 30$, 660 ± 60, 1100 ± 100, 2200 ± 200, $4400 \pm 400\}$. All genomes were sampled at 300-fold coverage.

For validation, we used MUMmer 3.20 [36] to align the finished contigs back to the reference. We choose MUMmer, which uses a suffix tree to quickly seed sequence alignments, primarily because of its speed. Finding all alignments between assembled contigs and a bacterial reference takes under a minute.

The results are presented in Table 4.4.1. We present results on three bacterial genomes as a reference point for comparison with other work on short read assembly. In addition, we present results on *S. cerevisiae* and *D. melanogaster*. For each genome, we present the length of the maximum contig generated, along with n50, n75 and n90 lengths. We also present the number of contigs with length $>$ 10Kb, the percentage of the genome that is covered by these

Organism	Size	Max	n50	n75	n90	Count	Mis	Cov
E. coli	5.4	224	85	43	10	91	0.2	90.0
S. cerevisiae	12.2	225	71	34	11	225	0.8	90.1
C. pneumoniae	1.0	867	867	867	132	2	0.0	99.9
S. pneumoniae	2.1	321	137	92	77	19	0.0	95.5
E. coli	5.4	378	231	104	42	42	0.5	94.0
S. cerevisiae	12.2	290	107	75	25	148	0.8	94.1
D. melanogaster	120.3	855	102	43	12	1,687	1.5	91.2

Table 4.2 Assembly quality for five organisms. The first group shows results for sequences using Protocol I, as described in the text. The second group was assembled from data matching Protocol II. In order, we show the size of the genome in megabases; the maximum, $n50$, $n75$, and $n90$ lengths, all in kilobases; the number of contigs with length $>$ 10Kb, the number of misassemblies per megabase, and percentage of the genome covered by these contigs at $>$ 99.9% identity.

contigs with at least 99.9% identity, and the number of large-scale misassemblies per megabase. Approximately four out of five assembled contigs aligned perfectly to the reference.

For testing the assembler, data was generated on a workstation, using the parameters described above. This data was then transferred to a 512-node Blue Gene/L system with 512 MB memory per node, at which point the parallel phases of the software were run to produce the intermediate string graph and features from paired reads. These results were transferred back to a workstation for production of contigs using bidirected graph traversal. For *Drosophila*, this process took ∼50 minutes for data transfer (depending on network congestion), ∼100 minutes for the parallel phases, and ∼20 minutes for the remaining. In the local processing phase of the algorithm, the running time was dominated by file I/O.

4.4.2 Experimental Data

For comparing our results with those produced by other short sequence assemblers, we downloaded the *E. coli* Illumina data set (Accession SRX000429) used to benchmark previous sequence assemblers in [60]. The data consisted of paired *E. coli* reads, with an insertion length

Assembler	≥ 100	Cov	n50	Max	Mis
OurName	217	98.47	45,592	126,490	758,880
Abyss	233	99.44	45,362	173,852	432,276
Velvet	286	99.81	54,359	164,194	471,204
EULER-SR	216	99.76	57,497	174,041	984,438
SSAKE	931	99.99	11,450	50,668	223,478
Edena	680	99.00	16,430	67,082	79,620

Table 4.3 A comparison of short sequence assemblers, using a 300x coverage Illumina data set with read length 36 and insertion length 200.

of approximately 200 bases. We give a comparison of our assembler to assemblies produced. We can see in *Table 4.4.2* that our assembler achieves a result on par with the ABySS short sequence assembler, and is roughly equivalent to the results obtained by the best assemblers, with the benefit that the solution scales to larger problem sizes.

4.5 Performance Results

While we are most interested in utilizing the large, distributed memory of high performance computers to enable the assembly of large genomes, for completeness we tested the scalability of the implementation. We tested varying the number of processors from 16 to 256 on an IBM Blue Gene/L supercomputer [18] named CyBlue. We ran the assembler with $k = 27$ on an experimental *E. coli* data set downloaded from the NCBI short read archive.

As we use a very structured approach to parallel processing, the scalability testing is really a test of the scalability of the all-to-all implementation and parallel I/O on Blue Gene. The CyBlue system uses the IBM General Parallel File System (GPFS) [54] connected to the system via a single communication pipe, and a three dimensional torus interconnect for all-to-all communication. We ran the system in coprocessor mode, in which a single processor per machine node handles computation and a second processor handles communication. We would expect the following factors to interact in producing the speedup numbers we see.

- The number of processors reading concurrently from the parallel file system doubles.

p	**Init**	**Read**	**Con**	**Wr**	**Clean**	**Wr**	**Pairs**	**Wr**	**Tot**	**-Wr**	**Per**
16	5	469	49	106	23	30	3369	3	4,054	3,915	4,054
32	11	294	26	155	22	45	1760	10	2,323	2,113	2,027
64	11	179	14	62	89	88	910	1	1,354	1,203	1,013
128	9	120	7	59	37	32	390	1	655	563	507
256	13	101	3	104	4	70	190	11	496	311	253

Table 4.4 Running time of the parallel assembler in seconds, broken down by stage of the algorithm. From left to right the columns are: p: the number of processors, **Init**: initialization time, from program startup to initial read, **Read**: read the $(k+1)$-molecules from the data file, **Con**: construct graph tuples and compact edges in the graph, **Wr**: write graph information, **Clean**: perform error correction by graph editing, **Wr**: write graph information, **Pairs**: read paired information and create clusters, **Wr**: write clusters, **Tot**: total running time, **-Wr**: total running time without write phases, and **Per**: perfect speedup.

- The total communication load on the network remains constant, while the load per processor halves.
- The number of processors communicating concurrently doubles.
- The number of processors reading concurrently from the parallel file system doubles.
- The amount of total available memory doubles, halving the number of stages needed to process the input data.
- The number of tuples per processor halves, reducing the log factor in the quick sort implementation by 1.
- The number of processors writing concurrently to the parallel file system doubles.

We timed each stage of the algorithm individually and present the results in *Table 4.5*. As we show in the table, the CyBlue system achieved reasonable scaling on all factors in the above decomposition, save parallel file write, which seemed to slow significantly as we increased the number of processors. The stages of the algorithm not involved with writing achieved a

respectable 12.6 speedup factor in our testing when increasing the number of processors by a factor of 16.

CHAPTER 5. CONCLUSION

In this work, we have described a parallel short sequence assembler and demonstrated the validity of novel assembly methods by assembling both experimental and synthetic short sequence data. As we summarized our contributions in the introductory chapter, we will conclude with eight problems that follow from our research.

Problem 1: *Assembly presentation.* In our discussion of assembly, we presented as our answer a set of contigs, the traditional presentation. By choosing such a presentation, we ignore information. For example, given a contig C, we might know that C is followed by some contig from a set $\mathcal{S}$ and preceded by some contig from a set $\mathcal{P}$; we have not however, discovered the specific contig from these sets. As another example, given a query Q, we can answer the question, "Does Q exist in the genome?" even if Q is not a submolecule of some assembled contig, by looking for a path labeled with Q in the graph. These two examples show the limitations of thinking of an assembly of an unknown genome as a set of contigs. Perhaps the reason we choose to represent the assembly as a set of contigs is that this representation is easy and intuitive. Finding a more complicated and more complete way to present the assembly of an unknown genome is interesting open problem in sequence assembly.

Problem 2: *Integration of coverage information and paired read information for transcriptome assembly and metagenomics.* In *Section 3.1* when we described transcriptome assembly, we hinted at an interesting problem in processing the loop reduction rule due to alternative splicing in eukaryotes. The alternative splicing problem takes a set of reads from a transcriptome and asks us to identify all alternative splicings of each gene as expressed in the data. It is likely that this problem becomes easier with paired read information, but making use of paired reads to answer the problem requires combining coverage information with paired read

processing. This combination and heuristics (or a formulation as an optimization problem) remain an open problem.

The combination of differential coverage and paired read information will likely be necessary in the successful processing of metagenomics data. A metagenomics data set contains read information from a set of organisms rather than a single organism, for example the set of bacteria residing in the gut or in a water sample [5]. Within this set, sometimes called a *metagenome*, organisms will be present at differing frequencies, as is the case for genes in transcriptome data. For example, in a metagenomics study done using a sample from the Sargosa Sea, 1,800 species of microorganism were present a sequences sample with 50-75 of the organisms sequenced at coverages that ranged from 1-fold to 30-fold [65]. A combination of coverage and paired reads will likely be needed to achieve an assembly of metagenomics data; Additionally, work related to Problem 1 will be needed to present the results.

Problem 3: *Using genetic and physical maps during assembly.* Genetic maps, physical maps, and optical maps have previously been used in sequence assembly for creating BAC tiling paths used in hierarchical assembly, for scaffolding contigs, or for independently verifying the assembly. Integrating the information in the genetic or physical map during the assembly process itself, as another traversal constraint, might form an interesting problem to solve.

Problem 4: *Parallel algorithms for the spectral alignment problem for distributed memory machines.* We described the spectral alignment problem in *Section 2.7*. For the spectral alignment problem we are given a set of k-molecules present in a genome (the k-spectrum of that genome), and wish to, for each read in the data set, find the shortest sequence of edits that will transform that read into a read such that each k length submolecule of the read is in the k-spectrum. By presenting parallel methods for error correction, we have given a parallel method for discovering the k-spectrum of an unknown genome. Recently, a method for solving the spectral alignment problem using GPUs was presented [58]. Solving the spectral alignment problem on distributed memory machines remains and interesting open problem to solve.

Problem 5: *Optimal experimental design for de novo assembly.* While our results indicate that having a data set with multiple insertion lengths is better than having data with a single

insertion length, we have not performed experimental analysis of what the best choice of insert length(s) might be. One would expect to want an insert length for every distance between the read length l and the genome length g given infinite coverage. However with finite coverage each additional insertion length dilutes the amount of data allocated to each length, which in turn reduces the covered range for a particular pair of edges (the quality of the $(k+1)$-pair cluster. The best choice might change from organism to organism. It might change for different coverage levels. A thorough exploration of the possibilities using synthetically generated data could inform the design of real sequencing projects.

Problem 6: *Parallel k-string graph traversal.* We give parallel methods for constructing and manipulating the k-string graph and give a parallel method for processing paired reads and finding a set of summary features. We then perform the final phase of the assembler, graph traversal, using a serial traversal algorithm. Parallelizing the final stage of the assembler is an open problem. A straightforward approach to solving this problem would be to do a number of traversals using different seeds concurrently and then combining the results, although a more complicated way to distribute the work of traversal might be found.

Problem 7: *Assembly with heterogeneous reads.* In our experimental results section, we present the assembly of both synthetic and experimental Illumina reads. While it seems likely that we could easily adapt the assembler to make use of heterogeneous reads, we have not yet explored this possibility, either with synthetic or experimental data.

Problem 8: *Quality score awareness.* We consider the reads as a set of strings and, except for the initial trimming, ignore the quality scores. We chose this approach because including quality scores for each position throughout the assembly pipeline makes certain aspects of the assembler more difficult to design. For example, summarizing all k-molecules seen in the data with a single count is no longer possible; we would instead have to summarize the quality score at each position in some way.

We could consider quality scores in the assembly at two points, during the determination of the k-spectrum of the genome or metagenome or during sequence editing. One would watch for systematically low quality scores at a specific motif in the data. We might find quality

scores particularly useful when processing very low coverage genomes, like low copy genes in transcriptomes or low copy genomes in metagenomes.

These eight problems demonstrate that sequence assembly, while an old problem, remains an evolving one, much like the genomes we attempt to reconstruct. Our presentation of a parallel short read assembler for de novo genome reconstruction is a step in the continuing process of finding good methods to solve this problem, which we hope serves as a basis for continued advances in this field.

Bibliography

[1] S.F. Altschul, W. Gish, W. Miller, and M. Myers. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[2] S. Batzoglou, D.B. Jaffe, K. Stanley, J. Butler, S. Gnerre, E. Mauceli, B. Berger, J.P. Mesirov, and E.S. Lander. ARACHNE: a whole-genome shotgun assembler. *Genome Research*, 12:177–189, 2002.

[3] S. Bennet. Solexa ltd. *Pharmacogenomics*, 5(4):433–438, 2004.

[4] D.R. Bentley, S. Balasubramanian, H.P. Swerdlow, and G.P. Smith. Accurate whole human genome sequencing using reversible terminator chemistry. *Nature*, 456:53–59, 2008.

[5] M. Breitbart, P. Salamon, B. Andresen, J.M. Mahaffy, A.M. Segall, D. Mead, F. Azam, and F. Rohwer. Genomic analysis of uncultured marine viral communities. *Proceedings of the National Academy of the Sciences*, 99:14250 – 14255, 2002.

[6] J. Butler, I. MacCallum, M. Kleber, I.A. Shlyakhter, M.K. Belmonte, E.S. Lander, C.N. Nusbaum, and D.B. Jaffe. ALLPATHS: De novo assembly of whole-genome shotgun microreads. *Genome Research*, 18:810–820, 2008.

[7] M.J. Chaisson and P.A. Pevzner. Short fragment assembly of bacterial genomes. *Genome Research*, pages 18:324–330, 2008.

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation*, page 10, 2004.

[9] F. Dehne and S.W. Song. Randomized parallel list ranking for distributed memory multiprocessors. In *Asian Computing Science Conference*, pages 1–10, 1996.

[10] R. Dementiev, L. Kettner, and P. Sanders. STXXL: standard template library of XXL data sets. *Software: Practice and Experience*, 38:589–637, 2007.

[11] I.M. Dew, B. Walenz, and G. Sutton. A tool for analyzing mate pairs in assemblies (TAMPA). *Journal of Computational Molecular Biology*, 12:497–513, 2005.

[12] J.C. Dohm, C. Lottaz, T. Borodina, and H. Himmelbauer. SHARCGS, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing. *Genome Research*, 17:1697–1706, 2007.

[13] A. Edwards, H. Voss, P. Rice, A. Civitello, J. Stegemann, C. Schwager, J. Zimmerman, H. Erfle, C.T. Caskey, and W. Ansorge. Automated DNA sequencing of the human HPRT locus. *Genomics*, 6:593–608, 1990.

[14] S. Emrich, S. Aluru, Y. Fu, T. Wen, M. Narayanan, L. Guo, D. Ashlock, and P.S. Schnable. A strategy for assembling the maize (zea mays l.) genome. *Bioinformatics*, 20:140 – 147, 2004.

[15] B. Ewing and P. Green. Base-calling of automated sequencer traces using phred. *Genome Research*, 8:186–194, 1998.

[16] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[17] J. Gallant, D. Maier, and J.A. Storer. On finding minimal length superstrings. *Journal of Computer and System Sciences*, 20(1):50–58, 1983.

[18] A. Gara, M.A. Blumrich, D. Chen, G.L.T. Chiu, and P. Coteus. Overview of the Blue Gene /L system archetecture. *IBM Journal of Research and Development*, 49(2), 2005.

[19] P. Green. Documentation for Phrap. Technical report, Genome Center, University of Washington, 1996.

[20] P. Havlak, R. Chen, K.J. Durbin, A. Egan, and Y. Ren. The atlas genome assembly system. *Genome Research*, 14:721–731, 2003.

[21] D.R. Helman, J. Ja'Ja', and D.A. Bader. A new deterministic parallel sorting algorithm with an experimental evaluation. Technical Report CS-TR-3670 and UMIACS-TR-96-54, College Park, MD, 1996.

[22] D. Hernandez, P. Francois, L. Farinelli, M. Osteras, and J. Schrenzel. De novo bacterial genome sequencing: Millions of very short reads assembled on a desktop computer. *Genome Research*, 18:802–809, 2008.

[23] S. Hossain, N. Azimi, and S. Skiena. Crystallizing short-read assemblies around lone Sanger reads. *Bioinformatics*, 2009.

[24] X. Huang and A. Madan. CAP3: A whole-genome assembly program. *Genome Research*, 9:868–877, 1999.

[25] X. Huang, J. Wang, S. Aluru, and S.P. Yang. PCAP: A whole-genome assembly program. *Genome Research*, 13(9):2164–2170, 2003.

[26] R.M. Idury and M.S. Waterman. A new algorithm for DNA sequence assembly. *Journal of Computational Biology*, 2:291–306, 1995.

[27] B.G. Jackson and S. Aluru. Parallel construction of bidirected string graphs for genome assembly. In *Proc. 37th International Conf. on Parallel Processing*, pages 346–353, 2008.

[28] B.G. Jackson, S. Aluru, and P.S. Schnable. Consensus genetic maps: A graph theoretic approach. In *Proc. 5th Annual Computational Systems Bioinformatics Conf.*, pages 35–45, 2005.

[29] B.G. Jackson, P.S. Schnable, and S. Aluru. Consensus genetic maps as median orders from inconsistent sources. *IEEE/ACM Trans. on Computational Biology and Bioinformatics*, 5:161–171, 2008.

[30] B.G. Jackson, P.S. Schnable, and S. Aluru. Assembly of large genomes from paired short reads. In *Proc. 1st International Conference on Bioinformatics and Computational Biology*, volume 5462, pages 30–43, 2009.

[31] B.G. Jackson, P.S. Schnable, and S. Aluru. Parallel short sequence assembly of transcriptomes. *BMC Bioinformatics*, 10:S14, 2009.

[32] B.N. Jackson and S. Aluru. *Pairwise Sequence Alignment*, page Chapter 1. 2006.

[33] D.B. Jaffe, J. Butler, S. Gnerre, E. Mauceli, K. Lindblad-Toh, , J.P. Mesirov, M.C. Zody, and E.S. Lander. Whole-genome sequence assembly for mammalian genomes: Arachne 2. *Genome Research*, 13:91–96, 2003.

[34] R.M. Karp. Reducibility among combinatorial problems. In R.E. Miller and J.W. Thatcher, editors, *Complexity and Computer Computations*, pages 85–103. 1972.

[35] J.D. Kececioglu and E.W. Myers. Combinatorial algorithms for DNA sequence assembly. *Algorithmica*, 13(1/2):7–51, 1995.

[36] S. Kurtz, A. Phillippy, A.L. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S.L. Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, 5, 2004.

[37] X. Li, P. Lu, J. Schaeffer, J. Shillington, P.S. Wong, and H. Shi. On the versatility of parallel sorting by regular sampling. *Parallel Computing*, 19(10):1079–1103, 1993.

[38] M. Margulies and M. Egholm. Genome sequencing in open microfabricated high density picoliter reactors. *Nature*, 437(7054):376–380, 2005.

[39] A.J. Matlin, F. Clark, and C.W.J. Smith. Understanding alternative splicing: towards a cellular code. *Nature Reviews*, 6:386–398, 2005.

[40] P. Medvedev and M. Brudno. Ab initio whole genome shotgun assembly with mated short reads. In *Lecture Notes in Computer Science*, volume 4955, pages 50–64, 2008.

[41] P. Medvedev, K. Georgiou, G. Myers, and M. Brudno. Computability of models for sequence assembly. *Lecture Notes in Computer Science*, 4645:289–301, 2007.

[42] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge Press, New York, NY, USA, 1995.

[43] E.W. Myers. Toward simplifying and accurately formulating fragment assembly. *Journal of Computational Biology*, 2:275–90, 1995.

[44] E.W. Myers. The fragment assembly string graph. *Bioinformatics*, 21:ii79–ii85, 2005.

[45] E.W. Myers, G.G. Sutton, A.L. Delcher, and I.M. Dew. A whole-genome assembly of drosophilia. *Science*, 287(5461):2196–2204, 2000.

[46] S. Ossowski1, K. Schneeberger1, R.M. Clark, C. Lanz, N. Warthmann, and D. Weigel. Sequencing of natural strains of *arabidopsis thaliana* with short reads. *Genome Research*, preprint, 2008.

[47] V. Pandey, R.C. Nutter, and E. Prediger. *Applied Biosystems SOLiD System: Ligation-Based Sequencing*. Wiley, 2008.

[48] P.A. Pevzner and H. Tang. Fragment assembly with double-barreled data. *Bioinformatics*, 21:S225–S233, 2001.

[49] P.A. Pevzner, H. Tang, and G. Tesler. De novo repeat classification and fragment assembly. *Genome Research*, 14:1786–96, 2004.

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

[51] M. Ronaghi. Pyrosequencing sheds light on DNA sequencing. *Genome Research*, 11:3–11, 2001.

[52] J.J. Ruan, C.W. Fuller, A.N. Glazer, and R.A. Mathies. Flourescence energy transfer dye-labeled primers for DNA sequencing and analysis. *Proceedings of the National Academy of Sciences*, 95:4347–4351, 1995.

[53] F. Sanger, S. Niclen, and A.R. Coulson. DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences*, 74:5463–5467, 1977.

[54] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the FAST'02 Conference on File and Storage Technologies*, 2002.

[55] R.V. Shankar and S. Ranka. Random data accesses on a coarse-grained parallel machine. II. one-to-many and many-to-one mappings. *Journal of Parallel and Distributed Computing*, 44(1):24–34, 1997.

[56] J. Shendure, G.J. Porreca, N.B. Reppas X. Lin, J.P. McCutcheon, A.M. Rosenbaum, M.D. Wang, K. Zhang, R.D. Mitra, and G.M. Church. Accurate multiplex polony sequencing of an evolved bacterial genome. *Genome Research*, 309:1723–1732, 2005.

[57] H. Shi and J. Schaefer. Parallel sorting by regular sampling. *Journal of parallel and distributed computing*, 14(4):361–372, 1992.

[58] H. Shi, B. Schmidt, W. Liu, and W. Mller-Wittig. Accelerating error correction in high-throughput short-read DNA sequencing data with CUDA. In *Proceedings of the 8th IEEE International Workshop on High Performance Computational Biology*, 2009.

[59] J.F. Sibeyn, F. Guillaume, and T. Seidel. Practical parallel list ranking. *Journal of Parallel and Distributed Computing*, 56:156–180, 1999.

[60] J.T. Simpson, K. Wong, S.D. Jackman, J.E. Schein, S.J. Jones, and I. Birol. ABySS: a parallel assembler for short read sequence data. *Genome Research*, Preprint, 2009.

[61] M.B. Soares, M.F. Bonaldo, P. Jelene, L. Su, L. Lawton, and A. Efstratiadis. *Proceeding of the National Academy of the Sciences*, 91:9228–9232, 1994.

[62] A. Sundquist, M. Ronaghi, H. Tang, P. Pevzner, and S. Batzoglou. Whole-genome sequencing and assembly with high-throughput, short read technologies. *PLoS ONE*, 2:e484, 2007.

[63] G.G. Sutton, O. White, M.D. Adams, and A.R. Kerlavage. TIGR assembler: A new tool for assembling large shotgun sequencing projects. *Genome Science and Technology*, 1:9–19, 1995.

[64] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in MPICH. *International Journal of High Performance Computing Applications*, pages 49–66, 2005.

[65] J.C. Venter, K. Remington, J.F. Heidelberg, A.L. Halpern, and D. Rusch. Environmental genome shotgun sequencing of the sargasso sea. *Science*, 304:66 – 74, 2004.

[66] J. Wang, W. Wang, R. Li, and Y. Li. The diploid genome sequence of an Asian individual. *Nature*, 456:60–65, 2008.

[67] R.L. Warren, G.G. Sutton, S.J.M. Jones, and R.A. Holt. Assembling millions of short DNA sequences using SSAKE. *Bioinformatics*, 23:500–501, 2007.

[68] D.L. Wheeler, T. Barrett, D.A. Benson, S.H. Bryant, K. Canese, V. Chetvernin, D.M. Church, M. Dicuccio, R. Edgar, S. Federhen, M. Feolo, L.Y. Geer, W. Helmberg, Y. Kapustin, O. Khovayko, D. Landsman, D.J. Lipman, T.L. Madden, D.R. Maglott, V. Miller, J. Ostell, K.D. Pruitt, G.D. Schuler, M. Shumway, E. Sequeira, S.T. Sherry, K. Sirotkin, A. Souvorov, G. Starchenko, R.L. Tatusov, T.A. Tatusova, L. Wagner, and E. Yaschenko. *Nucleic Acids Research*, 36:D13–D21, 2008.

[69] T. Wicker, A. Narechania, F. Sabot, G.T.H. Vu, A. Graner, D. Ware, and N. Stein. Low-pass shotgun sequencing of the barley genome facilitates rapid identification of genes, conserved non-coding sequences and novel repeats. *BMC Genomics*, 9:518, 2008.

[70] Business Wire. Helicos biosciences enters molecular diagnostics collaboration with renowned research center to sequence cancer-associated genes. *Genetic Engineering and Biotechnology News*, 2008.

[71] D. Zerbino and E. Birney. Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. *Genome Research*, 18:821–829, 2008.

ACKNOWLEDGEMENTS

I would like to use some space to acknowledge a number of people who have helped me, either directly or indirectly, with the presented work.

I would like to thank the Multidisciplinary Graduate Educational Training (MGET) Program, the National Science Foundation (NSF CNS-0521568), the Binational Agricultural Research and Development Program (Project US-3873-06), and the Plant Sciences Institute Innovative Research Grants Program for funding my graduate education.

I would like to thank Professor Srinivas Aluru for his guidance. His success is a testament to his wisdom, inteligence, and tenacity, and I can only hope that I have absorbed some of these traits during our six year association. I thank him for supporting my work and for his efforts in establishing the collaborations antecedent to this work's success. He was willing to work late hours as deadlines approached and has a keen editorial eye. I would also like to thank his wife, Maneesha, for this.

I would like to thank Patrick S. Schnable for contributing to the collaborative environment that served as the germ of my work. As a Biologist who also enjoys Mathematics and Statistics (and does not even mind Computer Science), he served as a great sounding board and source of ideas.

I would like to thank Henry B. Grant for giving me "Ludwig Wittgenstein: The Duty of Genius" by Ray Monk because "a person shouldn't receive a Doctorate of Philosophy without having read at least one philosopher."

I would like to thank Scott J. Emrich for making sure I had seen the latest and greatest sequence assembly paper.

I would like to thank Jaroslaw Zola for his feedback when designing my software and his

wife Olga for providing invaluable help in debugging templatized C++.

I would like to thank Xiao Yang and Chad Brewbaker for working on the maize validation project with me, briefly mentioned in this work. Chad's idea for displaying overlap graphs using a force directed layout algorithm was fun.

I would like to thank Pang Ko for being my climbing buddy as we failed to scale the optimal distributed suffix tree construction mountain (and for challenging my red pepper tolerance).

I would like to thank Chad Brewbaker, Jason Stanek, Scott Emrich, Anantharaman Kalyanaraman, Pang Ko, John Mathews, Olga Nikolova, Sarah Orley, Abhinav Sarje, Sudip Seal, Stephan Rajko, Andre Wehe, and Xiao Yang for listening to my presentations, giving me feedback on my work, and being good friends.

I would like to thank my parents Charles and Barbara Jackson for buying a computer when I was a child. I can still remember composing simple BASIC programs, replete with line numbers. I might have passed on Computer Science without this influence.

I would like to thank my brother Richard Jackson for listening to the psychosis of a Ph.D. student for the five years we lived together during my graduate studies. I would also like to thank the rest of my siblings, my extended family, and my friends for forgetting to ask me when I would graduate.

Finally, I would like to thank my wife Adrianna for enduring life as a thesis widow for the period of approximately one year. Thank you dear, with love.