COMPUTATIONAL METHODS FOR FAST AND ACCURATE DNA FRAGMENT ASSEMBLY

by

Carolyn F. Allex

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

1999

To Peter, Matthew, Rachel, and Paul

for their love, laughter, and support

Abstract

As advances in technology result in the production of increasing amounts of DNA sequencing data in decreasing amounts of time, it is imperative that computational methods are developed that allow data analysis to keep pace. In this dissertation, I present methods that improve the speed and accuracy of DNA fragment assembly.

One critical characteristic of automatic methods for fragment assembly is that they must be accurate. Currently, to ensure accurate sequences, the data that underlies questionable base calls must be examined by human editors so that the correct base call can be determined. This manual process is both error-prone and time-consuming. Automatic methods that yield high accuracy and few questionable calls can reduce errors and lessen the need for manual inspections. In my work, I developed a method, *Trace-Evidence*, that automatically produces highly accurate consensus sequences, even with few aligned sequences.

Most assembly programs analyze only base calls when determining a consensus sequence. The key to the high accuracy is that I incorporate morphological information about the underlying ABI trace data. This is accomplished through a new representation of traces, *Trace-Class*, that characterizes the height and shape of traces. The new representation not only yields high accuracy when used in consensus-calling methods, but also produces improved results when used in removing poor-quality data, and when used as inputs for neural networks for consensus determination.

The need for fast processing is becoming more important as the size of sequencing projects increases. Almost all existing fragment assembly programs perform pairwise comparisons of

reads, resulting in execution times proportional to n^2, where n is the number of reads. I describe a new algorithm for fragment layout, *SLIC*, that runs in time proportional to n. *SLIC* relies on subsequences of bases that occur in overlapping regions of fragment reads. Subsequences that are common to two or more fragment reads are aligned to determine the overall layout of reads.

The work I present provides improvements to currently available computational methods for DNA sequencing that can serve as a foundation for further study in developing better solutions to problems in fragment assembly.

Acknowledgements

I wish to acknowledge the funding and resources who made my research possible. I received funding from National Research Service Award 5 T32 GM08349 from the National Institute of General Medical Sciences (NIGMS), National Institutes of Health (NIH) Small Business Innovative Research (SBIR) phase I grant #R43 HG01893-01, and DNASTAR Inc. Sequence data used in all testing is from the *E. coli* Genome Project under the direction of Fred Blattner at the University of Wisconsin–Madison. Source code for *Phred*, *Phrap*, and *CrossMatch* was supplied by Phil Green at the University of Washington.

In addition, there are myriad people without whom I could not have accomplished this work. Key in the process were my advisor, Jude Shavlik in Computer Sciences, and my minor advisor, Fred Blattner in the Genetics Department. I especially thank Jude for setting high standards; I have become a much better researcher under his tutelage. Others who contributed are my colleagues at DNASTAR Inc.: Schuyler Baldwin, John Schroeder, Chris Stern, Hang Ying Dong, Jeff Briganti, and Tim Burland. As my lunch buddies and fellow software engineers, Schuyler, John, and Chris are not only remarkably helpful, but are also great friends and allies. I also thank those who went before me in my research group who gave me sound advice and instruction: Mark Craven, Rich Maclin, Dave Opitz, and honorary group member, Nick Street. I am also grateful to CS graduate coordinator, Lorene Webber, with whom I shared many tales over the past seven years.

Others have not contributed directly to my work, but their support was essential nonetheless. In particular, my children, Peter, Matthew, Rachel, and Paul, and their dad, Mike

Allex, have been patient and encouraging as I have worked toward this goal. Now almost 14 years old, Rachel and Paul cannot remember a time that I was not engrossed in school work. I am grateful for the great job Peter has done when required to take care of things at home as the "responsible person" left in charge. Matthew has been very gracious every time I have had to declare, "OK, you guys have to find yourselves something to eat until after I finish my project (exam, paper, presentation, qualifiers, prelim, dissertation, final defense...)."

While on the subject of terrific family members, I must say how much I appreciate my parents, Cornelius and Marion Boerboom. Mom and Dad have been forever generous with their encouragement and love. They have always made me believe that I am capable of attaining any goal that I set out to achieve.

I have been fortunate to have wonderful friends who provided me with distractions and attentive ears during my school years. My best friend, Carrie Pritchard, has actually seen me through three degrees now, a BS in 1992, MS in 1994, and now finally my PhD. I could not have made it through all that without her as my soul mate. My first day of graduate school orientation I was lucky to choose to sit next to Ernie Colantonio, a man who would become one of my closest friends. He has sustained me with his humor, music, and pizzas. Starting an internship at DNASTAR in 1995 also proved to be a great decision for me as I met another wonderful friend, Schuyler Baldwin. Schuyler is not only a special friend, but also a co-author in my research and the holder of all Macintosh computer knowledge. Finally, I made a fateful choice when I decided to take up bicycling two years ago. Among the enjoyable group of friends I have met on Wednesday night bike rides, one in particular, Brian Cassel, has become an important person in my life. Although he has not been around to see me through the whole process, he has cheered me on during the last year and a half when things have been the most hectic.

Last, but not least, I thank the faculty members who have taken the time to serve on my preliminary and final committees, providing me with much helpful advice. In addition to Jude Shavlik and Fred Blattner, they are: Anne Condon, Lloyd Smith, Chuck Dyer, and Olvi Manasarian.

Contents

Chapter 1

Introduction

Are you a man or a mouse?

Actually, if you look deep down inside, at genomes residing in the depths of cells, you will find that the answer is not as obvious as you might guess. Human genomes, mouse genomes, and, for that matter, the genomes of all organisms share a surprising similarity. As a matter of fact, human genes for cell cycle and growth can be swapped without harm with those of an extremely distant cousin, yeast (Green & Waterston 1991).

Genomes carry the totality of the genetic material for an organism. They are one or more molecules of *deoxyribonucleic acid*, commonly known as *DNA*. Sequences composed of four types of deoxynucleotide bases form DNA molecules. The four bases are: adenine (A), cytosine (C), guanine (G), and thymine (T). Encoded in the base sequences are *genes* – the blueprints for proteins that are responsible for the functions that enable us to grow and exist.

1.1 The Human Genome Project

Finding all the genes and discovering the functions of their proteins represent the Holy Grail of knowledge for human life and health. Over 10 years ago, the United States, along with several other countries, embarked on the search for this Holy Grail, christening it the *Human Genome Project* (HGP). The task is ambitious; there are over three billion base pairs in the human genome that embed approximately 100,000 genes. For fiscal year 1990, near the beginning of the endeavor, the National Institutes of Health (NIH) and Department of Energy (DOE) budgeted almost $90 million for the project, estimating that to reach their goal would eventually require 15 years and about $3 billion (Goodman 1990).

From the start of the project, funding agencies recognized the need for computational approaches to problems. Initially, twenty percent of monies budgeted for the HGP were earmarked for research into bioinformatics and a *Joint Informatics Task Force* was formed (Frenkel 1991). The purpose of the task force was to identify user needs, set goals, establish research and development priorities, and to enhance the effectiveness of computational solutions to genome informatics problems.

Clearly, computational solutions to problems in genomics are crucial to the success of the HGP. Back in the early 1970s, it was a laborious task to determine the sequence of even a 25 base-pair sequence with confidence. After 1977, with the introduction of gel electrophoresis-based sequencing technology (Maxam & Gilbert 1977, Sanger, Nicklen & Coulson 1977), finding the sequence of several hundred base pairs became routine. Fast-forward to the present and you will find that we now have the complete sequences of a number of organisms – *C. elegans* (Wilson 1999), *E. coli* (Blattner *et al.* 1997), and *Haemophilus infuenzae* (Sutton *et al.* 1995), among them. In 1998, Dr. J. Craig Venter, Perkin-Elmer Corporation, and the Institute for Genomic Research (TIGR), proposed to launch a joint venture that would sequence the entire human genome in three years (Marshall 1999). Inspired by this declaration, the National Human Genome Research Institute moved up its target for completing the human genome sequence by two years, to 2003 (Wade 1998).

Given the size and quantity of sequencing efforts and goals, the amount of data that must be stored and analyzed is tremendous. Sequence data is stored in public and private databases throughout the world. A public repository for annotated DNA sequence data is GenBank, administered by the NIH (Benson *et al.* 1998). As of April 1999, GenBank alone contained sequence data on over 2.5 billion bases (NCBI 1999); the thought of analyzing this much data by hand is inconceivable.

Computational genomic data analysis tools have matured dramatically since their debut. In the early years, methods to align DNA sequences were developed that became the necessary cores of automatic sequence analysis. The goal of DNA sequence alignment is to align the bases in two or more sequences such that the number of mismatches is minimized. *Needleman-Wunsch* was one of the original methods that made its appearance in 1970 (Needleman & Wunsch 1970). This technique uses dynamic programming to find an optimal alignment for a pair of sequences. Since *Needleman-Wunsch* was introduced, a number of refined and new alignment methods have emerged (Vogt, Etzold, & Argos 1995, Huang 1994, Brutlag *et al.* 1993, Streletc *et al.* 1992, Berger & Munson 1991, Subbiah & Harrison 1989, Johnson & Doolittle 1986, Boswell & McLachlan 1984, Fickett 1984, Smith & Waterman 1980).

Of course, aligning sequences is only one problem in genomics. Myriad other problems must also be addressed. Among them are: base calling (e.g. Tibbetts, Bowling & Golden 1994), finding protein coding regions (e.g. Uberbacher & Mural 1991), differentiating exons from introns (e.g. Chen & Zhang 1998), predicting protein structure (e.g. Rost & Sander 1993), identifying motifs (e.g. Sun *et al.* 1996), finding sequence homologies (e.g. Karplus, Barrett & Hughey 1998), searching databases (e.g. Lavorgna *et al.* 1999), detecting protein binding sites (e.g. Heumann, Lapedes & Stormo 1994), and finding RNA polymerase binding sites (e.g. Pedersen & Engelbrecht 1995).

1.2 DNA Sequencing

Specific to my work, DNA sequencing presents numerous computational challenges since segments of DNA longer than about a thousand bases cannot be sequenced directly. First, smaller overlapping fragments of the DNA are sequenced. The overlapping regions of the sequences are aligned and their *consensus* is the sequence of the original fragment. This process is called *fragment assembly* and is described in greater detail in Chapters 2 and 8. Major steps in fragment assembly include: *base calling* (interpreting output from sequencing machines to call the sequence of bases), *sequence layout* (aligning overlapping base sequences), and *consensus calling* (determining the consensus of the aligned sequences). Figure 1-1 illustrates an overview of the process of DNA sequencing.

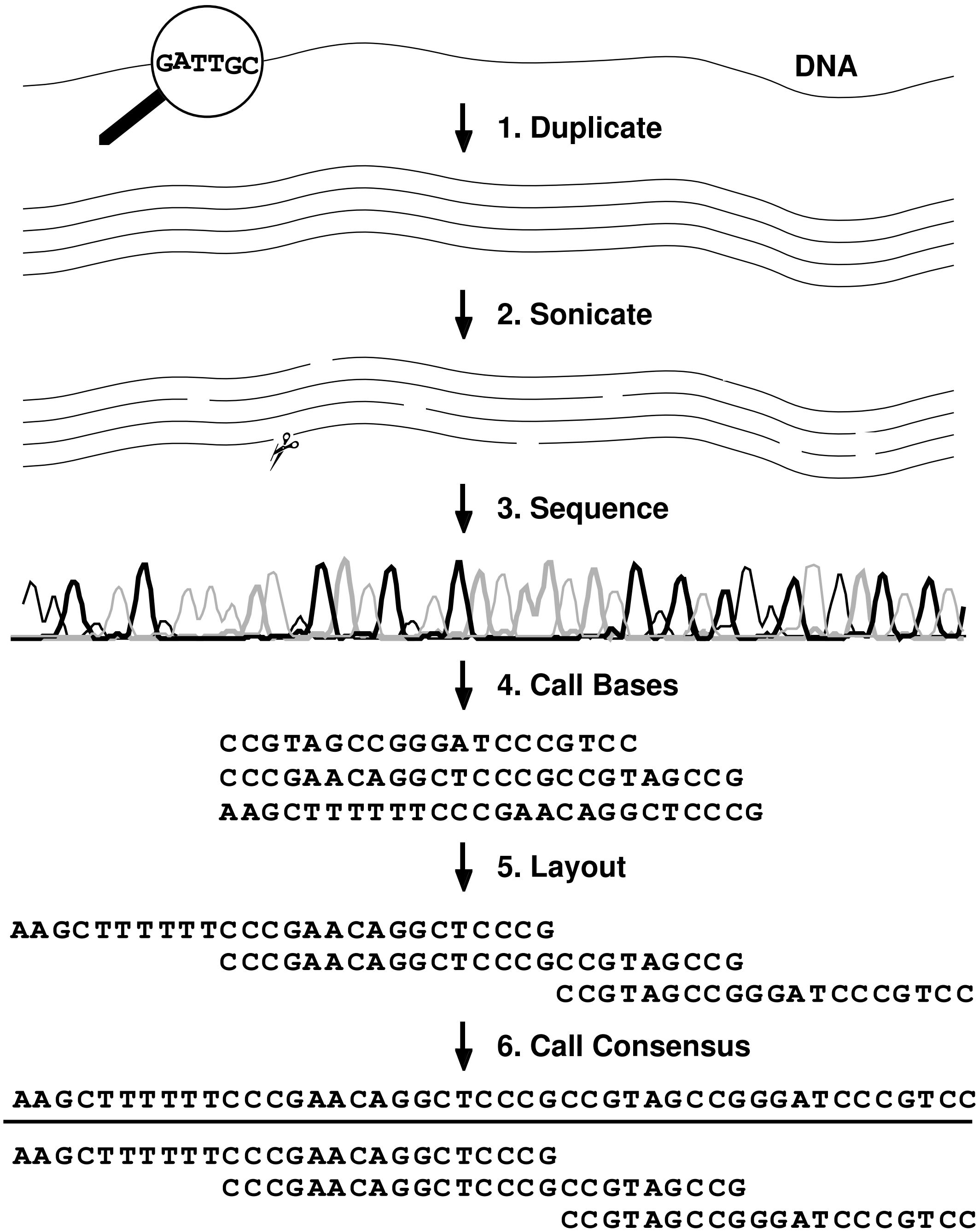


Figure 1-1. DNA Sequencing. The first three steps in DNA sequencing occur in the laboratory where large fragments of DNA are duplicated and then broken into smaller fragments that are sequenced individually. Computational methods are used in steps 4 to 6. First the sequence of bases in the individual fragments is determined. The base-call sequence of the individual fragments are overlapped and aligned and their consensus is the sequence of the original fragment of DNA.

A number of characteristics inherent in DNA and limitations in the chemistry of sequencing reactions can hinder the completion of these steps. One problem is noise in the data; the data output from a sequencing machine is not perfect, leading to miscalled bases, insertions, and deletions in the base-call sequence. In addition, the fragments of DNA may still have *vector* sequence (a fragment of DNA used to carry and replicate the fragment of interest) at their ends or even whole contaminant fragments not from the target of interest may be mixed in. One significant problem that has yet to be effectively addressed is the occurrence of repeated regions in genomes. All of these lead to difficulties in aligning the overlapping portions of sequences and in determining the consensus sequence (Chapter 8).

An additional layer of complexity is added to problems in DNA sequencing as technology produces not only much higher throughput from machines, but also allows the sequencing of much larger fragments of DNA. As it becomes possible, the goal for many researchers expands to sequencing larger fragments, and even whole genomes of DNA (Weber & Myers 1997). The amount of data that must be processed may be getting too large for current software to handle in reasonable amounts of time.

1.3 New Methods

In my work, I develop software for DNA sequencing directed at making DNA fragment assembly *fast* and *accurate*. The need for fast processing is becoming more important as the size of sequencing projects increases. Almost all existing programs perform pairwise comparisons of fragment reads, resulting in execution times proportional to n^2, where n is the number of reads. With an n^2 method, the assembly time may take years for a large project. In this dissertation, I describe a new algorithm I developed for sequence layout, *SLIC* (*Sequence Layout into Contigs*), that, in practice, runs in linear time with respect the the number of reads.

The *SLIC* layout algorithm relies on subsequences of bases, or *mers*, that occur in overlapping regions of fragment reads. Mers that are common to two or more fragment reads are aligned to determine the overall layout of reads. The premise is that large DNA fragments contain many mers that occur only once (or infrequently) and that can be used to tag a relative positions of fragment reads (Jain and Myers 1997).

Another critical characteristic of automatic methods for fragment assembly is that they must be accurate. Inaccurate sequences can lead to serious problems; the change, insertion, or deletion of even a single base can result in a translated protein of dramatically different nature than the true protein. Often, questionable sequence bases can be identified by computer software. Human sequence-editors then visually inspect the underlying data for each of these questionable calls and determine the correct base call. This manual editing is a real bottleneck in the sequencing process. An even worse case is when a call that is incorrect is not identified as suspicious. The sequence with the incorrect call is analyzed or deposited in a database while containing the error. Automatic methods that yield high accuracy can greatly diminish the number of errors and lessen the need for expensive manual inspections. The cost of sequencing can also be reduced by the use of a consensus-calling method that is highly accurate with fewer sequences. In my work, I developed a method, *Trace-Evidence* (Chapter 6), refined as *Trace-EvidenceII* (Chapter 8), that automatically produces highly accurate consensus sequences, even with few aligned sequences.

Most assembly programs analyze only the sequence of bases when determining the consensus sequence for an alignment of fragment reads. The key to the high accuracy I realize with the *Trace-Evidence* method is that I look beyond the base calls to the underlying data of the sequence. As shown in Figure 1-2, the underlying data is in the form of a set of four sequences of fluorescent-dye intensities, known as *traces*. The traces are output from sequencing machines (such as the Perkin-Elmer Applied Bioysystems Inc. (ABI) 3700) and are used for determining the base call sequences. As input to the *Trace-Evidence* method, I have developed and refined a new representation of the traces, *Trace-Class*, that I use to improve the accuracy of consensus sequences (Chapter 3).

The *Trace-Class* representation and its various refinements are also useful in other fragment-assembly tasks. For example, I obtain better quality assemblies when I use the *Trace-Class* representation in trimming poor quality data from the ends of sequences before assembly (Chapter 4). This helps to eliminate some problems caused by noisy data containing incorrect base calls. Also, when used as inputs for neural networks for consensus determination, the *Trace-Class* representation produces more accurate sequences than networks that use only base calls as inputs (Chapter 7). In every problem for which I incorporated trace information via the *Trace-Class* representation or one of its variants, I find improvement in the results I obtain.

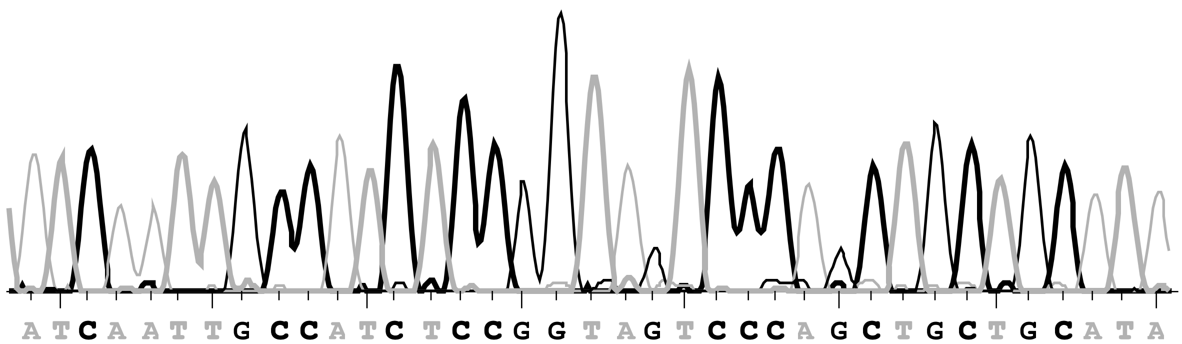


Figure 1-2. Fluorescent Trace Data. Sequences of fluorescent-dye intensities are used to call the sequence of bases for a fragment of DNA. (Actual data shown.)

1.4 Thesis Statement

In this dissertation, I describe novel computational approaches to problems in DNA fragment assembly. The hypothesis I put forward is that the accuracy and speed of DNA fragment assembly may be increased by computational methods that incorporate fluorescent trace information, and by the use of a fragment read layout algorithm that identifies and aligns probable unique subsequences that are common to two or more fragment reads. In practice, the layout algorithm executes in linear time with respect to the number of fragment reads.

1.5 Dissertation Organization

After background information on DNA sequencing is given in Chapter 2, the core of this dissertation is organized into three broad sections: 1) definition of the new fluorescent trace representation, 2) case studies utilizing the trace representation, and 3) methods for fragment assembly. In Chapter 3 I define *Trace-Class*, the new representation of trace data. Next, a case study applying it to trimming of low-quality sequence ends is covered in Chapter 4, and case studies in consensus calling are in Chapters 5 through 7. In Chapter 5, the consensus calling problem is defined, Chapter 6 describes an algorithmic technique, and Chapter 7 details a neural network approach. Finally, the last section, spread among Chapters 8 to 11, describes methods for fragment assembly. Chapter 8 defines fragment assembly, existing methods, and describes my new linear-time layout algorithm, *SLIC*, and its ancillary methods for fragment assembly. In Chapter 9, I evaluate the effectiveness of *SLIC* and its companion methods by comparing it to *Phrap* from the University of Washington and DNASTAR Inc.'s *SeqManII*. Chapter 10 analyzes the computational complexity of *SLIC*. I then report additional related research in Chapter 11, and outline conclusions and future work in Chapter 12. The chapters are followed by four appendices: *A* is a glossary of biological terms, *B* contains pseudocode for trimming algorithms, *C* lists pseudocode for assigning Trace-Class scores, and *D* is a detailed definition of the *SLIC* algorithm. Finally, a list of references completes the dissertation.

Chapter 2

DNA-Sequencing Background

The focus of my research is computational methods for DNA sequencing – determining the sequence of bases (*A*, *C*, *G*, and *T*) in DNA molecules. State-of-the-art sequencing systems, such as the Perkin-Elmer Applied Biosystems Inc. (ABI) 3700, use fluorescent-dye labeling of DNA fragments in their processes (Ansorge *et al.* 1986, Smith *et al.* 1986). Fluorescent-dye sequencing technology will be described in this chapter.

The goal of a sequencing effort may be as modest as determining the sequence of a small fragment of DNA less than a kilobase (kb) long or as ambitious as sequencing an entire genome. The sizes of genomes vary widely; a small genome is about a million bases long, the sequence of a typical bacterial genome is millions of bases long, and there are over 3 billion bases in the human genome. For clarity in this chapter, I will assume that the goal is to sequence an entire genome with the understanding that *genome* may refer to any large DNA segment of interest. In brief, the sequencing procedure consists of producing overlapping short fragments of the genome, sequencing each fragment, and finally aligning the overlapping areas of the fragments to determine the overall sequence of the genome. The procedure of breaking the genome and sequencing the smaller fragments is necessary because modern technology only allows the sequencing fragments that are usually less than one kb long and the segments of interest are generally much longer.

2.1 Fragment Sequencing

Genomes consist of two strands of DNA that intertwine to form a double helix. The two strands are held together by bonds between pairs of bases. Each of the four bases forms a pair with a specific *complementary* base: A pairs with T and C pairs with G. In a double helix, one strand of DNA is the complement of the other and when DNA is replicated, the strands are used as a templates for synthesizing complementary strands. Figure 2-1 illustrates a double strand of DNA and its replication.

To sequence an individual fragment, first a set of sub-fragments needs to be produced. Building upon *primer* fragments (short fragments of DNA used to prime replication), the set is generated through DNA replication. At each replication step, deoxynucleotides (A, G, C, and T) and dideoxynucleotides (A^*, G^*, C^*, and T^*) compete for addition to a growing sequence. Deoxynucleotides permit elongation whereas dideoxynucleotides terminate replication (Prober *et al.* 1987). The result is a set of sub-fragments that encompasses all possible lengths (except those of the initial primer).

Each dideoxynucleotide that terminates a sub-fragment is labeled with a fluorescent dye. Since a different dye labels each of the the four bases, all sub-fragments of a given length are labeled with the same dye. (A dye-primer labeling method also exists, but will not be described here (Ansorge *et al.* 1986).) Figure 2-2 shows an example of a fragment sequence and its corresponding set of sub-fragments.

The set of labeled sub-fragments is placed on a plate of polyacrylamide gel and a voltage is applied. The current induces the migration of sub-fragments through the gel. Since smaller pieces of DNA migrate more quickly than larger ones, the sub-fragments become separated by size. The fluorescent labeling then provides the means for determining the fragment sequence.

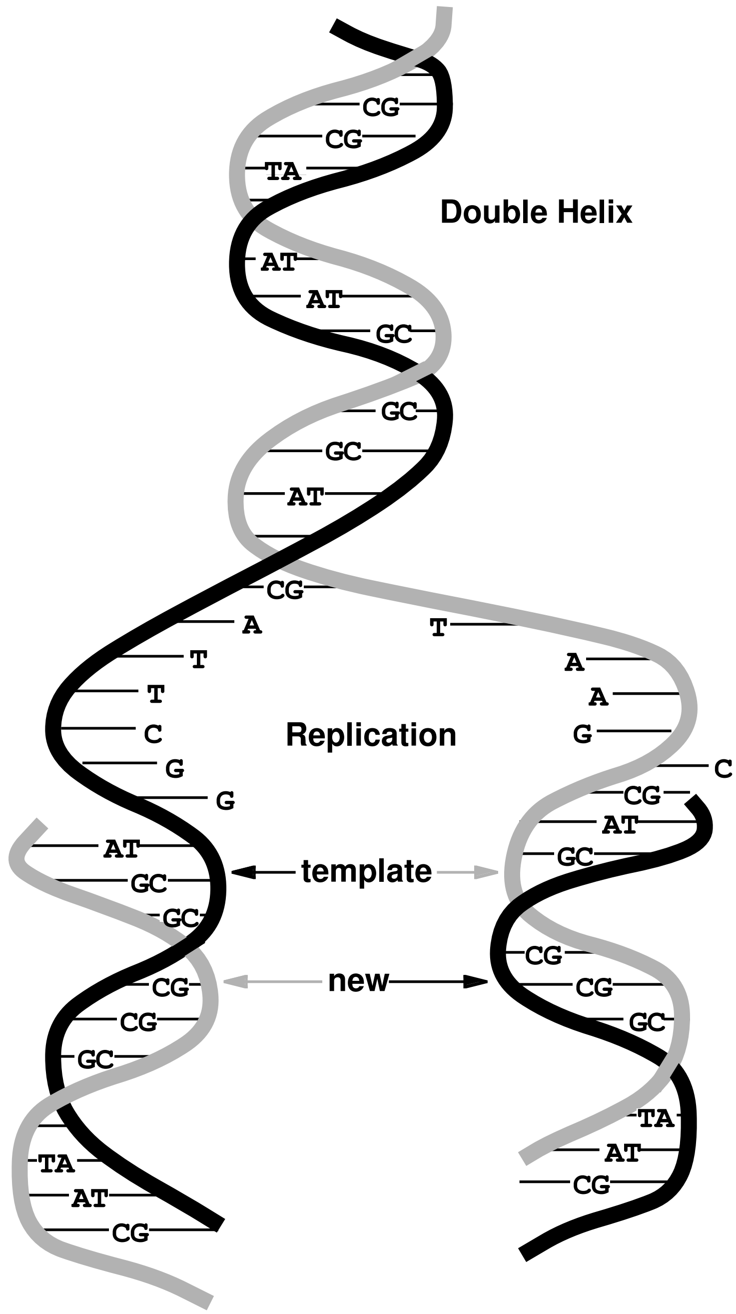


Figure 2-1. DNA Replication. DNA forms a double helix where each base pairs with its complement: *T* bonds with *A* and *C* bonds with *G*. Each strand serves as a template during replication.

Fragment: **CTTGCTACCCTTCGGA**
+ Primer: **GAACG**
+ Deoxynucleotides: **A, G, C,** and **T**
+ Dideoxynucleotides: **A*, G*, C*,** and **T***

Yields →

Complementary sub-fragments:

GAACGA*
GAACGAT*
GAACGATG*
GAACGATGG*
GAACGATGGG*
GAACGATGGGA*
GAACGATGGGAA*
GAACGATGGGAAG*
GAACGATGGGAAGC*
GAACGATGGGAAGCC*
GAACGATGGGAAGCCT*

Figure 2-2. DNA Sub-Fragments. Quantities of deoxynucleotides, and dye-labeled dideoxynucleotide terminators are added to copies of a fragment to produce a set of sub-fragments. The asterisks designate fluorescently labeled dideoxynucleotide terminators.

A detection device in the sequencing machine reads the intensity *trace* of each of the four fluorescent dyes as the sub-fragments migrate past. This process is called *reading the trace*, and the data produced are called *traces*. There is one set of trace data for each of the four fluorescent dyes. Although each trace is composed of discrete measurements, the points can be interpolated to form a continuous curve. A simplified diagram of an automated DNA sequencer is in Figure 2-3.

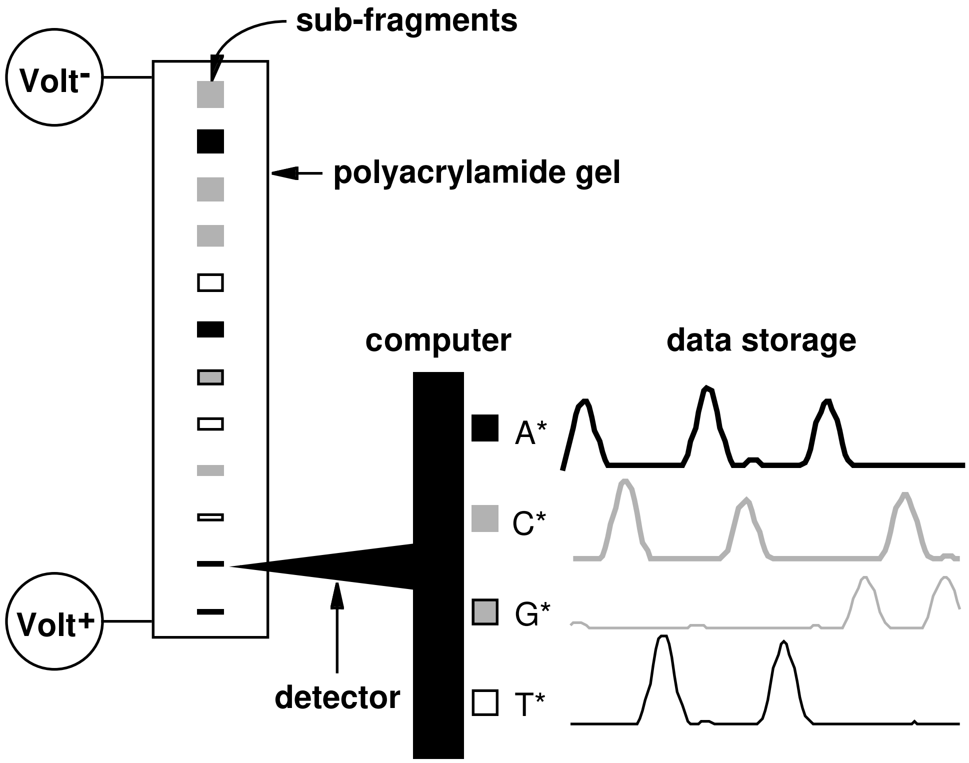


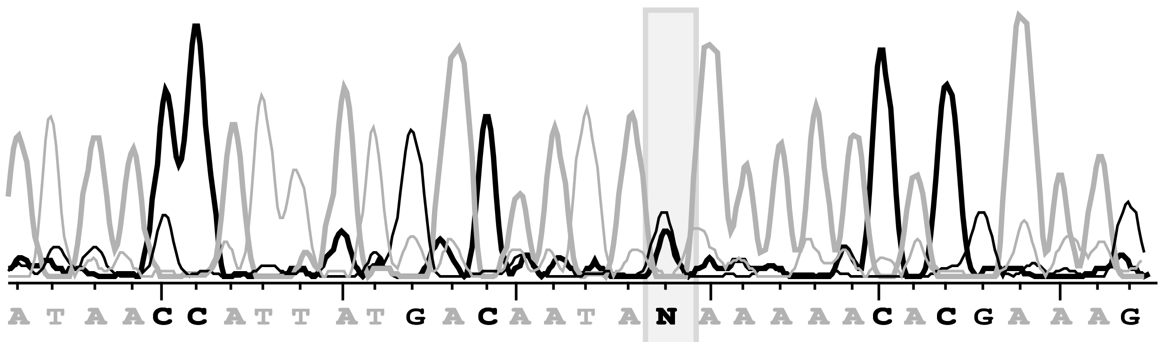
Figure 2-3. DNA Sequencer. DNA sub-fragments are placed on a plate of polyacrylamide gel and an electric current is applied. Fragments in order from smallest to largest migrate past a detection device. The detector reads the fluorescent intensity of each of the four dyes. A computer records the sequence of intensities that are subsequently used for determining the base call sequence of the fragment.

2.2 Base Calling

The traces are used by base-calling software to determine the sequence of bases in the fragment; this is referred to as *base calling*. The four sets of traces are kept synchronized as they are scanned during base calling. The base caller expects to call a base at fairly regular intervals and calls one base for each of these intervals in a trace (Perkin-Elmer 1995). There are usually about 10 to 15 trace data points per interval, and a record is kept of the points at which the calls are made.

A sequence of base calls and corresponding trace graphs and sequence of intensities are depicted in Figure 2-4. The sequencer calls the bases in order from the beginning (called the *5′ end*) to the end (*3′ end*) of the sequence. Calls are made by examining the values of the traces. Ideally, the trace values for only one base form a distinct peak. In this case, the base corresponding to that trace is the one that is called. Sometimes the traces for two or more bases form similar peaks. In this case, the sequencer makes a *no-call* and labels the base with an *N*. The goal is to obtain the exact sequence of bases that is the complement of the fragment. In practice, the accuracy of the base calls made by sequencers is 98-99% for the first several hundred bases (Chen 1994, Kelley 1994). (In personal observations, I see that in many sequencing runs, the first 0 to 50 base calls also have lower accuracy.)

Trace Graphs



Base Calls

Trace intensities

A: 344 89 50 199 420 633 389 167 40 23 11 2 0 0 0 0 0 0 0 0 …
C: 0 12 34 39 42 45 33 12 180 404 654 920 789 670 556 887…
G: 87 50 35 23 12 3 25 49 106 208 324 207 135 88 47 18 6 10…
T: 12 4 33 77 10 6 2 0 0 4 23 26 22 5 5 3 1 0 0 0 0 0 0 0 0 0 0…

Figure 2-4. Trace Graphs, Base Calls, and Intensities. Automatic base callers scan sequences of intensities, calling the base that is associated with the trace with the highest peak intensity. The shaded base is a *no-call* labeled with an *N* since two peaks are similar and the correct call is not obvious. (Actual trace graphs and base calls shown.)

2.3 Fragment Assembly

Once all the fragments of the original genome have been sequenced, the fragments are *assembled* into larger segments (McCombie & Martin-Gallardo 1994, Myers 1994, Rowen & Koop 1994). The fragments overlap, so an assembly is produced by aligning the overlapping regions of the sequences. The goal of DNA sequence alignment is to align the bases in two or more sequences such that the number of mismatched bases in a column is minimized. Both the commonly used *Needleman-Wunsch* (Needleman & Wunsch 1970) and *Smith-Waterman* (Smith & Waterman 1980) alignment methods use dynamic programming to find an alignment between two sequences. In an alignment, the bases aligned in a column are used to determine the *consensus* call of the column. Ordered by columns, the consensus base calls for the columns form the overall consensus sequence for an alignment (see Figure 2-5). To make a multiple alignment, a greedy approach may be used in which sequence reads are added one at a time to a growing alignment. To add a read, a pairwise alignment is formed between the read and the consensus sequence of the alignment. When all sequences have been overlapped and aligned, the alignment forms a contiguous sequence of DNA that is known as a *contig* (Staden 1980). The base call sequence of a contig is its consensus sequence.

```
Consensus: ACGAGCGGGCAGACAGCATTCGACACGCCCATGTACGCCAATGGGT
           ----------------------------------------------
           ACGAGCGGGCAGACAGCATTCGACACGCC
                     AGACAGCATTCGACACGCCCATGTAC
                                 ACACGCCCATGTACGCCAATGGGT
```

Figure 2-5. Column Consensus Calls. Three sequences in a multiple alignment are listed horizontally. The consensus sequence is computed by making a consensus call for each column in the alignment.

When sequences are assembled that contain errors, there are base locations where sequences align but do not agree completely (McCombie & Martin-Gallardo 1994). A consensus base call in these cases may be assigned one of 12 *ambiguity codes* as listed in Table 2-1. (An *ambiguity* is any call that is not *A*, *G*, *C*, nor *T*.) In some cases, it is necessary to insert a *gap* (indicated by a hyphen) in a sequence to optimize the alignment. A gap indicates that either a base is missing from the sequence of base calls (a *deletion*) or that a false base has been called in one or more of the aligned sequences (an *insertion*). Figure 2-6 portrays a multiple sequence alignment of overlapping fragment reads containing some gaps and ambiguities.

Table 2-1. Base Ambiguity Codes.

Base	Code	Base	Code
A or ***G***	***R***	***C*** or ***T***	***V***
A or ***T***	***W***	not ***A***	***B***
A or ***C***	***M***	not ***C***	***D***
G or ***T***	***K***	not ***G***	***H***
G or ***C***	***S***	not ***T***	***V***

In an ideal assembly where the data is flawless and available, the sequences align to form one contig and each consensus base call is *A*, *G*, *C*, or *T*. In fact, this is rarely the case. Difficulties inherent in the preparation and sequencing of fragments lead to incorrect base calls. Also, the quality of the traces becomes progressively worse near the end of the fragment. Many more incorrect calls and no-calls are in this region (Kelley 1994, Perkin-Elmer 1995).

```
Consensus: CACATACTTACGGCGRGGACAGCATTCGACAGBCCATGACGGATTTT
           -----------------------------------------------
           CACATACTTACGCCCGGGACAGCATTCGAC-GGCCATGACGGATTTT
                     CGGCGAGG-CAGCATTCGACAGTCC-TGACGGATTT
               TACTTACGGCGAGGACAGCCTTCGACACCCCATG-CGGAT
             CATAC-TACGGGGGGGACAGCAT-CGACAGCCCATGA
```

Figure 2-6. DNA Sequence Alignment. Four overlapping fragments are aligned and gapped to determine the sequence of a segment of DNA. Ambiguous calls occur in the consensus for columns (in gray) that are not in total agreement. The base sequence of this contig is the consensus of the aligned fragments.

A particularly difficult problem is introduced into the assembly process by subsequences that occur more than once in a genome. These subsequences are called *repeats* or *repetitive* *elements*. When fragment sequences contain repeated elements, their placement in a contig is not clearly defined. Figure 2-7 illustrates this problem. Some repeats are exact duplicates of others, while others contains some variance. Repeats vary both in length and in number of tandem occurrences. Those that are less than a fragment read in length are fairly straight forward to handle, since flanking sequences can be used to position overlaps. Longer repeats make finding correct overlaps far more difficult. Although most established assembly programs address repeats with varying amounts of sophistication and success, the problem is far from solved.

```
Genome
  TCTTGGTCATGTCATACGTCGTCATGTCATTGGTCCC

Sequence Reads
            TCTTGGTCATGTCAT
            GTCATGTCATACGTC
            ACGTCGTCATGTCAT
            GTCATGTCATTGGTCCC

Layout
  TCTTGGTCATGTCAT
       GTCATGTCATACGTC
                 ACGTCGTCATGTCAT
                      GTCATGTCATTGGTCCC

                     or

          TCTTGGTCATGTCAT
               GTCATGTCATTGGTCCC

           ACGTCGTCATGTCAT
                GTCATGTCATACGTC
```

Figure 2-7. Repeated Subsequences. A fictitious genome contains a subsequence that is repeated (shown in gray). Two layouts are possible when overlapping the repeated regions in the sequence reads. One layout results in a single contig (the correct layout), and the other in two contigs.

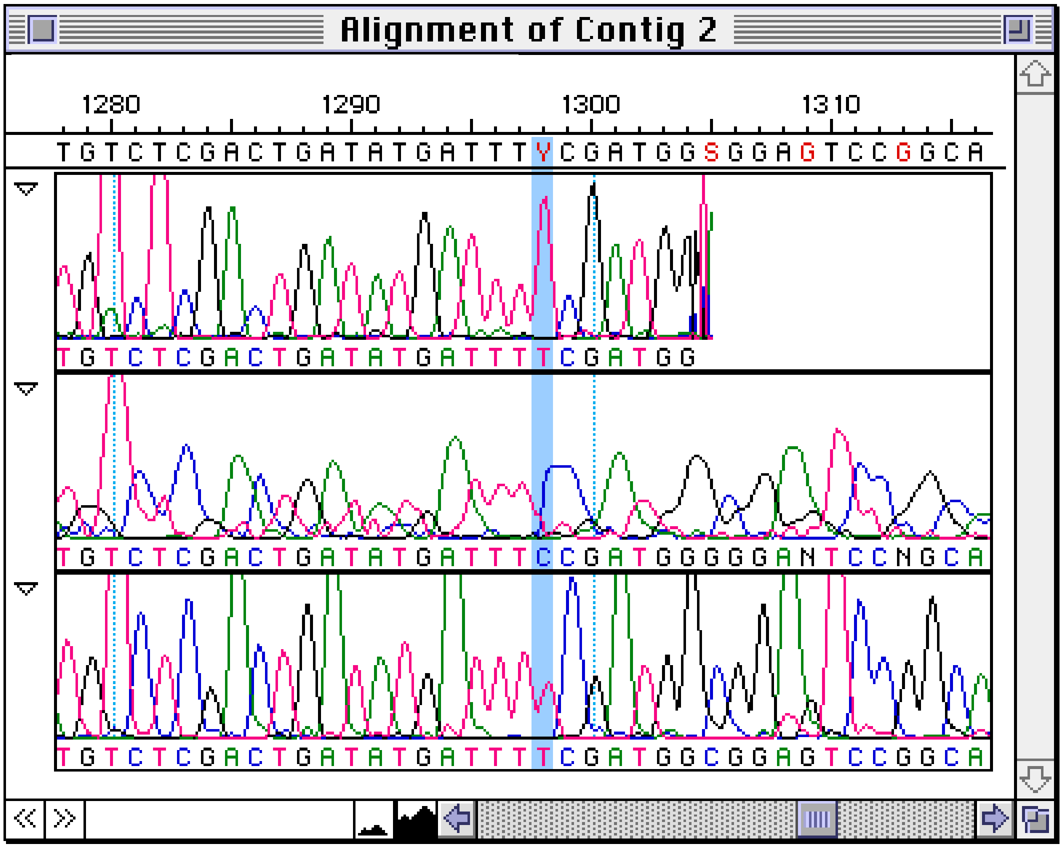
2.4 Manual Editing

In general, after assembly human editors resolve ambiguous calls to one of the four bases before analysis or submission to GenBank. Ambiguous calls serve to focus editors' attention on areas in the consensus that warrant closer examination. Manual editing is a time-consuming task performed by human sequence-editors that entails visual analysis of the assembly and data (Rowen & Koop 1994).

As an example of manual editing decisions, in Figure 2-8, three sequences have been aligned and the consensus computed. In the shaded column the first and third sequence have been called as a *T* and the second as a *C*, resulting in an ambiguous consensus call of *Y* (*C* or *T*). An editor examines the traces associated with the sequences and observes not only that overall the trace is quite good for the first and third sequences, but also that the *T* peaks in the column are sharp and well-defined. In contrast, the trace for the second sequence is not as good and furthermore, the *C* peak is not well-defined. Given this evidence, an editor is likely to assign a *T* to the consensus call in this column. As the size of sequencing projects continually grows, it becomes increasingly important to reduce the need for these kinds of costly manual operations (McCombie & Martin-Gallardo 1994, Rowen & Koop 1994).

2.5 Summary

Modern sequencing machines can only determine the sequence of DNA fragments of at most one kb. The machine produces a *trace* of the dye intensities for each of the four bases while scanning fluorescent-dye labeled sub-fragments as they migrate past a detection device. Base-calling software scans the four traces in unison to detect high-intensity peaks. Processed in order of migration time, the bases associated with the peaks form the sequence of bases in a fragment. This process is known as *base calling*. The fragment reads are joined into larger contiguous segments of DNA (*contigs*) by aligning overlapping regions of the reads. The sequence of a contig is the *consensus* of its aligned reads. Some difficulties in aligning and determining the consensus sequence are introduced by errors in base calling. Unless reliable automatic methods are available, these and other difficulties discussed in later chapters must be corrected by human editors. Since manual editing is time-consuming and error-prone, a worthwhile task is to develop useful automatic methods.



Screen shot from DNASTAR Inc.'s *SeqManII*

Figure 2-8. Manual Editing. Three sequences have been aligned. The shaded column contains conflicting base calls resulting in an ambiguous consensus. Human editors examine the traces and resolve the ambiguity.

Chapter 3

DNA Fluorescent-Trace Representation

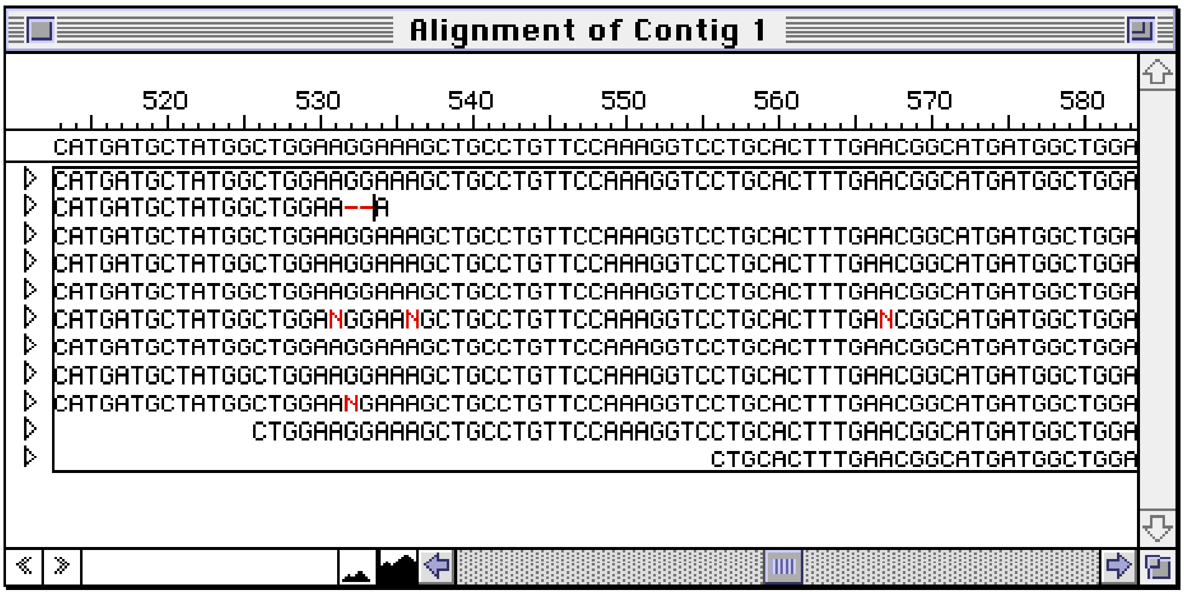
One of my important goals is to reduce the expense and increase the accuracy of sequencing by improving the quality of automatic assemblies. I believe that this may be accomplished by the direct incorporation of fluorescent-dye trace information into automatic processes. This solution requires an appropriate representation of the traces. Since existing representations were inadequate for this purpose, the focus of this portion of my research is to develop a representation of trace data that is descriptive, yet easy to incorporate into automatic sequencing tasks.

3.1 Existing Representations

Three fundamental tasks in DNA sequencing utilize traces; each uses a different representation of the traces. The tasks are: base calling, fragment assembly, and manual editing. A brief overview of each task follows.

A detailed representation of trace data is as a sequence of fluorescent-dye intensities. The sequence for each base call lists about 10 to 15 intensity values. The sequence of intensities is the representation used in base calling. Base calling is a straightforward task with traces that contain peaks that are well-defined and scaled high. The base caller can simply call the bases (A, C, G, or T) that are associated with the highest peak intensities. In cases where a call must be made, but two or more peaks are similar, the base caller makes a *no-call* and labels the base with an *N*.

A second representation of trace data is a sequences of base calls. This forms a much simplified representation that is used in automatic fragment assembly programs. Virtually all assembly programs align sequences of bases so that their consensus can be computed. Figure 3-1 shows an example of sequences aligned in an assembly.



Screen shot from DNASTAR Inc.'s *SeqManII*

Figure 3-1. DNA Fragment Assembly. Automatic fragment assembly programs align overlapping sequences and compute their consensus. In this example, 11 sequences have been aligned and their consensus appears across the top.

The third representation of traces is as 2-D graphs made by interpolating the sequence of trace intensities. The graphs are studied by human editors to assist in resolving ambiguous calls, fine-tuning alignments, and merging contigs (Rowen & Koop 1994). The trace data output from an ABI DNA sequencer is found in the data files of the ABI *Analysis* program. There are four sets of data for a fragment of DNA – one for each of the four fluorescent dyes. The traces appear in two forms; one is a sequence of raw intensities, and in the other, the data has been processed via a proprietary algorithm such that trace peaks are more distinct and uniform. It is the processed data that is used to produce the graphs that are made available to users of fragment assembly programs such as DNASTAR Inc.'s *SeqmanII*, Gene Codes's

Sequencher, and *PhrapView* from the University of Washington.

The goal is to reduce the need for time-consuming and expensive manual editing processes. My premise is that this can be accomplished by the direct analysis of trace characteristics in automatic processes. I examined the decisions made by human editors and observed that most of their decisions are quite straightforward. I believe that with proper input representations, many such decisions can be made automatically. The problem is that current representations are inadequate for incorporation into automatic processes. Sequences of intensities are cumbersome and undescriptive, sequences of base calls are too coarse and crucial information is lost, and 2-D graphs are extremely complex to incorporate directly. My approach is to define a new representation that captures the same information used by editors in their work in such a way that it can be easily incorporated into automated tasks.

3.2 *Trace-Class*

My personal observation is that while studying traces, human editors pay particular attention to the relative intensities and characteristic shapes of trace data. It is a measure of these shapes and relative intensities found in the 2-D graphs of processed ABI data that I describe in my new representation. By representing this information, I can make available to an assembly program the same information that is available to editors. I call the new representation *Trace-Class* (Allex *et al.* 1996).

For my new *Trace-Class* representation, I am interested in classifying the shape and intensity of the local trace data that is used for each particular base call. I define this local trace data to be the data from midway between the previous call and the current call to the data midway between the current call and the next call (Figure 3-2). I will refer to each of these intervals of data as *base trace-data*. Each set of base trace-data is composed of about 10 to 15 data points representing the intensities of the fluorescent dyes.

In examining the discrete regions of data associated with a single base call, I observe six basic shapes. For the *Trace-Class* representation, I classify base trace-data according to these six shapes. Figure 3-3 illustrates the six classes and the criteria used to distinguish among them. There are three *peak* classes and three *valley* classes defined by curvature. At the base call location, peaks have negative curvature and valleys have positive curvature. Peak and valley classes each come in three varieties: *strong*, *medium*, and *weak*. Whereas peaks and valleys are differentiated by curvature, slope distinguishes among strong, medium, and weak character. Strong is characterized by a change in the sign of the slope, medium is characterized by the occurrence of a shoulder with a slope of zero, and weak has neither a change in sign nor an area of zero slope.

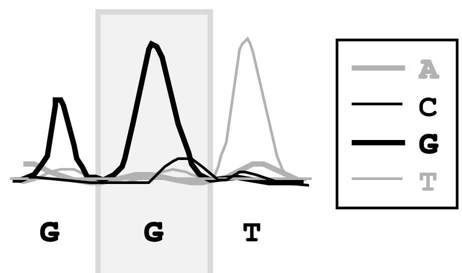


Figure 3-2. ***Base Trace-Data.*** The new *Trace-Class* representation is designed to capture the visual characteristics of the trace data for a single base call as shown in the shaded region.

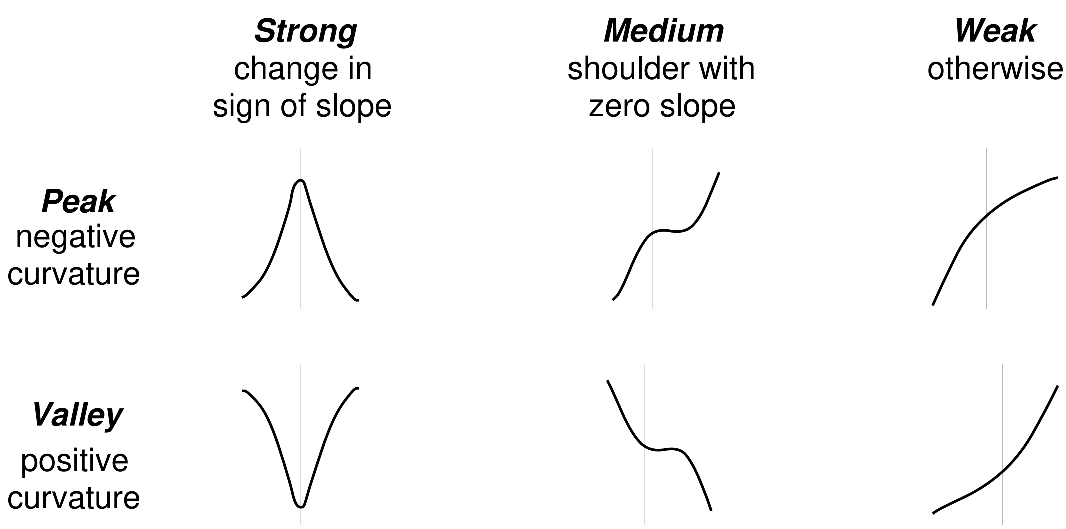
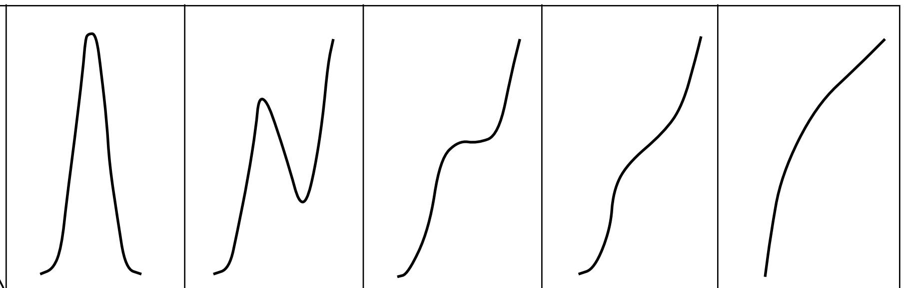


Figure 3-3. Stereotypical Trace-Classes. *Peaks* and *valleys* are defined by curvature, and *strong*, *medium*, and *weak* classes are defined by slope. The gray line indicates the location of the base call.

I believe that too much information would be lost by simply assigning to base trace-data the single class that best characterizes it. Rather, to each of the six classes I assign a score from 0 to 100 that reflects the amount of character of that class that is exhibited by the data. Figure 3-4 presents examples of scores that are assigned to various peak shapes (valley scores are omitted for clarity). The scores are on a continuum, so that any pair of adjacent scores that sum to 100 is possible. In fact, the scores may sum to less than 100 since they are also adjusted to reflect the distance from the base call location and intensity relative to the other three traces.



Shape / Class					
Strong	100	50	0	0	0
Med	0	50	100	50	0
Weak	0	0	0	50	100

Figure 3-4. Peak Score Examples. As the descending slope increases, the scores change from 100% strong to 50/50% strong/medium, to 100% medium, and so on.

The trace data associated with a single base may contain a peak, or a valley, or both a peak and a valley. The base is called at a particular point in the trace data – I assign scores for both the peak and the valley that are the closest to this location. The class scores are weighted by proximity to the base-call location. Peaks or valleys that are closer to where the base is called have a relatively higher score than those that are further away.

Sometimes I may need to make comparisons among the four sets of trace data associated with a single base call. For this situation, the class scores are adjusted to reflect the relative differences among the intensities (heights) of the A, C, G, and T peaks or valleys; higher peaks score higher than lower peaks, and lower valleys score higher than higher valleys.

3.2.1 Algorithmic Details

The sequence of trace data points is scanned for strong peaks and valleys, then for medium peaks and valleys, and finally, if neither of these is found, a weak peak or valley is assumed. These steps are summarized next and in Appendix B.

Assign Trace-Class Scores

Assign Strong and Medium Scores
If no scores assigned then
    Assign Medium and Weak Scores
If no scores assigned then
    Assign Weak Scores

Since multiple peaks or valleys may exist in the data, at each step, I look for the peak and valley that are the closest to the point where the base was called. Scores are assigned based on proximity to the base-call location, relative intensity, and on the amount of strong, medium, and weak character exhibited.

Each class score is adjusted as it is computed to reflect the proximity of a peak or valley to the location where the base is called. The scores are adjusted as follows.

$$S_{new} = S_{old} * (1 - |E - B| / N)$$

where

S is a *Trace-Class* score
E is the location of the peak or valley
B is the location of the base call
N is the number of base trace-data points

Peaks and valleys that are closer to where the base is called get higher scores since they are the ones that are most likely to have been detected by the base calling software.

After the *Trace-Class* scores have been computed for all four sets of trace data for a base, the scores are modified to account for the relative intensity differences among them. The following formulas accomplish this.

$$P_{new} = P_{old} * (P / max(T))$$

$$V_{new} = V_{old} * (1 - V / max(T))$$

where

P is a strong, medium, or weak peak score

V is a strong, medium, or weak valley score

T is the set of four base trace-data values

Higher peaks and lower valleys get higher scores.

I first examine the data for strong peaks or valleys. An overview of the algorithm is given next; pseudocode details are contained in Appendix B.

Assign Strong and Medium Scores

```
For each trace data point
    Compare previous slope to current slope
    If the slope goes from positive to negative, a peak is found then
        If this peak is closer to base call location than any previous then
            Save this peak
    Else if the slope goes from negative to positive, a valley is found then
        If this valley is closer to base call location than any previous then
            Save this valley
If a peak was found then
    Assign SP and MP scores
    Adjust scores for peak distance from base call location
If a valley was found then
    Assign SV and MV scores
    Adjust scores for valley distance from base call location
```

A strong peak is detected when there is a change from a positive to a negative slope, and likewise, a strong valley is detected when there is a change from a negative to a positive slope. The slopes are measured as the change in intensity from one data point to the next. In my observations, this sensitive measure of change in direction works well since the ABI data has been smoothed during processing; insignificant changes in the direction of the slope rarely occur. If a strong peak or valley is found, it must be checked for amount of strong and medium character. Peaks that start at the baseline (zero intensity) and return to the baseline are scored as 100% strong and 0% medium. The same is true for valleys that start at the maximum intensity, descend to the baseline, and return to the maximum intensity. Any other peaks or valleys found in this step possess a combination of strong and medium strengths.

To calculate the strong and medium scores, I measure the local size of the peaks and valleys. I do this by looking on either side of the peak or valley to find extremes where the slopes again change directions (changing from positive to negative or vice-versa). If there is no change on one or both sides, the first and/or last intensity value(s) are used. The intensities at the extreme locations are used in determining the fraction of the total height of the local area that is the peak or valley. Three local extremes are thus used in the calculation: one at the center of the peak or valley, and one to each side. The scores for strong and medium classes are computed as follows.

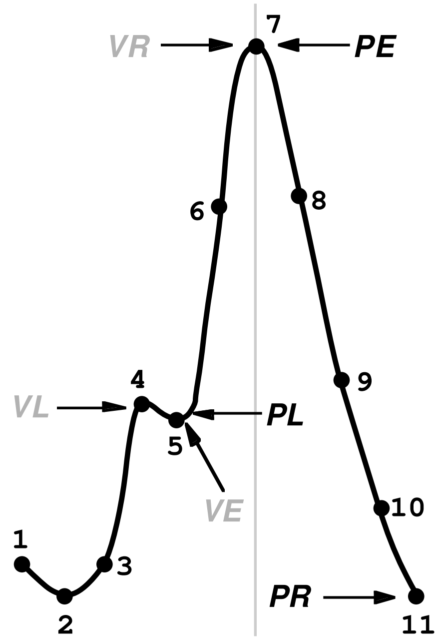
$$SP = 100 * (E - (L + R) / 2) / E$$
$$MP = 100 - SP$$
$$SV = 100 * ((L + R) / 2 - E) / (L + R) / 2$$
$$MV = 100 - SV$$

where

SP is a strong peak score
MP is a medium peak score
SV is a strong valley score
MV is a medium valley score
E is the value at the peak or valley location
L is the value of the extreme to the left of the E location
R is the value of the extreme to the right of the E location

An example of some data and strong and medium scores are shown in Figure 3-5. In the new *Trace-Class* representation, a peak that is 82% strong and 18% medium is at the base-call location. A valley with 40% strong and 42% medium strength has been detected to the left of where the base was called. The valley scores have been adjusted to reflect that the valley is offset from the base-call location.

Intensities Location	Value
1	111
2	42
3	130
4	551
5	528
6	1098
7	1536
8	1110
9	628
10	276
11	31



Strong Peak	82
Medium Peak	18
Strong Valley	40 (49)
Medium Valley	42 (51)

PE	peak extreme
PL	peak left extreme
PR	peak right extreme
VE	valley extreme
VL	valley left extreme
VR	valley right extreme

Figure 3-5. ***Trace-Class*** **Strong/Medium Scores.** A base has been called at point 7 (at the gray line). A strong peak is detected at that point and its left extreme is at point 5 where the slope changes direction. Since there is no change in the slope direction to the right of the peak, the last point, 11, is the location of the right extreme. (The peak at point 4 is not scored since the peak at 7 is closer to the base call location.) The peak scores are calculated as:

$$SP = 100 * (1536 - (528 + 31)/2)/1536 = 82$$

$$MP = 100 - 82 = 18$$

Since the peak is located where the base is called, the scores are not adjusted for distance.

A valley and its right and left extremes are at points 5, 4, and 7, respectively. The valley scores are calculated as:

$$SV = 100 * ((551 + 1536)/2 - 528)/(551 + 1536) /2 = 49$$

$$MV = 100 - 49 = 51$$

The valley scores have been adjusted to reflect the distance of the valley from the point where the base was called. (The scores prior to adjustment are in parentheses.) The adjustment calculation is:

$$SV = 49 * (1 - |5 - 7| /11) = 40$$

$$MV = 51 * (1 - |5 - 7| /11) = 42$$

If no strong peaks or valleys are found, the data is scanned for peaks or valleys of medium strength. An overview of the algorithm is listed next; pseudocode detailing the calculations is contained in Appendix B.

Assign Medium and Weak Scores

```
For each trace data point
    Compare previous slope and current slope
    If the slope decreases (a peak is found) then
        If an increasing slope (a valley) was previously found then
            If the peak is closer to base call location than any previous then
                Save this peak
            If the previous increasing slope is closer to base call location
                            than any previous valley then
                Save the increasing slope location as a valley
    Else if slope increases (a valley is found) then
        If a decreasing slope (a peak) was previously found then
            If the valley is closer to base call location than any previous then
                Save this valley
            If the previous decreasing slope is closer to base call location
                            than any previous peak then
                Save the decreasing slope location as a peak
If a peak was found then
    Assign WP and MP scores
    Adjust scores for peak distance from base call location
If a valley was found then
    Assign WV and MV scores
    Adjust scores for valley distance from base call location
```

A medium peak is found when a decreasing slope is found that is either preceded or followed by an increasing slope (a valley). The peak is located at the point where the largest change in slope occurs in the decreasing slope. Likewise, if an increasing slope is preceded or followed by a decreasing slope (a peak), a valley is identified. The location of the valley is at the point where the largest change in slope occurs in the increasing slope.

If a medium peak or valley is found, the amount of medium and weak character is computed. To assign medium and weak strengths I determine the fraction of the overall height of the local area that is the peak or valley. I do this by first finding the locations of the preceding or following peak (for valley scores) or valley (for peak scores). Often, there is not both a preceding and following peak or valley. In this case, the first or last data point location is used. The intensities at these locations and that of the peak or valley are the three locations used in the following calculation of medium and weak scores.

$$WP = 100 * (max(L,R) - E) / max(L,R)$$
$$MP = 100 - WP$$
$$WV = 100 * (E - min(L,R)) / E$$
$$MV = 100 - WV$$

where

MP is a medium peak score
WP is a weak peak score
MV is a medium valley score
WV is a weak valley score
E is the value at the peak or valley location
L is the value at the increasing or decreasing slope to the left of the *E* location
R is the value at the increasing or decreasing slope to the right of the *E* location

The computation of the medium class scores defined here do not conflict with the computation given for assigning strong and medium scores since medium and weak scores will not be calculated if strong and medium scores were already calculated in the previous step.

An example of the medium and weak scores calculated for some data is shown in Figure 3-6. A peak that is 85% medium and 5% weak is detected at the left of the base-call location. A valley with 77% medium and 5% weak strength has been detected to the right of where the base was called. The peak and valley scores have been adjusted to reflect that they are offset from the base-call location.

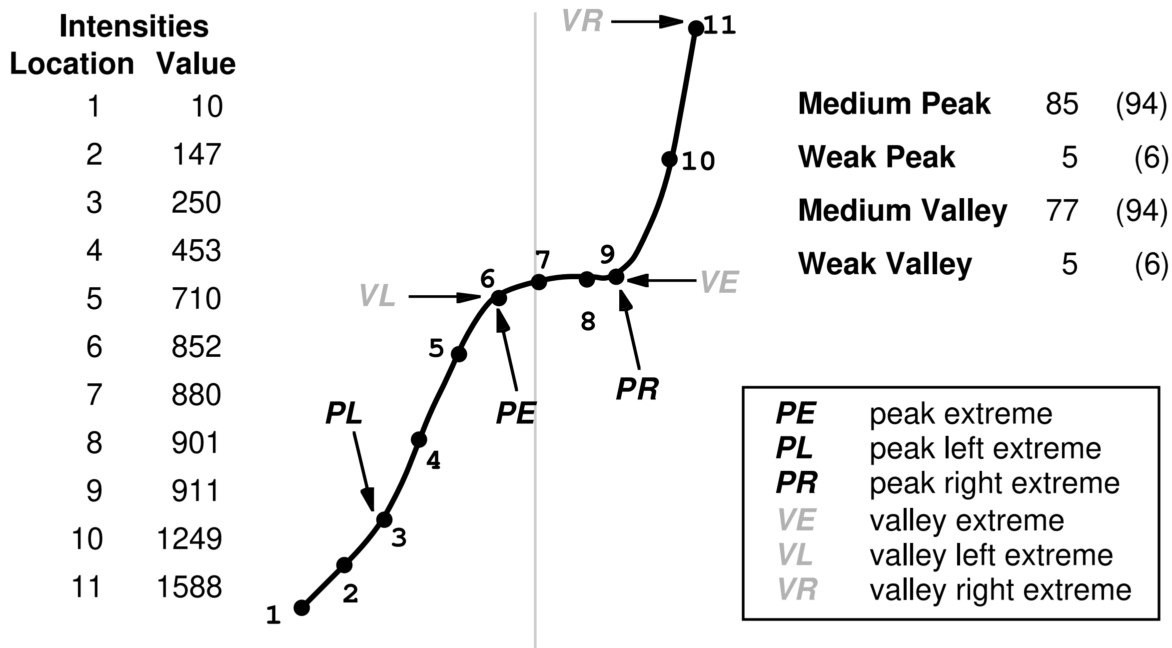


Figure 3-6. ***Trace-Class*** **Medium/Weak Scores.** A base has been called at point 7 (at the gray line). A medium peak is detected at point 6 and the locations of its left and right slope extremes are points 3 and 9 respectively. The peak scores are calculated as:

$$WP = 100 * (max(250, 911) - 852)/max(250, 911) = 6$$

$$MP = 100 - 6 = 94$$

A medium valley is detected at point 9 and its left extreme is at point 6 where a decreasing slope (peak) is found. Since there is no decreasing slope (peak) to the right of the valley, the last point, 11, is the location of the right extreme.

$$WV = 100 * (911 - min(852, 1588))/911 = 6$$

$$MV = 100 - 6 = 94$$

Both the peak and the valley scores have been adjusted to reflect their distances from the point where the base was called. (The scores prior to adjustment are in parentheses.) The scores are adjusted as:

$$WP = 6 * (1 - |6 - 7| /11) = 5$$

$$MP = 94 * (1 - |6 - 7| /11) = 85$$

$$WP = 6 * (1 - |9 - 7| /11) = 5$$

$$MP = 94 * (1 - |9 - 7| /11) = 77$$

Finally, if the data has not yet been classified in the strong or medium assignment steps, a weak peak or valley is assumed and assigned a 100% weak score. The algorithm is listed next; detailed pseudocode is in Appendix B.

Assign Weak Scores

Compare slopes preceding and succeeding the base call location
If previous slope is greater than succeeding slope, a peak is found then
    Assign *WP* score
Else if previous slope is less than succeeding slope, a valley is found then
    Assign *WV* score

Partial weak and medium scores are not assigned here since that would have been done in the previous step. Figure 3-7 illustrates the assignment of weak scores to sample data.

Intensities Location	**Value**
1	11
2	210
3	407
4	578
5	762
6	938
7	1098
8	1252
9	1361
10	1499
11	1597

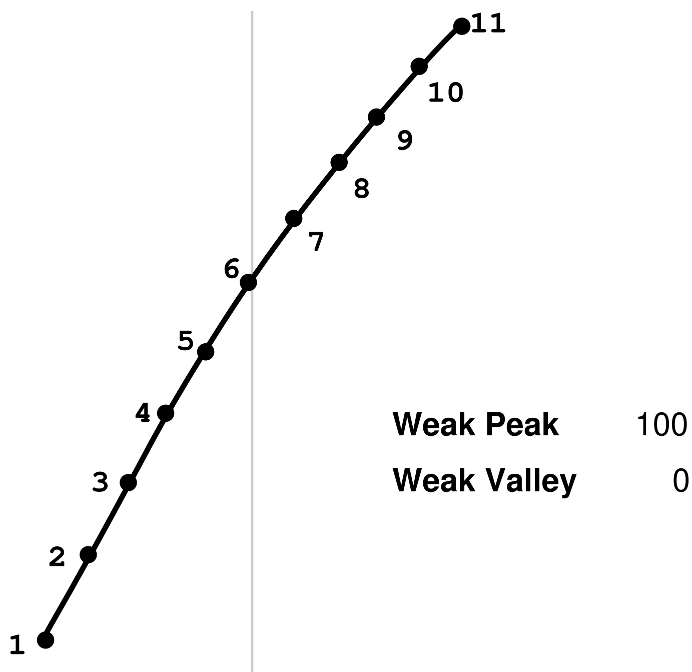


Figure 3-7. *Trace-Class* Weak Scores. A base has been called at point 6 (at the gray line). The trace has a decreasing slope at point 6, so a weak peak score of 100% is assigned.

Each class in the *Trace-Class* representation is now assigned a score between 0 and 100. For some applications, I may need to use a simpler representation of a *Trace-Class*. For these cases I choose the single class that best characterizes the data. This class, the *characteristic class,* is assigned by first selecting peak or valley according to which has the higher sum of scores, and then strong, medium, or weak according to which has the highest score. For example, if a set of trace data is assigned scores of *SP=75*, *MP=14*, *WP=0*, *SV=0*, *MV=11*, and *WV=3*, peak has the higher sum of scores (75 + 14 + 0 = 89) compared to valley (0 + 11 + 3 = 14), and the highest scoring class is strong (75). Given this, the single characteristic class is strong peak.

3.2.2 Base-Call Weights

One of my uses for *Trace-Class* scores is to aid in determining the local quality of the trace surrounding each base. The quality of traces can vary widely both among and within traces. Since I incorporate trace information into automatic processes, it is important that I take these quality differences into consideration. In making decisions, I want to give more weight to information that comes from more reliable (higher quality) traces. To facilitate this goal, I assign a weight score to each base call that reflects the quality of the trace in the locality of the base.

I use the *Trace-Class* scores as an indicator of trace quality. My premise is that base calls that are highly reliable are made from trace peaks that are sharp and well-defined – those that classify as strong peaks. Base calls made from base trace-data that classify as medium peaks are less reliable, and those made from weak peaks or valleys are even less reliable. Therefore, to determine quality, I examine the *Trace-Class* scores for the traces associated with the called bases. (For example, if the sequence of bases has been called as *GGTACG*, only the *Trace-Class* scores for the corresponding *G*, *G*, *T*, *A*, *C*, and *G* traces are calculated.) If the strong peak scores are high, it is likely that the trace is of good quality – the higher the scores, the better the quality of the data. On the other hand, if the bases have been called with low peak scores or valley scores, the base calls are not as obvious and the data is likely to be less reliable and of lower quality. Figure 3-8 compares the relative quality of some traces.

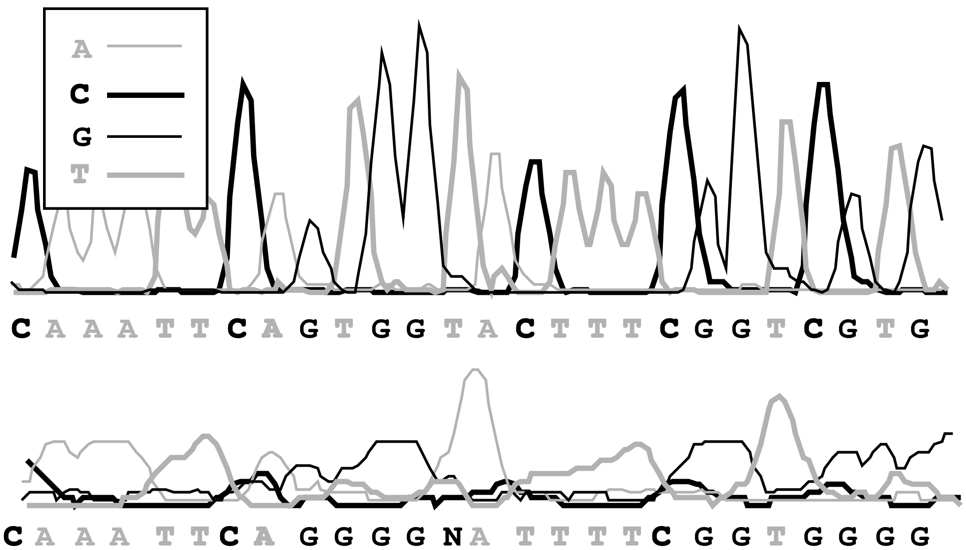


Figure 3-8. Quality of Trace Data. In the top sequence, the trace associated with each called base exhibits a sharp well-defined peak. The corresponding *Trace-Class* scores all show high strong peak scores. In contrast, the traces in the bottom sequence are flattened and overlapping; corresponding strong peak scores are generally much lower. The top sequence is much more reliable and is of high quality. (Actual data shown.)

For use in my calculations of weights, I define a constant summary vector, V, so that classes (such as SP) that imply better-quality data for a base are given higher values than those (such as SV) that imply lower-quality data. The definition follows.

Let

$$V = [V_{SP}\ V_{MP}\ V_{WP}\ V_{WV}\ V_{MV}\ V_{SV}]$$

where

V_i is the multiplier for class i

and

$$1 \geq V_{SP} \geq V_{MP} \geq V_{WP} \geq V_{WV} \geq V_{MV} \geq V_{SV} \geq 0$$

In upcoming sections I describe work that uses weights. While performing tests for the work, I find that $V = [1\ .67\ .33\ 0\ 0\ 0]$ yields good results. Using this definition, the weights, W, are between 0 and 100 since peak classification scores sum to 100 or less. I chose this summary vector from among those listed in Table 3-1. I chose these vectors for evaluation because they all conform to the restriction $1 \geq V_{SP} \geq V_{MP} \geq V_{WP} \geq V_{WV} \geq V_{MV} \geq V_{SV} \geq 0$, while varying the emphasis on the scores for different classes. Two of the vectors, described in the table as *Linear Peaks* and *Parabolic Peaks*, assign zero values to the three valley classes. I find that these vectors consistently outperformed the others (results not reported). This indicates that not only do peak scores contain sufficient information, but also that valleys in a trace rarely or never suggest a base call corresponding to the trace.

Once a summary vector has been set, a weight for each base can be determined. Rather than assigning a score that only reflects the quality of a single base, I set scores that indicate the quality of the trace in the local area surrounding a base. To do this, I calculate values for each individual base in a window surrounding the base of interest and then average the values. Specifically, the weight of the data for each sequence in a column is calculated as follows.

1. For each base, i, in a window of size n centered on the base of interest, calculate the vector of *Trace-Class* scores, S_i, for the trace associated with the base that has been called:

   $$S_i = [SP_i \;\; MP_i \;\; WP_i \;\; WV_i \;\; MV_i \;\; SV_i]$$

2. The dot product of S_i and V produces a quality measure, W_i, for base i:

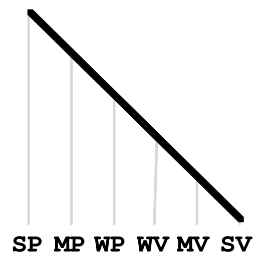
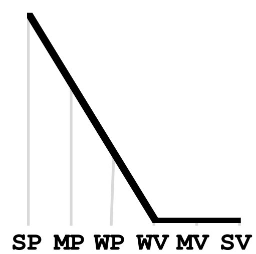
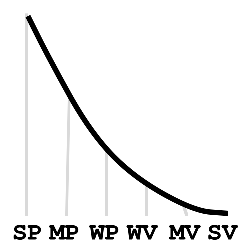
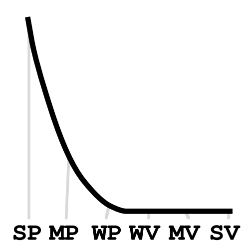
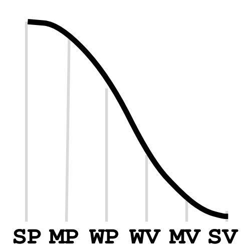
   $$W_i = S_i \bullet V$$

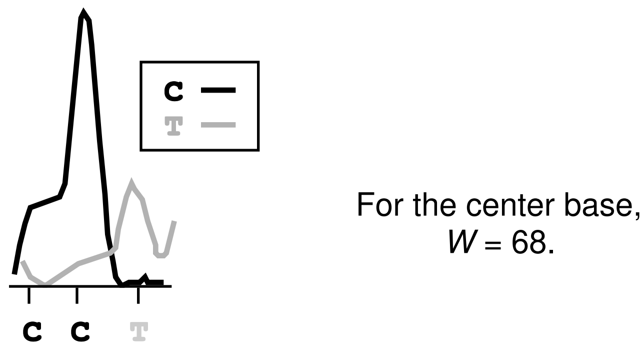
3. Average the measures to produce an overall quality score, W, for the base at the center of the window:

   $$W = (W_1 + W_2 + \ldots + W_n) \;/\; n$$

Figure 3-9 contains an example calculation of a weight for a window size, n, of 3.

Table 3-1. Potential Summary Vectors.

Description	Function	Graph	***V*** (summary vector)
Linear	$V_i = \frac{5-i}{5}$		[1 .8 .6 .4 .2 0]
Linear Peaks	$V_i = max\left(0,\frac{3-i}{3}\right)$		[1 .67 .33 0 0 0]
Parabolic	$V_i = \left(\frac{5-i}{5}\right)^2$		[1 .64 .36 .16 .04 0]
Parabolic Peaks	$V_i = \left(max\left(0,\frac{3-i}{3}\right)\right)^2$		[1 .44 .11 0 0 0]
Trigonometric	$V_i = \frac{cos\left(\frac{\pi}{5}i\right)+1}{2}$		[1 .9 .65 .35 .1 0]



	SP	MP	WP	WV	MV	SV	S•V
C	0	59	41	2	8	0	53
C	83	17	0	1	5	0	94
T	43	21	0	0	2	7	57
						Total	204
						W	68

Figure 3-9. Weight Calculation. In this example, the window size, n, is 3 bases. (The actual value of n in my work is 21, a value found to work well in experiments – results unreported.) I want to calculate the weight score, W, for the center base, C. Three sets of *Trace-Class* scores, S, have been calculated: one for each of the C traces corresponding to the first two C base calls, and a third for the T trace data associated with the T call. The dot product of each set of scores with the summary vector ([1 .67 .33 0 0 0]) is computed. The average of the three is the weight for the C base in the center of the window.

Figure 3-10 graphs an actual example of the computed weight values of data as a function of base position. The graph shows a fast increase to a high quality region followed by a slow decrease in quality. The pattern is as expected in ABI sequences (Chen & Hunkapillar 1992).

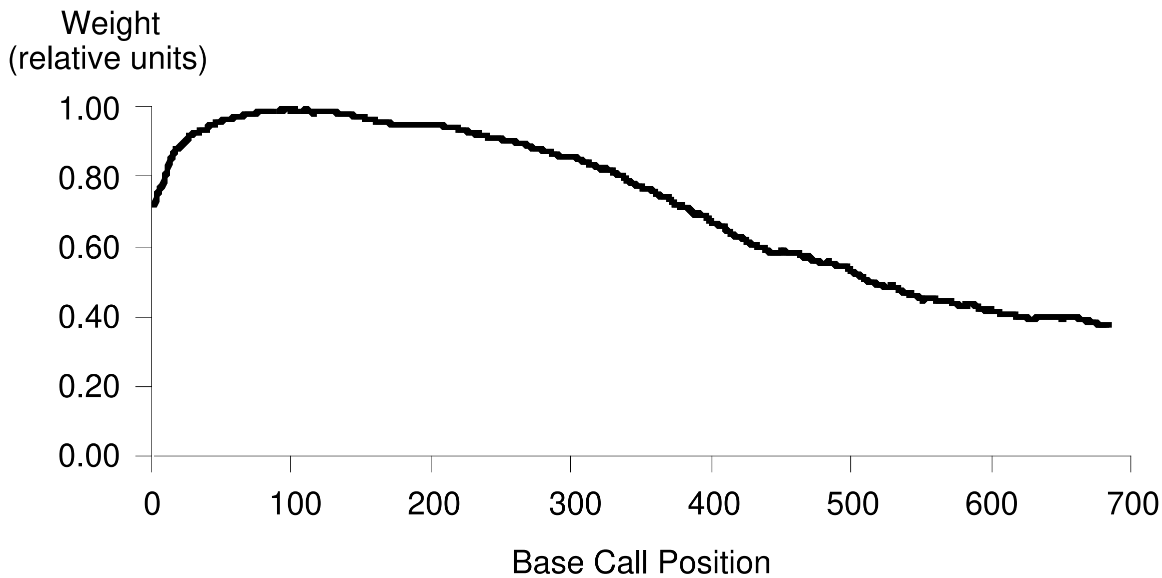


Figure 3-10. Quality of Traces. Weight as a function of position averaged over the first 700 base calls of 116 actual sequences is graphed. Base calls made by ABI sequencers are highly reliable out to several hundred bases. Reliability slowly decreases to the 3' end of the sequence.

3.3 Summary

Virtually all large-scale sequencing projects use automatic sequence-assembly programs to aid in the determination of DNA sequences. The computer-generated assemblies require substantial manual editing to transform them into submissions for GenBank. As the size of sequencing projects increases, it becomes essential to improve the quality of the automated assemblies so that this time-consuming manual editing may be reduced. Current ABI sequencing technology uses base calls made from fluorescently-labeled DNA fragments run on gels. I present a new representation, *Trace-Class*, for the fluorescent trace data associated with individual base calls. In summary, I define a *Trace-Class* representation of *base trace-data* as follows:

- Three *peak* and three *valley* classes are defined as follows.

  Peaks: negative curvature
  Valleys: positive curvature

- Peak and valleys are divided into *strong, medium,* and *weak* classes that are defined as follows.

Strong: change of sign in slope
Medium: shoulder with zero slope
Weak: otherwise

- Scores reflect the amount of strong, medium, or weak peak and valley character exhibited.
- Scores reflect the proximity of peaks and valleys to the base-call location.
- Scores reflect relative intensity to corresponding traces.
- A single class (the *characteristic class*) may be assigned that best characterizes the data.

In Figure 3-11, the new *Trace-Class* representation of trace data as six *Trace-Class* scores based on the shape and intensity of trace data is contrasted with the previous representations of trace data as sequences of discrete intensity values, 2-D graphs, and a base call.

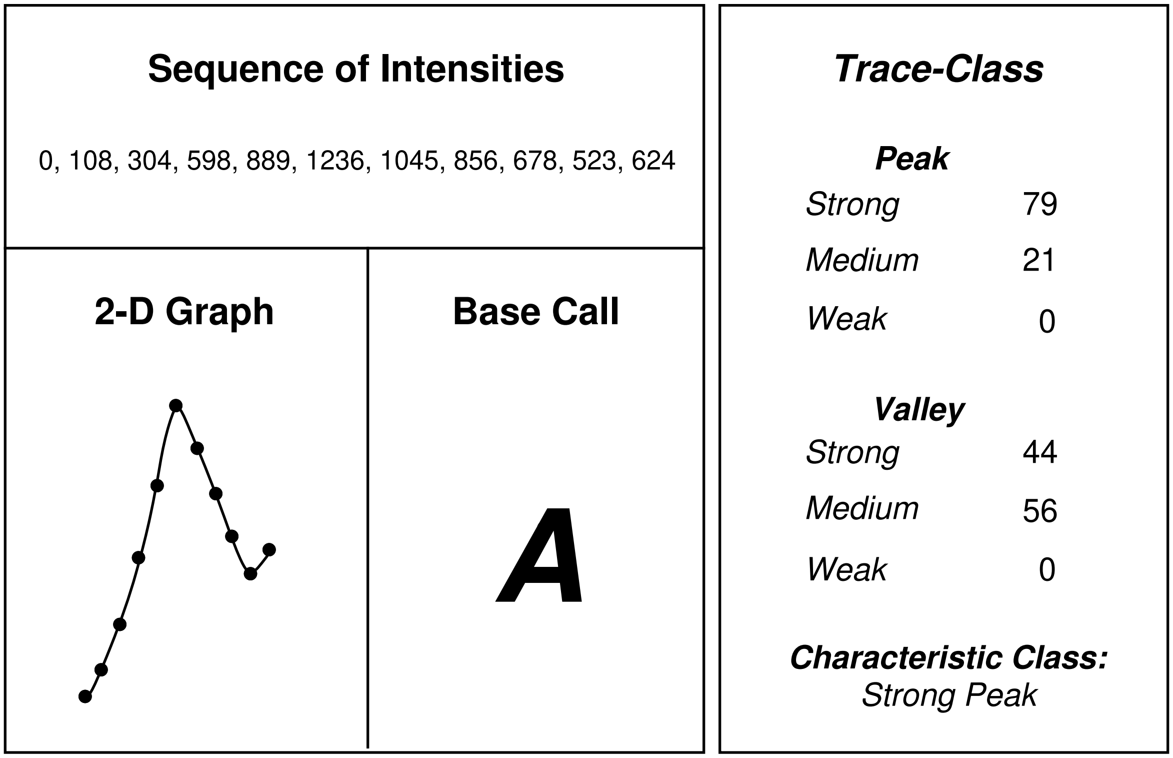


Figure 3-11. Comparison of Trace-Data Representations. Shown in this figure are four representations of a single fluorescent trace (the *A* trace). One representation of trace data is a *sequence of intensities* associated with a base call. A *2-D* graph of the trace data shown as a curve interpolated from the data points is a second representation. A much simplified representation is the *base call* made from the four traces. The new *Trace-Class* representation is a classification of the trace data based on the visual shape and intensity of the trace data. A score from 0 to 100 is assigned for each of six classes that reflects the amount of strong, medium, and weak peak and valley characteristic that is exhibited by the data. A single characteristic class is assigned that best characterizes the data.

Chapter 4

Sequence End-Trimming Case Study

In Chapter 3 I defined a new representation for trace data, and the question is: *Does it effectively capture characteristics of trace data such that it can be used to improve the quality of automatic assemblies?* To answer this question, I incorporated the trace-data information via the new *Trace Class* representation into two processes in automatic assembly: *end-trimming* of sub optimal data before assembly and *calling the consensus* of aligned sequences. The end-trimming experiments are described in this chapter; work on consensus calling is covered in Chapters 5 through 7. For these studies, I use modifications of DNASTAR Inc.'s *SeqMan* fragment-assembly software.

The first problem I addressed is how to remove poor quality data before assembly (Allex *et al.* 1996). In general, as a sequencing run progresses, the quality of the trace data deteriorates (Kelley 1994). This idea is illustrated in Figure 4-1. Near the 5' (beginning) end, the data quality is high – peaks are sharp and well-defined (Perkin-Elmer 1995), but near the 3' end (end), the data is erratic and contains several no-calls (*N*s). The use of poor quality data like that near the 3' end tends to produce a poor quality assembly that requires extensive manual editing. One solution is to remove the poor quality data before assembly (Seto, Koop & Hood 1993, McCombie & Martin-Gallardo 1994, Rowen & Koop 1994). This is the approach I explored in my work, with the goal of reducing the amount of poor quality data by a direct, automatic examination of trace data.

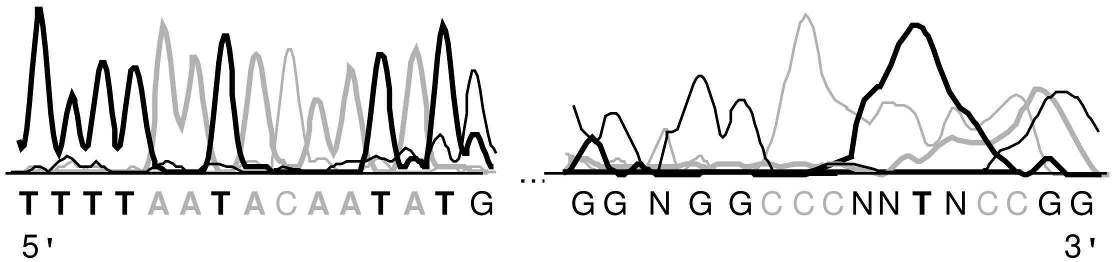


Figure 4-1. Deterioration of Trace Data. Trace data becomes progressively lower in quality as a gel is read. (Actual data shown.)

4.1 Existing Method

A scheme that uses sequence-specific information to trim poor quality data is *N-Trim*, an adaptation of the *End-Clip* method (Seto, Koop, & Hood 1993). *N-Trim* relies on the assumption that as trace data deteriorates, the number of no-calls increases. The idea is that the number of *N*s can be used as an indication of trace quality. With this procedure, bases are scanned in a sliding window starting at the 3' end, and a count is kept of the number of *N*s that occur in the window. When the number of *N*s is sufficiently few, the poor quality data from that window to the 3' end of the sequence is trimmed off. In the example in Figure 4-2, `window_size` is set to 20 and the `max_Ns` allowed in a window is two. Scanning from the 3' end of the sequence, the boxed window is the first one that contains two or fewer *N*s. The data to the 3' end of this window is considered poor quality and is trimmed off.

Figure 4-2. *N-Trim*. The `window_size` is 20 and `max_Ns` is two. When scanning from the 3' end, the boxed window is the first to contain two or fewer *N*s and data in the shaded region is trimmed.

My method for end-trimming that incorporates trace data via *Trace-Class* scores is called *Trace-Class Trim.* This method also scans data in windows. As it scans, it assigns the characteristic class associated with the called base. The classes can be used to indicate the quality of data. Base calls that are highly reliable are made from peaks that are sharp and well-defined – the kind of data classified as strong peaks. Base calls made from trace data classified as medium peaks are less reliable, and those made from weak peaks or valleys are correspondingly less reliable.

4.2 Algorithmic Details

Trace-Class Trim consists of scanning the window from the 3' end as in *N-Trim*, but rather than keeping track of the number of *N*s, it counts the number of poor classes associated with the called base. When the number of poor classes is sufficiently few, I trim the data from that window to the 3' end of the sequence. In addition to `window_size` and `max_poor_classes` allowed, I must also define `poor_classes`. Figure 4-3 shows an example in which `poor_classes` is defined to be all classes except strong and medium peaks (*SP* and *MP*). The `window_size` is again set to 20 and `max_poor_classes` allowed in the window is two. When the sequence has been scanned as far as the boxed window, the number of poor classes is sufficiently few and data from the window to the 3' end is trimmed. Appendix C contains pseudocode for *Trace-Class Trim*.

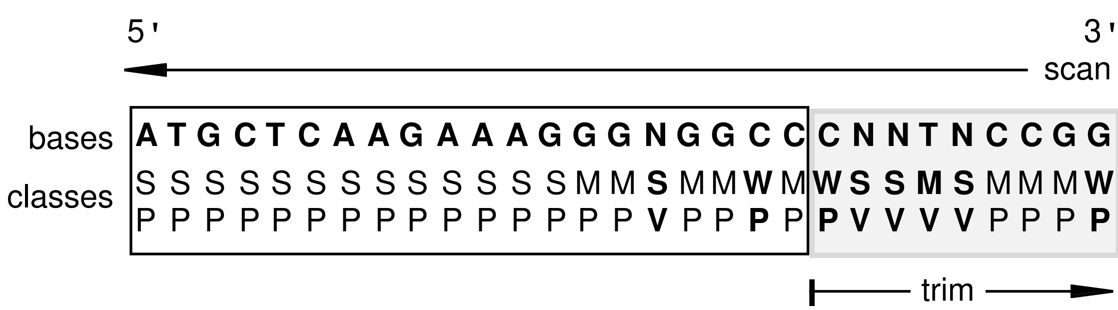


Figure 4-3. *Trace-Class Trim*. In this example, characteristic class for the *A* trace data associated with this leftmost base call is strong peak (*SP*), the characteristic class for *T* trace data associated with the next base is also *SP*, and so on. Here, `poor_classes` are defined as all but strong and medium peaks (*SP* and *MP*). The `window_size` is 20 and `max_poor_classes` is two. When scanning from the 3' end, the boxed window is the first to contain two or fewer poor classes and data in the shaded region is trimmed.

4.3 Evaluation

I empirically evaluate *Trace-Class Trim* and compare it to *N-Trim* by optimizing the parameters for each method over one set of data and then testing the best parameters on a second set of data. I used data from the *E. coli* Genome Project at the University of Wisconsin that was gathered for an assembly of a 243 kb fragment of *E. coli* (Blattner *et al.* 1997). Data sets were formed in the following way. The 2021 sequences in the set of data for the assembly were trimmed extensively such that only bases from locations 50 to 200 remained in each sequence. To this set, I added longer *E. coli* sequences from GenBank that were believed to fall in the 243 kb section of the *E. coli* genome. The sequences were then automatically assembled. In this way, only the very best data was used and contigs were formed with sequences that should align (given the nearly ideal data).

All contigs containing ten or more sequences were chosen for inclusion in data sets. In these contigs, the GenBank sequences were removed and the full untrimmed length of sequences was reinstated. Each set of sequences in a contig formed a separate data set, called a *project*, that could be independently assembled. The result was 20 projects for evaluating trimming methods. Ten projects form a *training set* used to optimize parameters and the other ten sets form a *test set* used to test the quality of subsequent assemblies using the optimized parameters. Training and test sets were chosen such that the number of projects is equal and the total contig lengths and total numbers of sequences and contigs are similar.

For use in my evaluations, I estimated the expected number of contigs and total contig length for each project. Although each project is formed from a single contig, in some cases, the expected number of contigs is greater than one because regions in the contig were bridged by (now removed) GenBank sequences. To estimate the expected total contig length, I simply use the length of the contigs after they have been extended with complete, untrimmed sequences. The data sets are described in Table 4-1.

Table 4-1. End Trimming Data Sets. The number of sequences is the actual number and the number of contigs and the contig length are the expected values for the project.

(a) Training Set

Project	Number Sequences	Number Contigs	Contig Length
1	11	2	2235
2	14	1	1715
3	15	1	2364
4	18	1	3352
5	20	2	5229
6	22	1	1473
7	26	1	824
8	32	3	7067
9	69	3	11,088
10	37	3	9050
Total	264	18	44,397

(b) Test Set

Project	Number Sequences	Number Contigs	Contig Length
1	20	2	2810
2	16	1	1221
3	18	3	4271
4	24	3	6221
5	27	2	4503
6	35	2	6696
7	38	1	776
8	13	2	3010
9	15	1	3408
10	57	3	11,382
Total	263	20	44,298

In addition to the projects in the test set, I evaluated my system with an unrelated set of sequences. These are from a 7 kb segment of human DNA. This project has reached completion so the number of contigs and contig length is known. Table 4-2 describes this set.

Table 4-2. Human DNA Data Set.

Project	Sequences	Number Contigs	Contig Length
Human	98	2	7257

I optimized parameters for the *Trace-Class Trim* method and separately for *N-Trim*. For *N-Trim*, I varied `window_size` from 10 to 50 in increments of five and `max_Ns` allowed in a window from zero to five. For *Trace-Class Trim*, I varied the `window_size` from 10 to 50, `max_poor_classes` to be allowed from zero to five, and the `poor_classes` cutoffs over strong peaks, medium peaks, and weak peaks. Valleys were always included in `poor_classes`. Each project in the training set was assembled with every combination of parameters and the quality of assemblies was evaluated.

The goal of end-trimming is to produce better-quality automated assemblies of DNA fragments. I used three metrics to measure the quality of assemblies. One is the number of contigs. In general, I want a group of sequences to assemble into a small number of contigs (the ultimate goal is to have only a single contig). The second metric is the number of ambiguities in the consensus sequence. Fewer ambiguities means not only that the sequences align well, but also that less manual work is needed. The third measure is the total length of the contigs. Contigs should be as long as possible without incorporating too many ambiguities.

I measure the number of contigs as the number in excess of the expected number, and contig length as the absolute deviation from the expected length. The number of ambiguities are measured as the average number of ambiguous calls per kb. To score each set of parameters, I normalize and individually sum the three metrics across all data sets for each set of parameters. The overall score, S_i, for parameter set i is

$$S_i = \alpha C_i + \beta T_i + \gamma A_i$$

where C_i, T_i, and A_i are the normalized sums of the number of contigs, total length of contigs, and number of ambiguities metrics, respectively; α, β, and γ are constants. I believe that the order of importance of the metrics is: 1) number of contigs, 2) number of ambiguities, and 3) total length of contigs. Consequently, I set $\alpha = 3$, $\beta = 1$, and $\gamma = 2$ to weight the metrics.

Using the scheme described above, I scored and sorted the parameter sets. I found that, in general, the best *Trace-Class Trim* assemblies resulted when the `window_size` was large (40 to 50 bases), the cutoff defined all but strong and medium peaks as `poor_classes`, and the `max_poor_classes` to be allowed was between 5% and 10% of the window size. The best *N-Trim* assemblies resulted when the `window_size` was large (40 to 50 bases), and the `max_Ns` allowed was small (0 to 2).

The ten minimum scoring parameter sets for *N-Trim* and for *Trace-Class Trim* were chosen as optimal parameter settings. Next, test set projects were assembled using each of the top ten parameter settings for *N-Trim* and *Trace-Class Trim* settings. The human DNA project was assembled using only the top-scoring parameter sets. As a baseline, the projects were also assembled with no trimming.

4.4 Discussion

I compare assemblies resulting from *Trace-Class Trim* to those performed after *N-Trim* and no trimming. Figure 4-4 graphs the results for the ten test-set projects. With one exception, I find that in all 10 sets by all three metrics, trimming with the *Trace-Class Trim* method results in assemblies superior to those produced after *N-Trim* or no trimming. In each column, lower values are better, and a significant reduction between the results for *Trace-Class Trim* and the others is seen. Differences for all three measures are statistically significant using a paired one-tailed t-test at the 95% confidence level.

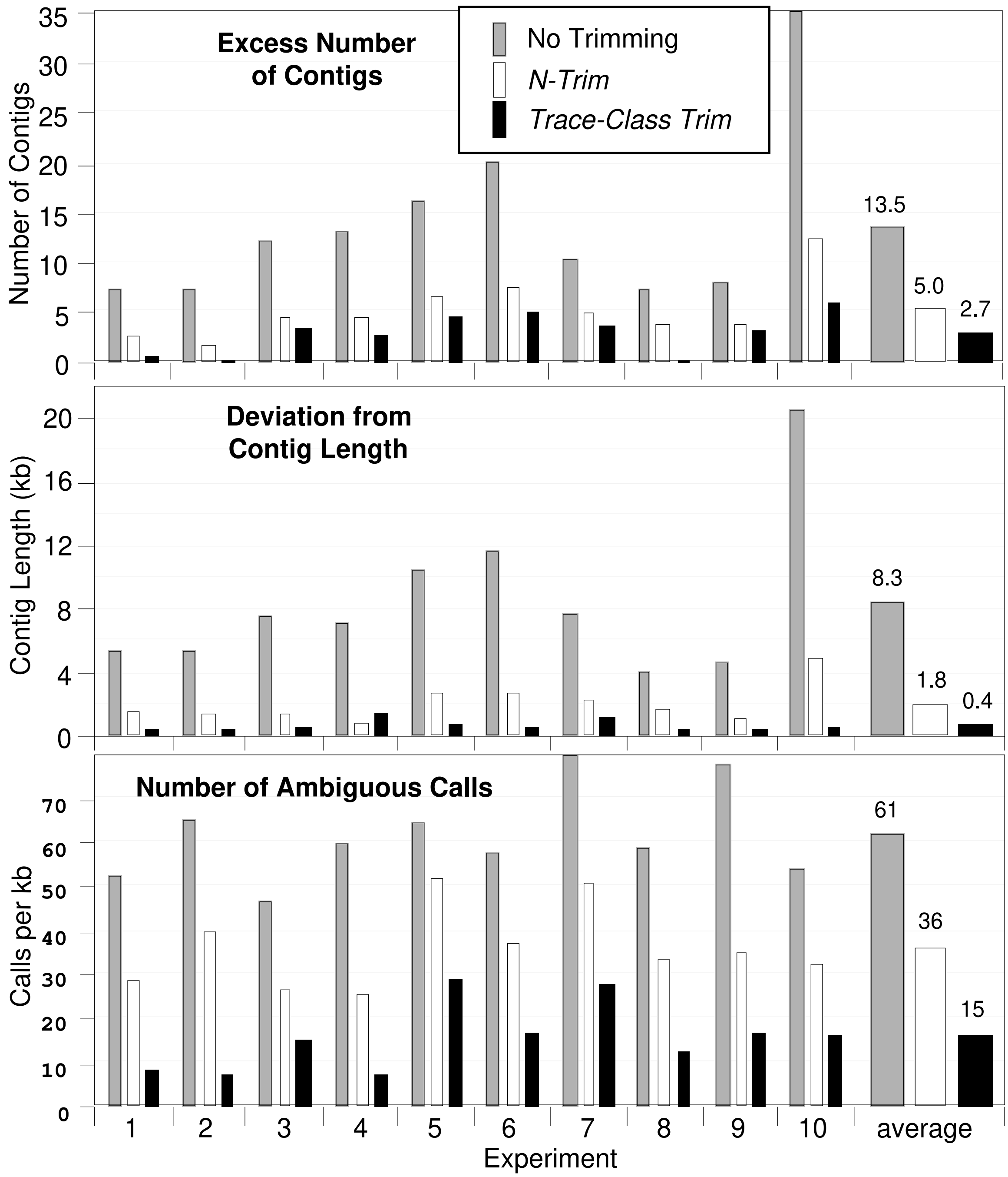


Figure 4-4. End-Trimming Results. Results for the 10 test sets are shown. On average, about a 50% reduction from *N-Trim* to *Trace-Class Trim* is seen for excess contigs and ambiguities per kb. Deviation from contig length falls by over 75%.

On average over the test-set projects, the absolute deviation from the expected length of contigs falls by over 75% and both the excess beyond the expected number of contigs and the number of ambiguities per kb falls by about 50% from assemblies using *N-Trim* to those using *Trace-Class Trim*. The decrease in the number of ambiguities represents a significant decrease in the amount of manual editing that would need to be done on assembled projects. For example, in a 243 kb project, the number of ambiguities to be resolved would decrease from nearly 10,000 bases using *N-Trim* to fewer than 5000 using *Trace-Class Trim*. These results demonstrate a clear improvement when trace-data information is included in end-trimming via the new *Trace-Class* representation.

With the human DNA project, I again see a significant improvement in the assembly done after *Trace-Class Trim* over the assemblies done after *N-Trim* or no trimming. Table 4-3 contains the results for the human DNA project. After *Trace-Class Trim*, the assembly produces three contigs, compared to five contigs with *N-Trim* (the expected number is two). It also results in a 40% reduction in the number of ambiguities per kb over the assembly done after *N-Trim*.

Table 4-3. Human DNA Project Test Results. *Trace-Class Trim* yields an assembly with one more than the expected number of contigs compared to three more with *N-Trim*. The *Trace-Class Trim* assembly had 40% fewer ambiguities than the assembly done with *N-Trim*.

Trimming Method	Excess Number of Contigs	Contig Length Deviation	Ambiguities per kb
Trace-Class	1	576	32
N	3	3113	54
None	13	12,818	149

4.5 Summary

The key to the success of *Trace-Class Trim* is that it uses the information contained in trace data in the form of base *Trace-Class* representation defined in Chapter 3. These classifications directly reflect the morphology of trace data, and are good indicators of the accuracy of the associated base calls. The *N-Trim* method does not use trace data, rather it examines only the sequence of bases for no-calls. Since modern sequencers make base calls even when the trace data is erratic, searching for no-calls as done in *N-Trim* is no longer as useful for assessing the accuracy of base calls.

I incorporated the *Trace-Class Trim* method into the *SeqManII* sequence assembly package that is part of the *Lasergene* suite of applications developed by DNASTAR Inc. The previous version, *SeqMan*, offered the *N-Trim* method. The new *Trace-Class Trim* method was commercially available until it superseded in *Lasergene99* by the *Trace-Quality Trim* method that I developed (described in Chapter 8).

Chapter 5

Consensus-Calling Case Studies

The second problem I addressed with the incorporation of *Trace-Class* scores is consensus calling (Allex, Shavlik, & Blattner 1999, Allex *et al.* 1997). For these case studies I developed both an algorithmic and a neural network solution. The algorithmic solution is described in Chapter 6 and the neural network approach is covered in Chapter 7. In the current chapter I define the consensus calling problem and describe the development of the data sets used in the studies.

Accuracy in consensus sequences is an important concern – *The National Human Genome Research Institute* (*NHGRI*) set a standard for sequencing accuracy of 99.99% (NHGRI 1998). Unfortunately, the error rate for sequences in GenBank has been estimated to be from 0.3 to 0.03% (Lawrence & Solovyev 1994) – much higher than the standard. When imperfect DNA sequences are translated, the effect on the resulting protein sequence can be substantial. Even the mutation of a single base can cause critical changes in the character of a predicted protein. Furthermore, the deletion or insertion of bases can result in incorrect translation into protein and the failure to recognize regions that code for genes.

Currently, sequencing accuracy is significantly dependent upon careful human examination and editing of consensus sequences in fragment assemblies. The hand process is time-consuming, expensive, and error-prone, making it unsuitable for large-scale sequencing projects. Automatic methods that produce highly accurate consensus calls reduce errors and alleviate the need for manual editing.

5.1 Existing Method

A common, simple method to calculate the consensus counts the number of calls of each base in an aligned column (Staden 1982a). If the majority base count is above a given fractional threshold of the total count, that base is called unambiguously (*A*, *C*, *G*, or *T*); otherwise the consensus is called as the appropriate ambiguity (combination of *A*, *C*, *G*, and/or *T*). We refer to this method as *Majority*. Figure 5-1 contains an example of calculating the consensus by *Majority*.

Since the *Majority* approach examines only the base calls and not the underlying trace data, it is prone to errors. There is no distinction between base calls made with well-defined peaks and those made with indefinite peaks. *Majority* also requires a minimum number of sequences to make an unambiguous call when a column of base calls is not in total agreement. Methods that directly analyze the trace data help to avoid these problems.

```
Consensus ... A A G A A G C A C T W A G G A T T T G G T ...
          ------------------------------------------------
          ... A A G A G G C A C T A A G G A T T T G G T ...
Aligned   ... A A G A A G C A C T T A G G A T T T G G T ...
Reads     ... A A G A A G C A C T A A G G A T T T G G T ...
          ... A A G A A G C A C T T A G G A T T T G G T ...
```

Figure 5-1. *Majority* Consensus Calls. Four sequences are aligned and the consensus is computed using *Majority*. In this example, the threshold is set at 75%. The consensus call for left shaded column is an *A* since three of four (at least 75%) of the calls are *A*. In the middle shaded column, 50% of the calls are A and 50% are *T*; the call is *W* (*A* or *T*) since both percentages are below the threshold. In the right shaded column, all calls are *T*, resulting in a consensus call of *T*.

5.2 Test Data Sets

I test the effectiveness of consensus calling methods by comparing their accuracies with different distinct amounts of *coverage* (number of aligned sequences). Since almost any reasonable algorithm can make correct calls when the coverage is high, I believe that one criterion that can be used to identify a superior method is its accuracy even when the coverage is low. In addition, since every step required to sequence a fragment adds to the overall expense of sequencing, reducing the needed coverage means a decrease in sequencing costs. In large sequencing projects, it is typical to produce a coverage of six to ten to ensure accurate results (Li *et al.* 1997) This much coverage is not needed when using a method that is highly accurate with fewer aligned sequences.

Fragment assemblies for a 124 kb section of *E. coli* are used to compare consensus calling methods. The data for the assemblies are supplied by the *E. coli* Genome Project at the University of Wisconsin (Blattner *et al.* 1997). Correct consensus calls are taken from *E. coli* sequences submitted to GenBank. The original assembly of 2221 ABI sequences ranges in coverage from 1 to 45 sequences. The assemblies were created with DNASTAR Inc.'s *SeqManII* fragment-assembly program. Although most of the data and alignments in the assemblies are quite good, sequence traces do vary in quality and some areas present more of a challenge for consensus calling. Figure 5-2 contains an example of an aligned region in one of the test assemblies that contains a fair amount of discrepancies, indicating imperfect underlying trace data and difficulties for consensus calling.

```
GCAANTAAAAANTGTTCCTTTGGGGTGAANANCCAAANATN-CCCNGCTGGGT
GCAATGAAATACTGTGCGT--GGGGTGAG-AGGCGAACATT-CCCGGCTGG--
GCAATGAAATATTATGCGN--GGGGTGAGAGGGCGAACATTCCCCGGCTGG--
GCAATGAAATACTGTNCGTN-GGGNTAAA-AGGC-AANNNTCCCCGGNNGG--
    ??   ? ? ? ? ????   ? ? ????? ?  ?????   ? ??  ??
```

Figure 5-2. Test Assembly Alignment. The data used for testing is of varying quality. Displayed here is a region with four aligned sequences from one of the test assemblies. Columns whose base calls are not in total agreement are marked with a '?.' There is a fair amount of disagreement among the base calls, implying poorer-quality underlying trace data. Consensus calling in this region is more difficult than in areas with near-perfect data.

In order to generate an abundance of test cases with various amounts of coverage, I developed and applied a greedy minimization algorithm, *Minimize Coverage*, to the assembly. With *Minimize Coverage*, sequence fragments are removed from an assembly as long as the coverage for any single column does not fall below a specified coverage (unless the coverage is already below the threshold). The idea for the *Minimize Coverage* algorithm is simple. At each pass through the assembly, for each sequence I determine the lowest coverage, `low-coverage`, of any column in which the sequence occurs. I then remove the sequence with the highest `low-coverage`, provided that `low-coverage` is not at or below the threshold. If more than one sequence has the same `low-coverage`, the shorter one is removed. Passes over the assembly are repeated until no more sequences can be removed without violating the coverage threshold restriction. At completion, some columns will have more than the desired coverage (due to the restriction) and some less. The algorithm is summarized next.

Minimize Coverage **Algorithm**

Let S be the list of all n sequences, S_i, in the assembly.

$$S = \{S_1, \ldots, S_n\}$$

Let L be the list of all n sequences considered for removal. Each sequence, S_i is paired with its `low-coverage`, LC_{Si}.

$$L = \{(S_1, LC_{S1}), \ldots, (S_n, LC_{Sn})\}$$

While *not_empty(L)*

1. Remove from L sequences whose `low-coverage` is at or below the threshold.
2. Remove from S and L the shortest sequence with the highest `low-coverage`.
3. Update `low-coverage` values.

Figure 5-3 steps through an execution of the *Minimize Coverage* algorithm.

(a)

```
S1    GATCGGCTACATCTTACATCACCGTT
S2          CTACATCTTACATCACC
S3  CGGATCGGCTACATCTTACATCACCGTTGA
S4     ATCGGCTACATCTTAC
S5              ATCTTACATCACC
S6       CGGCTACATCTTACATCACCGT
```

(b)

Pass	***S***	***L***
0	$\{S_1, S_2, S_3, S_4, S_5, S_6\}$	$\{(S_1, 2), (S_2, 5), (S_3, 1), (S_4, 3), (S_5, 5), (S_6, 3)\}$
1	$\{S_1, S_2, S_3, S_4, S_6\}$	$\{(S_2, 4), (S_4, 3), (S_6, 3)\}$
2	$\{S_1, S_3, S_4, S_6\}$	$\{(S_4, 3), (S_6, 3)\}$
3	$\{S_1, S_3, S_6\}$	$\{(S_6, 3)\}$
4	$\{S_1, S_3\}$	$\{\ \}$

Figure 5-3. ***Minimize Coverage*** **Example.** (a) Six sequences, S_1 to S_6, are aligned in a fragment assembly. The sequences in bold, S_1 and S_3, provide the optimal minimization when the threshold is set to two. With these two sequences in the assembly, no column has fewer than two sequences (except those that already had fewer in the original assembly). In addition, neither sequence can be removed without causing coverage to fall below the minimum. (b) The algorithm to reduce coverage on the assembly completes after 4 passes. At the outset, all sequences are in S and L. The first pass removes S_5 from both lists since it is the shorter of two sequences with the highest `low-coverage` (5 sequences). Also, S_1 and S_3 are removed from L in the first pass since their `low-coverage` is at or below threshold – these sequences cannot be taken out. At the end of four passes, L is empty and the two desired sequences, S_1 and S_3, remain in the assembly.

I repeatedly applied the *Minimize Coverage* algorithm to the original assembly for the range of coverage thresholds from two to ten. This produced nine assemblies with differing coverages, each with an abundance of aligned columns whose coverage corresponded to its threshold. For testing, from each of the nine minimized assemblies, I extracted the statistics for consensus calling only for columns that corresponded to the coverage threshold. For example, for the assembly with a minimum coverage threshold of three, I compiled statistics only for those columns with a coverage of three sequences. The exception is that the statistics for the assembly with the desired coverage of ten include all columns with coverage of ten or greater (rather than just those with exactly ten) since results tend to remain constant with such high coverage. Table 5-1 lists the number of consensus calls used for each set of results.

Table 5-1. Consensus Calling Data Sets. For each coverage from two sequences to ten or more, the number of consensus calls included in test results is listed.

Coverage	Number of Consensus Calls
2	67,860
3	57,092
4	45,394
5	39,556
6	34,011
7	26,716
8	22,479
9	20,326
≥ 10	47,239

5.3 Summary

The simplest approach to *consensus calling* is to count the number of base calls of each type in an aligned column. This method is referred to as *Majority*. With this method, if the count of the most commonly occurring base is above a threshold fraction of the total calls, the consensus is called as a base. If the fraction is below the threshold, an ambiguity code (as in Table 2-1) is used to call the consensus.

One limitation is the *Majority* method is that it relies entirely upon the correctness of base calls. In the next two chapters I introduce highly accurate methods that look at the underlying trace data in determining the consensus. I confirm the effectiveness of the new methods by comparing their consensus accuracies for a range of *coverages* (number of aligned sequences). To accumulate an adequate amount of data for testing at each coverage, I developed and implemented a new algorithm, *Minimize Coverage*, that systematically reduces the predominant coverage in an assembly to a specified level. Repeatedly applying the technique to an assembly of *E. coli* data produced abundant data for testing at coverages from two to ten and over. Accuracy at low coverages is one criterion that can be used to evaluate consensus calling approaches; high accuracy at even low coverages identifies a superior technique. In addition, methods that are highly accurate at low coverages can reduce the cost of sequencing by lowering the required number of sequences in an assembly.

Chapter 6

Trace-Evidence Consensus

The first approach to the consensus calling problem that I investigated is an algorithmic method that directly incorporates ABI trace-data information via peak scores from the *Trace-Class* representation (Chapter 3). I refer to the new method as the *Trace-Evidence Consensus* method (Allex *et al.* 1997).

The *Trace-Evidence* method is based on the idea that each of the three peak *Trace-Class* scores supplies an amount of evidence that the associated base should be assigned in the consensus. High strong-peak (*SP*) scores supply the greatest amount of evidence, high medium-peak (*MP*) scores supply the next greatest amount of evidence, and high weak-peak (*WP*) scores provide the least. Figure 6-1 demonstrates the evidence idea.

6.1 Algorithmic Details

To determine the consensus for a column of aligned bases, I sum the evidence, *E*, based on *Trace-Class* scores for each of the four bases. The evidence for each base is multiplied by the weight values described in Section 3.2.2. The base with the highest evidence sum is identified as the `leader` and its evidence sum is the `leading-evidence`. The other three bases are `competitors`, and their evidence sums are `competing-evidence`. A threshold between 0 and 1 is specified that determines the ignorable fraction of `competing-evidence` to `leading-evidence`. If the `leader` has no `competitors` with `competing-`

`evidence` greater than the threshold, the `leader` is assigned as the consensus. If `competing-evidence` for any bases surpasses the threshold, then those bases are included in determining an ambiguous call.

To determine the consensus for a column of aligned bases, two types of values must be calculated for each sequence in the column: the *Trace-Class* scores and a measure of the weight (quality) of the data. I use the weight scores to apply appropriate emphasis to the evidence supplied by each set of classification scores. That way, more reliable, higher quality trace data supplies more evidence than trace data of lower quality.

When gaps occur in a column, the weight scores are also used to decide if the consensus should be called as a gap. To do this, I sum the weights for sequences with a gap in the column and compare them with the sum of the weights of sequences without a gap. If the gap weight sum exceeds the non-gap sum, the consensus is called as a gap.

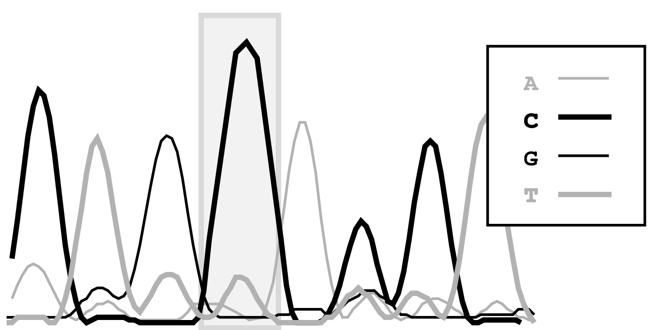


Figure 6-1. Evidence in Traces. Consider the evidence found in the four traces in the shaded region. The *C* trace will produce a high strong peak (*SP*) score, the *T* trace will yield a relatively smaller *SP* score, and both the *A* and *G* traces will produce peak scores of 0. A visual examination of the traces supports the premise that the vast majority of the evidence is for a base call of *C* and that there is essentially no evidence for a base call of *A* or *G*. (Actual data shown.)

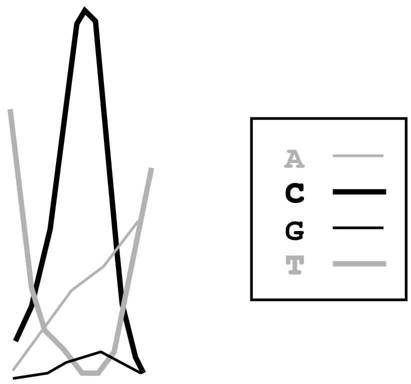
The steps used in the consensus calculation for a single aligned column appear next. Details of the calculations mentioned follow the algorithm.

Trace-Evidence Consensus **Algorithm**

For a single aligned column

1. For each sequence, find the quality of trace data, *W*, within a small window centered on the column.
2. Sum *W* for each sequence with a gap in the column and compare it to the sum of *W* for the remaining sequences. If the gap sum exceeds the non-gap sum, return *gap*.
3. Determine *S*, the 6x4 (six scores for each of four traces) matrix of *Trace-Class* scores for each sequence.
4. Reduce each *S* to a vector, *E*, of four values that summarize the evidence for each trace.
5. Multiply each value in *E* by its corresponding *W* to produce a vector *E′* that has been adjusted by data quality.
6. Sum each of the corresponding *E′*s to produce a vector, *T*, of the total evidence for each of the four bases.
7. Find the highest evidence (`leading-evidence`) in *T*; its corresponding base is the `leader`.
8. Multiply `leading-evidence` by the threshold to compute the maximum ignorable `competing-evidence`.
9. Compare `leading-evidence` to each `competing-evidence`. If no `competing-evidence` surpasses the maximum ignorable, then return `leader` as the consensus call, otherwise use all `competitors` who surpass the maximum to determine and return an ambiguity.

I use the same summary vector, V, used for weight calculations, and defined in Section 3.2.2, to summarize the *Trace-Class* scores during consensus computation. Multiplication by V ensures that scores supplying the most evidence (such as those with high SP scores) are given more credence than those that supply less evidence. Figure 6-2 demonstrates this idea.



	SP	***MP***	***WP***	***WV***	***MV***	***SV***	$S \bullet V$
A	0	4	30	8	1	0	13
C	89	11	0	0	0	0	96
G	3	1	0	0	0	0	4
T	0	0	0	0	13	76	0

Figure 6-2. Summarizing *Trace-Class* Scores. The *Trace-Class* scores, S, for each of the four traces are computed. When the dot product of S and V ([1 .67 .33 0 0 0]) is computed, the result is a high value for the C trace – the trace exhibiting highest evidence. The values for the A, G, and T traces are all low. When these summarized values are used to provide evidence, the C trace appropriately has the highest value. Note that in this calculation, the *Trace-Class* scores are computed for each of the four traces in contrast to the calculation of the weight measure in which only the scores for the trace associated with the called base are computed. Here, I need to know how much evidence each trace supplies.

For each sequence in a column, a vector, E, summarizes the evidence for each possible base (A, C, G, and T). For each base, the computed value reflects the amount of evidence that the call should be that base. The vector is computed as follows.

1. Form a 6x4 matrix of *Trace-Class* scores, S, by computing the scores for each trace:

$$S = \begin{bmatrix} SP_A & SP_C & SP_G & SP_T \\ MP_A & MP_C & MP_G & MP_T \\ WP_A & WP_C & WP_G & WP_T \\ WV_A & WV_C & WV_G & WV_T \\ MV_A & MV_C & MV_G & MV_T \\ SV_A & SV_C & SV_G & SV_T \end{bmatrix}$$

2. The matrix multiplication of V and S produces a vector of evidence values, E, for the possible bases:

$$E = \mathrm{V} \times S = [\, E_A \;\; E_C \;\; E_G \;\; E_T \,]$$

3. Multiply E by the quality of the local trace data, W, to produce evidence values, E', that have been adjusted by the quality of the data:

$$E' = E \times W$$

Finally, I sum the evidence for each base in an aligned column as described next.

Sum corresponding E' values to produce the total evidence, T_i, for each possible base i, where n is the number of sequences in the column:

$$\begin{aligned} T_A &= E_{A1}' + E_{A2}' + \ldots + E_{An}' \\ T_C &= E_{C1}' + E_{C2}' + \ldots + E_{Cn}' \\ T_G &= E_{G1}' + E_{G2}' + \ldots + E_{Gn}' \\ T_T &= E_{T1}' + E_{T2}' + \ldots + E_{Tn}' \end{aligned}$$

Once *T* has been calculated, consensus calling can be completed as described in steps 7-9 of the *Trace-Evidence Consensus* algorithm. An example determination of a consensus base call appears in Figure 6-3.

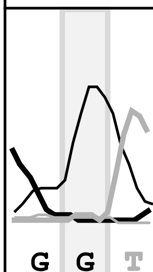
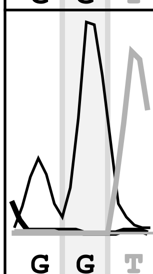
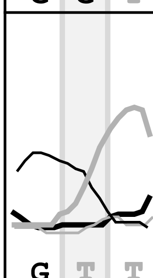
Seq	***W***	***E***				***W*** x ***E***			
		A	***C***	***G***	***T***	***A***	***C***	***G***	***T***
	.51	0	0	52	0	0	0	26.5	0
	.82	0	0	73	0	0	0	59.9	0
	.26	0	0	21	5	0	0	5.4	1.3
					Total	0	0	91.8	1.3

Figure 6-3. ***Trace-Evidence Consensus* Example.** The consensus base for the center column of three aligned sequences must be called. For each sequence, the evidence, *E*, for each base is multiplied by the corresponding weight, *W*, When these products are summed for the three sequences, the evidence for *A* and *C* is 0, for *G* is 91.8, and for *T* is 1.3. If the threshold is .50, *G* will be called unambiguously since no `competing-evidence` surpasses 45.9 (91.8 x .50). In contrast, the *Majority* method with a 75% threshold would make an ambiguous call of *K* (*T* or *G*).

6.2 Evaluation

All code for testing the new consensus calling method was incorporated into an experimental version of the DNASTAR Inc.'s *SeqMan* fragment assembly program for the Apple *Macintosh PowerPC*. *SeqMan* uses the *Majority* consensus calling method. (*SeqMan* has since been superseded by *SeqManII*, a more powerful, commercially available, version that incorporates trace analysis as described in this dissertation.)

Fragment assemblies as described in Chapter 5 are used to compare correct calls to *Majority* and *Trace-Evidence* calls. I report results for consensus calls made with coverages from two to ten or more aligned sequences. The threshold is set to the *SeqMan* default value of 75% for *Majority* and to 50% for *Trace-Evidence*. Graphs in Figure 6-4 display the number of correct calls, incorrect calls, and ambiguous calls per kb for the two methods.

The results show a significant improvement with the *Trace-Evidence* method, especially at lower coverages (number of aligned sequences). Differences are statistically significant using a paired one-tailed t-test at the 95% confidence level. With a coverage of only three, using *Trace-Evidence*, I see a leveling of the number of incorrect calls and a large improvement over the *Majority* method in the number of correct and ambiguous calls. With a coverage of four, the number of ambiguous calls has fallen to nominal values with *Trace-Evidence*.

6.3 Discussion

I observe striking examples of the utility of the *Trace-Evidence* method when base calls in a column are systematically incorrect. In some instances, a well-defined peak is hidden below a high-intensity valley. The base is often incorrectly called as the one associated with the high-intensity valley. *Majority* methods incorrectly call the consensus as this base. My new *Trace-Evidence* makes the correct consensus call even when all or most of the bases have been called incorrectly. Figure 6-5 contains an actual example of this occurrence.

I have identified three situations in which *Trace-Evidence* can make incorrect calls. Overwhelmingly, most problems involve gaps. In rarer cases I have difficulties with low evidence sums or poor-quality data. Next, I briefly describe these three sources of incorrect calls.

In the results reported here, all of the incorrect calls at coverages above three and at least half of those for coverages of two or three involve gaps in the column. The method for determining whether a gap should be inserted in the consensus consists of a simple comparison of gap versus non-gap sums of the weights of the traces in the column. However, the insertion of a gap affects not only the column in which it occurs, but also the columns to either side. When determining a gap call, it is probably necessary to consider more context and examine the data on either side of the base of interest. Finding a solution to calling the consensus when gaps are in the alignment would virtually eliminate incorrect calls made with the *Trace-Evidence* method with a coverage of at least four.

In some instances, incorrect calls can be associated with extremely low evidence sums. When the sums are quite low, even the maximum evidence is often not indicative of the correct call. One solution is to label the consensus as an ambiguous *N* and defer consensus determination to human editors. For the results reported in this chapter, this is the solution used (i.e. low-evidence calls are counted in the ambiguous category). To circumvent the low-evidence problem in the commercial version of *SeqManII*, consensus calling reverts to *Majority* when the maximum evidence is less than ten. (This number was chosen as one that works well in practice.)

A few incorrect calls occur in cases that are difficult for both *Majority* and *Trace-Evidence*. These are usually in regions of poorer-quality trace data where peaks are overlapping and ill-defined. The obstacle for *Majority* is that one or more of the base calls is likely to be incorrect in such regions. For *Trace-Evidence* the difficulty lies in the relative locations of the trace peaks. Often the peak associated with the correct base call is significantly offset from the base-call location. The result is that when the *Trace-Class* scores are computed, a peak is either not detected or is given a low score due to its distance from the base-call location. Another of the traces may exhibit a small, distinct peak near the base-call location that is scored relatively higher. *Trace-Evidence* then has more evidence associated with the small peak than with the correct trace and calls the consensus incorrectly. This case is illustrated in Figure 6-6.

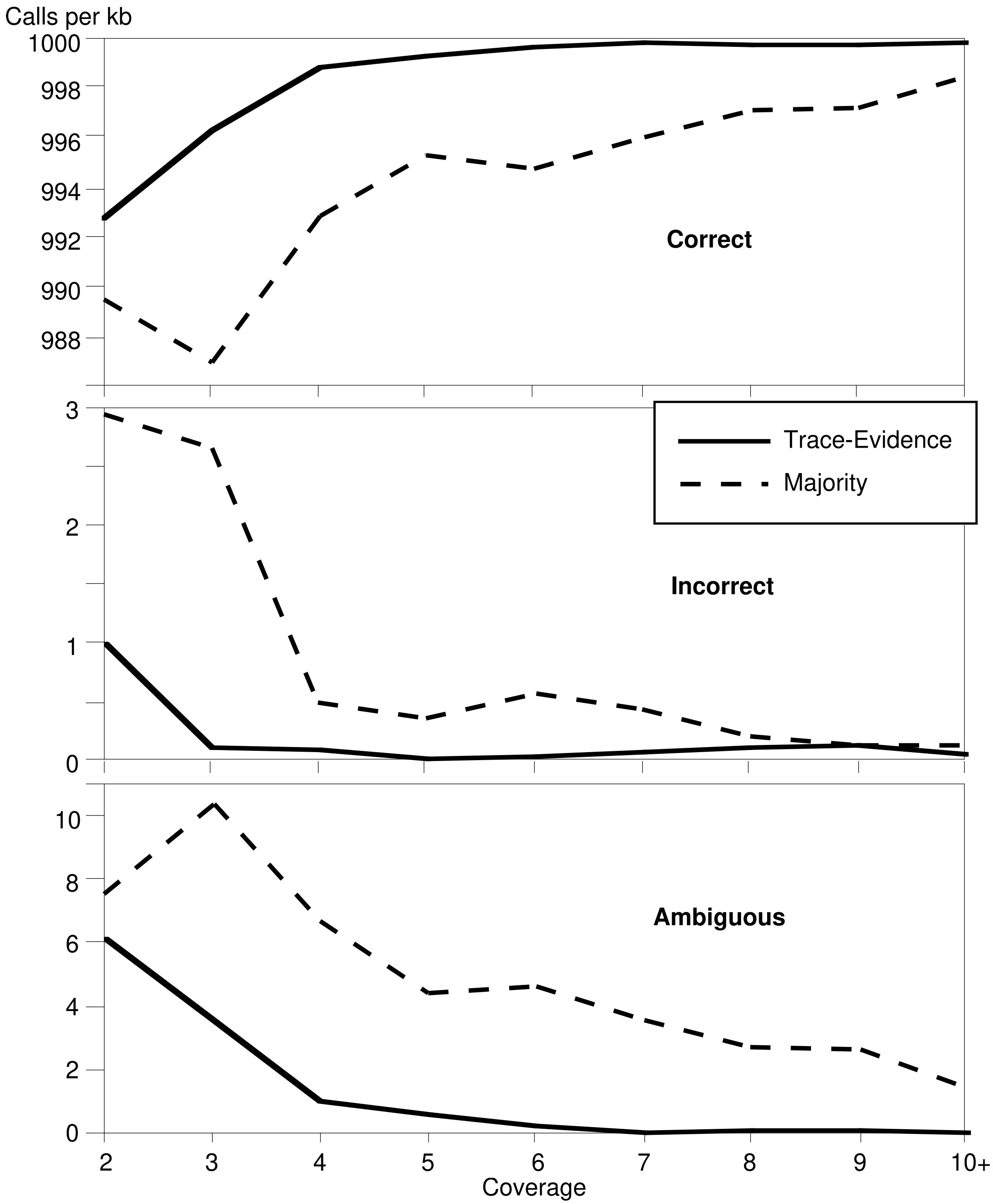


Figure 6-4. *Trace-Evidence* Test Results. Accuracy results by amount of coverage are graphed. Each data point is based on 20 - 68 kb consensus calls (Table 5-1). The new *Trace-Evidence* method produces more correct calls and fewer incorrect and ambiguous calls.

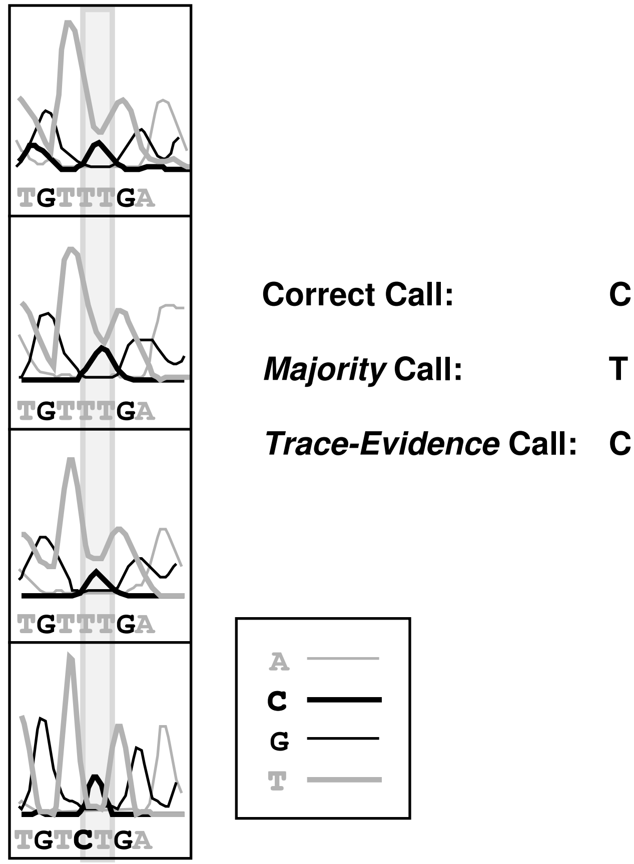


Figure 6-5. *Trace-Evidence* versus *Majority* Consensus. In the shaded column, three bases have been incorrectly called as a *T* and one correctly as a *C*. With a 75% threshold, the *Majority* method incorrectly computes the consensus as a *T*. The *Trace-Evidence* method detects no evidence for a *T*, ample evidence for a *C*, and calls the correct consensus. With *Majority* this situation would be even more troublesome if the fourth sequence were not in the assembly. In that case, the call would have no conflicting base calls and would likely go unquestioned during manual editing. In contrast, *Trace-Evidence* correctly computes a *C*, even in the absence of the fourth sequence. (Actual data shown.)

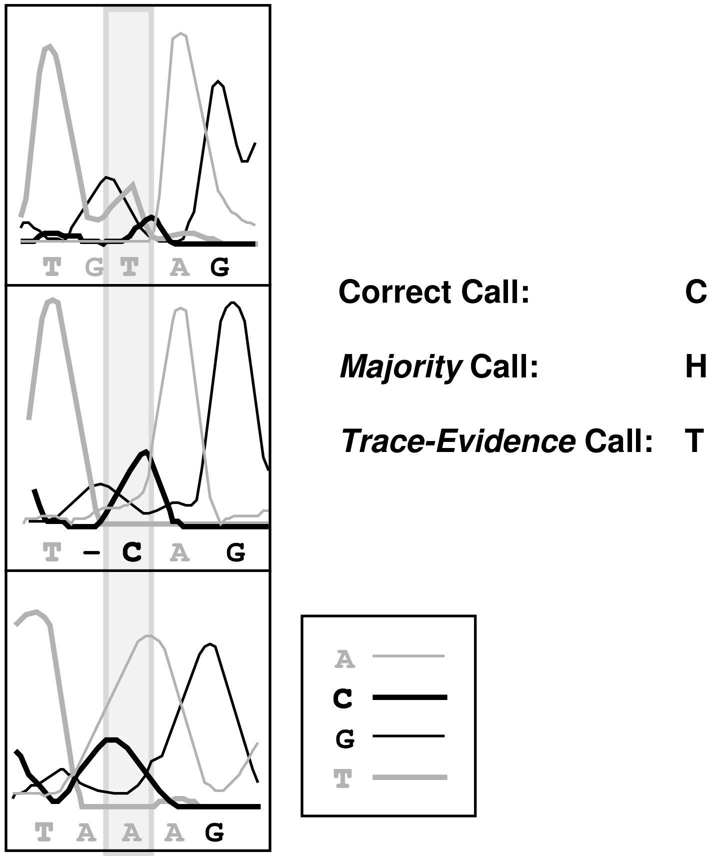


Figure 6-6. Difficult Consensus Call. Three sequences have been aligned; the correct call for the shaded column is *C*. *Majority* calls an ambiguous *H* for the consensus since the column includes conflicting base calls of *T*, *C*, and *A*. The *Trace-Evidence* method assigns negligible strong peak scores to the offset peaks associated with the *C* traces and a high strong peak score for the *T* trace in the first sequence. The scores incorrectly sum to adequate evidence for a *T* and insufficient evidence for *C*. (Actual data shown.)

6.4 Summary

The overall goal of my work is to improve the quality and efficiency of automatic fragment assemblies. Toward this goal, I have developed a new method for consensus calling, *Trace-Evidence*, that produces increased consensus accuracy, thereby reducing manual editing and decreasing the amount of coverage needed. Using the *Trace-Evidence* method results in automatically produced consensus sequences that are more accurate and less ambiguous than those produced with standard a majority-voting method. Additionally, these improvements are achieved with less coverage than required by the standard methods – using *Trace-Evidence* and a coverage of only three, error rates are as low as those with a coverage of over ten sequences. I accomplished this by direct incorporation of trace information into automatic consensus calling via the *Trace-Class* representation of trace data. In contrast to my new method, less accurate methods use only a limited representation of trace data – base calls – to determine the consensus.

I implemented the *Trace-Evidence* method for consensus calling in the commercially available version of DNASTAR Inc.'s *SeqManII* fragment assembly program. The previous version, *SeqMan*, used the *Majority* method to make consensus calls. In the latest version of *SeqManII*, available as part of the Lasergene99 suite of applications, *Trace-Evidence* has been updated to the *Trace-EvidenceII* method described in Chapter 8.

Chapter 7

Neural-Network Consensus

The second approach to computing a consensus that I investigated uses neural networks to process trace data (Allex, Shavlik, & Blattner 1999). Given inputs extracted from an aligned column of DNA bases and the underlying Perkin-Elmer Applied Biosystems (ABI) fluorescent traces, my goal is to train a neural network to correctly determine the consensus base for the column. Choosing an appropriate network input representation is critical to success in this task (Baldi & Brunak 1998, Craven & Shavlik 1993). I empirically compare five representations; one uses only base calls and the others include trace information.

One significant way that my system for consensus calling differs from most existing methods is that it directly processes information on the shape and intensity of ABI fluorescent traces. Other methods, such as those in the *TIGR Assembler* (Sutton *et al.* 1995), and *GAP* (Bonfield *et al.* 1995), examine only previously determined base calls when calculating the consensus.

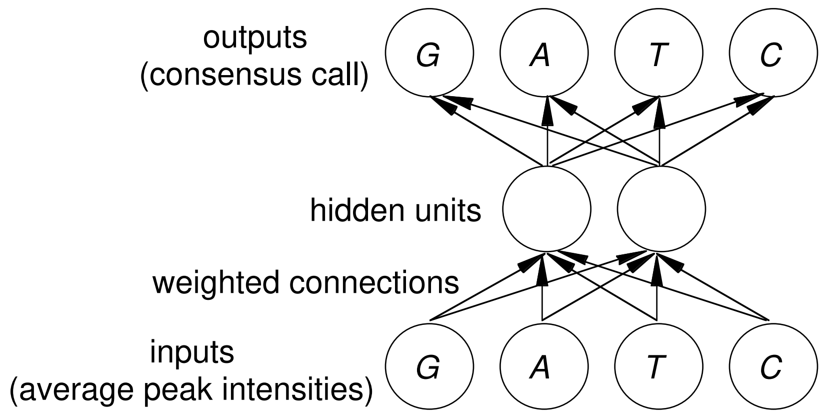
Two existing assemblers that do consider trace characteristics are *Phrap* (Green 1997b, *Phrap* source code documentation) and DNASTAR Inc.'s *SeqManII.* To make a consensus call, *Phrap* chooses the base call in an aligned column with the highest-quality trace as determined by its companion base-calling program, *Phred* (Ewing *et al.* 1998, Ewing & Green 1998). The method used in *SeqManII* is described in Chapters 6 and 8. It extracts and sums information about the shape and intensity of the traces in an alignment. The sums are used as evidence in determining the most likely consensus call.

Another difference between my system and others is the use of neural networks. Since their introduction in the 1940s (McCulloch & Pitts 1943), artificial neural networks have emerged from the realm of pure academic research into practical solutions for a plenitude of problems (Widrow, Rumelhart & Lehr 1994). In recent years, interest in developing neural network solutions for problems in molecular biology has surged. A sampling includes:

- protein-structure prediction (Rost & Sander 1993, Stolorz, Lapedes & Xia 1992, Qian & Sejnowski 1988),
- DNA base calling (Tibbetts, Bowling & Golden 1994),
- finding protein binding sites (Heumann, Lapedes & Stormo 1994),
- detection of protein-coding regions (Craven & Shavlik 1993, Snyder & Stormo 1993, Uberbacher & Mural 1991, Noordewier, Towell & Shavlik 1991), and
- identifying RNA polymerase binding sites (Pedersen & Engelbrecht 1995, Towell, Shavlik & Noordewier 1990).

Neural networks often provide a good solution to biological problems such as these since the problems involve intricate interactions, and the strength of neural networks lies in their ability to learn to recognize complex patterns. Given their success in the computational research community, neural networks have the potential to be a powerful tool for data analysis in biological research labs. Despite this, the use of neural networks for tasks in DNA sequencing has been scarcely explored. In one promising example, neural networks are used to make base calls in individual DNA sequences (Golden, Torgersen, and Tibbetts 1993). Note that Golden's work calls bases in *single* sequences whereas the work I describe determines the consensus for multiple aligned sequences.

Figure 7-1 contains a brief description of the operation of neural networks; details can be found in McClelland and Rumelhart (1986).



Inputs: Average relative *G*, *A*, *T*, and *C* trace peak intensities
Outputs: A consensus call for the aligned column

Categorized Examples

	Inputs				Desired Outputs			
example 1:	.32	.01	0	.03	1	0	0	0
example 2:	.05	0	.01	.35	0	0	0	1
...								
example *n*:	.38	.01	.04	0	1	0	0	0

Figure 7-1. Neural Networks. A *feed-forward backpropagation neural network* learns to categorize patterns of *inputs*. *Inputs* are numerical representations of features of a problem. Typically, there is one *output* for each category of the problem; the *desired output* is 1 for the correct category and is 0 otherwise. First the network is *trained* by processing a set of *categorized examples* (a *training set)*. A *categorized example* is an instance of the problem that includes its inputs and desired outputs. During training, weighted connections in the network are adjusted so that the error in the actual output is reduced. Hidden units in the network aid by allowing the input representation to be transformed. When the difference between the desired and actual inputs is sufficiently low, training is halted and the network can be used to categorize previously unseen instances of the problem. Future accuracy of the trained network is estimated by measuring a trained network's performance on a disjoint set of *testing examples.*

In this figure, I have an example of a simple neural network whose function is to call the consensus for a single aligned column of DNA bases when given inputs extracted from fluorescent traces. The network is given four inputs (the relative *G*, *A*, *T*, and *C* trace intensity averages), and outputs a consensus call (*G*, *A*, *T*, or *C*).

7.1 Algorithmic Details

The ability of a neural network to correctly categorize instances of a problem is critically dependent upon the input representation (Baldi & Brunak 1998, Craven & Shavlik 1993). For my work, this problem can be expressed as follows.

Given: An aligned column of base calls and traces
Do: Represent the column as numerical inputs

I define four features of an aligned column that can be used singly or in combination to form input representations for a neural network. Two of the features use information extracted from fluorescent traces. I believe that much valuable information is lost when the traces are reduced to base calls. My hypothesis is that a neural network can exploit the trace information to make consensus calls that are more accurate than those made with networks that use only base calls as inputs.

The inputs that use trace information are multiplied by the weight (quality) of the trace so that more emphasis is given to better data. A description of the calculation of the weight values I use appears in Chapter 3. One of the input features that uses fluorescent trace information captures the shape of the traces. To do this, I employ *Trace-Class* scores as described in Chapter 3.

The four input features I defined for an aligned column are listed next.

- *Base Call Fraction*
  The fraction of occurrences of *G*, *A*, *T*, and *C*.
- *Gap Fraction*
  The fraction of occurrences of gaps.
- *Trace Peak Intensities*
  For each base, the trace peak intensity multiplied by its weight and averaged over the number of aligned sequences.
- *Trace Peak Shapes*
  For each base, the strong (*S*) and medium (*M*) *Trace-Class* peak scores multiplied by its weight and averaged over the number of aligned sequences.

Figures 7-2 to 7-5 contain the details of calculating the numerical inputs for these features.

```
... C G A A G T A A T A ...
... C G A A C T A A T A ...
... C G A A C T A A T A ...
... C G A A - T A A T A ...
```

4 Inputs: 0.25 0 0 0.5

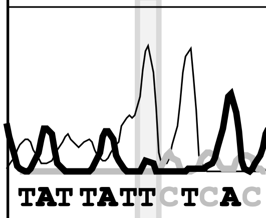
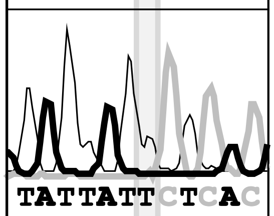
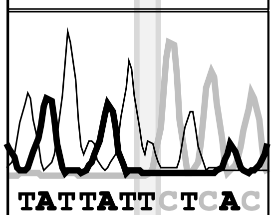
Figure 7-2. ***Base Call Fraction.*** There are four aligned sequences in the shaded column in this example. For each of the four bases I divide the number of its occurrences by the number of sequences. The *G* base call occurs once in four sequences, so its input is set to 0.25. Likewise, the inputs for *A*, *T*, and *C* are 0, 0, and 0.5 (2 of 4), respectively.

```
... C G A A G T A A T A ...
... C G A A C T A A T A ...
... C G A A C T A A T A ...
... C G A A - T A A T A ...
```

1 Input: 0.25

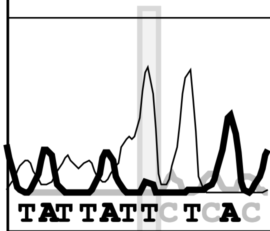
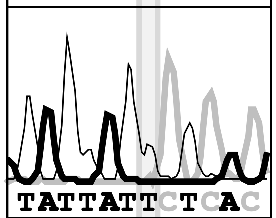
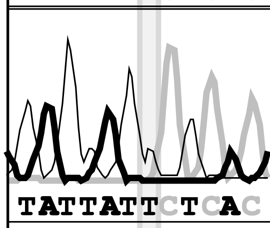
Figure 7-3. ***Gap Fraction.*** For this example, I again have four aligned sequences in the shaded column. For this input, I am only interested in gaps, so the single input is the number of gap occurrences divided by the number of sequences. Since a gap occurs once in the four sequences, the input is 0.25.

Maximum intensity = 1600

Aligned column	Weight	Peak Intensity (relative to maximum) G	A	T	C
	0.37	0 (0)	0.11 (0.04)	0.69 (0.26)	0 (0)
	0.42	0 (0)	0 (0)	0.18 (0.08)	0 (0)
	0.40	0 (0)	0 (0)	0.15 (0.06)	0.01 (0)
Weighted Average		0	0.01	0.13	0

4 Inputs: 0 0.01 0.13 0

Figure 7-4. ***Trace Peak Intensities.*** Three sequences are aligned in the shaded column. For each of the four bases in each sequence, the intensity (value at the center of the column) of the trace is divided by the maximum possible trace value. This fraction is then multiplied by the weight assigned to the base. The average over the weighted values forms the input for each base. In this example, the maximum trace value is 1600 (a typical value for ABI traces). In the first sequence, the intensity of the *T* trace is 1104 and its intensity relative to the maximum is 0.69 (1104/1600). Values for all other bases in each sequence are calculated in the same way. The values are then multiplied by their corresponding weights and the results are given in parentheses below each relative intensity. When averaged, the values yield the inputs 0, 0.01, 0.13, and 0. (When averaged over three sequences, the 0.01 sum for *C* rounds to 0.)

Aligned column	Weight	*Trace-Class* Scores *G* *S*	*M*	*A* *S*	*M*	*T* *S*	*M*	*C* *S*	*M*
	0.37	0 (0)	0 (0)	0.04 (0.01)	0.03 (0.01)	0.28 (0.10)	0.22 (0.08)	0 (0)	0 (0)
	0.42	0 (0)	0 (0)	0 (0)	0 (0)	0.25 (0.11)	0.20 (0.08)	0 (0)	0 (0)
	0.40	0 (0)	0 (0)	0 (0)	0 (0)	0.15 (0.06)	0.10 (0.04)	0 (0)	0 (0)
Weighted Average		0	0	0	0	0.09	0.07	0	0

8 Inputs:							
0	0	0	0	0.09	0.07	0	0

Figure 7-5. ***Trace Peak Shapes.*** To form the inputs for the three aligned sequences in the shaded column, I extract trace information using *Trace-Class* scores. I first compute the strong (S) and medium (M) peak scores for each of the four traces in each sequence. (I found weak scores to be irrelevant and do not use them.) Each score is then multiplied by the weight for its base. The weighted scores are given in parentheses below the scores. There are two inputs for each base: the average over all the sequences of the weighted strong scores and the average of the weighted medium scores. (When averaged over three sequences, the 0.01 sum for A-S rounds to 0.)

I tested five network topologies. Each has five hidden units and five outputs. The desired outputs for the networks always consist of four *0*s and a single *1* that represents either one of the four bases or a gap. The input representations use combinations of the four possible input features described above. The simplest network, referred to as *Base Call*, uses an input representation that consists of the *Base Call Fraction* and the *Gap Fraction* features. The *Base Call* network is used as the control in testing my hypothesis that inputs that include trace information produce more accurate results than those that only consider base calls.

A second network, called *Trace Shape*, uses nine inputs that include the *Trace Peak Shapes* and *Gap Fraction* input features. A third network, *Trace Intensity*, has five inputs that use *Trace Peak Intensities* and *Gap Fraction* input features. The fourth network, referred to as *Trace Shape and Intensity*, uses both the *Trace Peak Intensities* and the *Trace Peak Shapes* as well as the *Gap Fraction* features in its thirteen inputs. Finally, I tested one network that included all the possible input features: *Base Call, Trace Peak Intensities*, *Trace Peak Shapes*, and *Gap Fraction*.

The five network topologies are summarized in Table 7-1. To make a consensus call with one of these networks, I find the highest output value and its corresponding base or gap is the consensus call. Ambiguous calls may also be made by setting a threshold; if more than one output exceeds the threshold, then the appropriate ambiguous call is made. If only one output is above threshold, the call is unambiguous.

Table 7-1. Neural Network Topologies. Each of the five networks has five hidden units and five outputs. The number of inputs range from 5 to 17.

Neural Network Name	Number of Inputs	Input Features
Base Call	5	• *Base Call Fraction* • *Gap Fraction*
Trace Shape	9	• *Trace Peak Shapes* • *Gap Fraction*
Trace Intensity	5	• *Trace Peak Intensities* • *Gap Fraction*
Trace Shape and Intensity	13	• *Trace Peak Shapes* • *Trace Peak Intensities* • *Gap Fraction*
All	17	• *Base Call Fraction* • *Trace Peak Shapes* • *Trace Peak Intensities* • *Gap Fraction*

7.2 Evaluation

I used the assemblies produced with the *Minimize Coverage* method as described in Section 5.2 to construct training and test sets to analyze the effectiveness of the networks. I created example sets in which all of the examples for a particular set have the same *coverage* (number of aligned sequences). I chose examples with coverages of two, three, four, five, and six to form five sets. Each set contains 20,000 examples of categorized data. Ten training and test sets are constructed from each example set such that each network is trained on 18,000 examples and tested on the remaining 2000. Each example occurs in exactly one test set and nine training sets disjoint from the test set. In these sets, examples with a desired output of *gap* are far outnumbered by examples with desired outputs of G, A, T, or C. To enable the networks to learn to recognize gaps, *gap* examples are duplicated in the training sets so that they occur with about the same frequency as examples for each base. (Note that *gap* examples are not duplicated in test sets.)

NeuralWare Inc.'s *NeuralWorks Professional II* software was used for all neural network tests. I ran this software on an HP Pentium Pro 6/200 running *Windows NT*.

I trained and tested each of the neural network topologies with the five examples sets. For each coverage, I used 10-fold cross-validation and report accuracies averaged over the 10 test sets. During the training phase, each example in a training set was processed only once since accuracy fails to improve with more iterations.

Accuracy results for the five topologies are graphed in Figure 7-6. Of the five networks, I find that *Trace Shape & Intensity* produces the most accurate consensus calls. With a coverage of six, it makes only three errors in 20,000 calls. The range of accuracies is from 99.26% for a coverage of two to over 99.98% with a coverage of six.

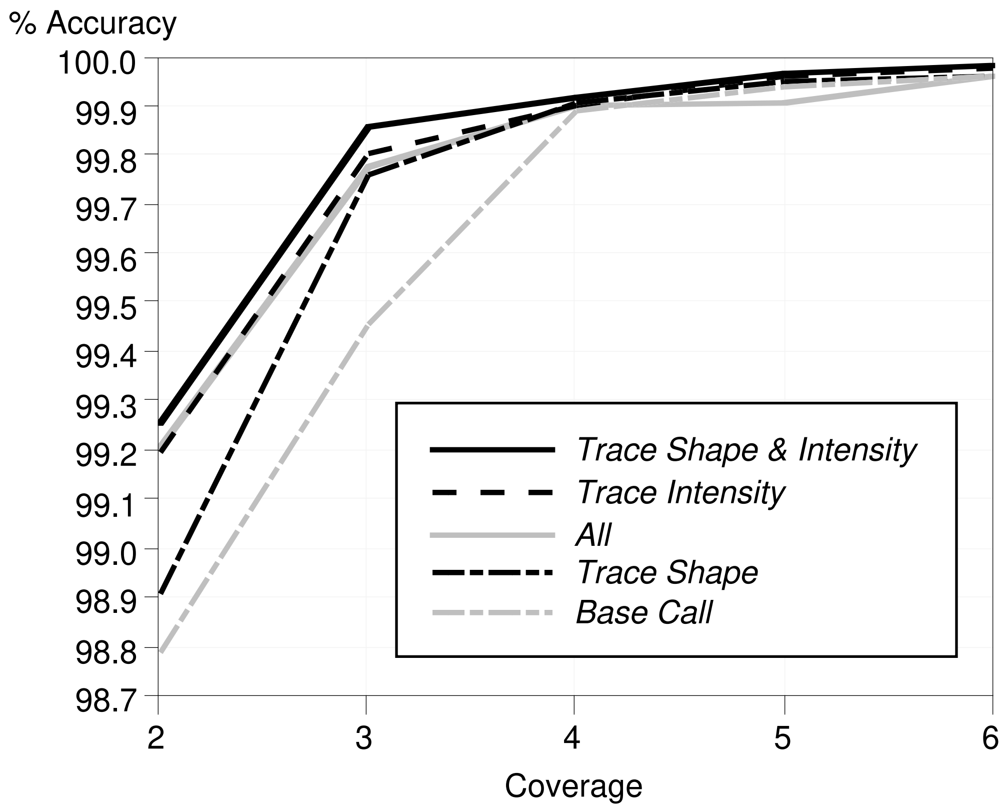


Figure 7-6. Neural Network Consensus Results. The *Trace Shape & Intensity* network produces the most accurate results at every coverage. With a coverage of four or more, the accuracies for all networks that use trace information are above 99.9%.

The network that uses only base-call information in inputs, *Base Call*, has the lowest accuracies at every coverage except five. At a coverage of five, the other network that incorporates base calls, *All*, has the lowest accuracy. With two or three aligned sequences, this network has substantially poorer results than any of the other four networks. Except when the coverage is four sequences, differences between the *Base Call* and the *Trace Shape & Intensity* networks are statistically significant using a paired one-tailed t-test at the 95% confidence level. As with the other networks, the best results using the *Base Call* network are achieved when the coverage is six. With six aligned sequences, the error rate is eight in 20 kb – more than double that of the best network that uses trace information.

I also compared the performance of the most accurate neural network, the *Trace Shape & Intensity* network, to the algorithmic *Trace-Evidence* method I developed. Here I report on results using the refined version of *Trace-Evidence*, *Trace-EvidenceII*, that is described in Chapter 8.2.1. Figure 7-7 graphs the accuracies of about 20 kb consensus calls output by the *Trace Shape & Intensity* network with the accuracies of about 680 kb consensus calls made by *Trace-EvidenceII*. The neural network data set is described in Chapter 5 and the data set used to test *Trace-EvidenceII* is described in Chapter 9.1. The *Trace-EvidenceII* consensus calls are more accurate than the neural network at all coverages.

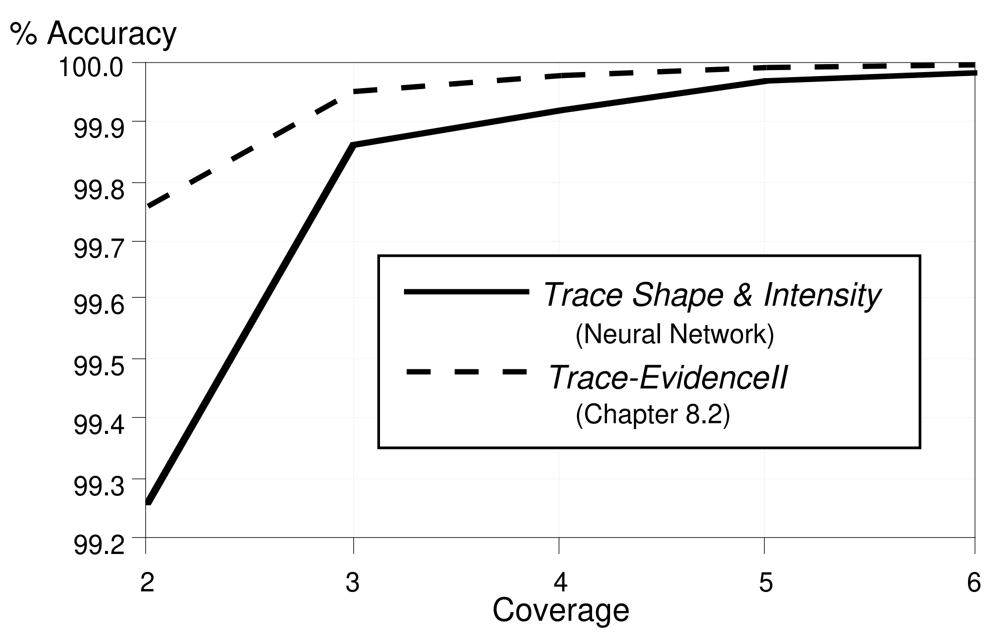


Figure 7-7. *Trace-Shape & Intensity* versus *Trace-EvidenceII* Accuracy.

7.3 Discussion

In additional (unreported) tests, I experimented with alternative plausible input representations. In one experiment, I extracted inputs from a broader context than a single column. My premise was that the accuracy of the consensus calls could be increased by extending the inputs to include trace information for one or more bases 5' to the base of interest. Parker *et al.* (1995) and Golden, Torgersen, and Tibbetts (1993) have reported that intensity values for a base are affected by 5' adjacent bases. For example, Parker *et al.* show that the intensity of a *C* peak following a *G* is relatively low. Several patterns such as these are described for fluorescent-dye labeled data (Perkin-Elmer 1995, Parker *et al.* 1995). I believed that the neural networks could be trained to recognize these patterns, but in practice found no improvement in accuracy with the extended inputs.

In another experiment, I provided not just a single intensity input for each trace, but rather the intensities in a window surrounding the center of the base peaks. These are the same values that I use in calculating *Trace-Class* scores, but rather than transforming them algorithmically, I allow the network to process them. The network using this alternate input representation required more inputs but yielded results very similar to the *Trace Shape & Intensity* network (results not reported).

7.4 Summary

Given inputs extracted from an aligned column of DNA bases and the underlying ABI fluorescent traces, I trained neural networks to determine the consensus base for the column. Choosing an effective input representation was the focus of this work. I compared five representations and found that networks trained with inputs incorporating fluorescent trace information are highly accurate. Based on estimates derived from using 10-fold cross-validation, the best network topology produces consensus accuracies ranging from 99.26% to over 99.98% for coverages from two to six aligned sequences. With a coverage of six, it makes only three errors in 20,000 consensus calls. In contrast, the network that only uses base calls in its input representation has over double that error rate – eight errors in 20,000 consensus calls. This represents a reduction in the need for manual editing. However, I find that when I compare the accuracies of the most accurate network, *Trace-Shape & Intensity*, with the best algorithmic consensus calling approach I have developed, *Trace-EvidenceII*, I find that the algorithmic method outperforms the network at every coverage.

Chapter 8

SLIC Fragment Assembly

In this chapter I introduce a new system for fragment assembly, the *SLIC Assembler*, that is another emphasis of my thesis research. First, recall from the discussion in Chapter 2, that to determine the sequence of bases in a genome or large segment of DNA, researchers first produce and sequence small, overlapping fragments of the genome. The base-call sequences of the small fragments are commonly referred to as *fragment reads, sequences*, or simply *reads*. The overlapping regions of the fragment reads are aligned into one or more *contigs* (contiguous segments) and the resulting layout is used to determine the consensus sequence of the genome. The process of determining the layout of the reads is called *fragment assembly* or *sequence assembly*. Assembly is complicated by *repeats*, subsequences that occur more than once in a genome. When a read contains a repeat, its placement in the layout is often ambiguous.

Techniques for sequencing whole genomes and other large fragments of DNA are constantly evolving. The *whole genome shotgun sequencing* strategy for bacteria involves creating a random, or *shotgun,* library of small fragments from the whole genome, then sequencing the small fragments. This strategy is successful for genomes up to several megabases in size. For larger genomes, a popular strategy involves first cloning large DNA fragments (about 200 kb long), then applying the shotgun sequencing strategy (e.g. Boysen, Simon & Hood 1997). To determine the sequence of the whole larger genome, assembled contigs must be arranged according to their positions on chromosomes. This strategy has not yet been successful in completing the sequence of a whole genome larger than that of a bacteria.

Although there are arguments for (Weber & Myers 1997) and against (Green 1997a) applying the whole genome shotgun sequencing approach to sequencing genomes as large as *Human*, it is clear that the approach will work with genomes considerably larger than those of a typical bacteria (Weber & Myers 1997). As strategies evolve toward sequencing larger fragments and even whole genomes, the amount of sequence data that will need to be assembled is immense. It is crucial that fragment-assembly software evolve in tandem to avoid bottlenecks in sequencing.

8.1 Existing Method

Several assembly software programs are commonly used to sequence large genomes. The program favored by many large genome centers is *Phrap*, developed at the University of Washington (Green 1997b, *Phrap* source code documentation). In this section, I will describe the operation of *Phrap*; in Chapter 11, I describe five other methods.

The method for assembly used in *Phrap* relies heavily on quality scores assigned to each base call by *Phrap*'s companion base calling program, *Phred* (Ewing & Green 1998). Each base call made by *Phred* is assigned a quality score that reflects the estimated probability of error of the base call. During assembly by *Phrap*, pairwise comparisons of the reads are used to adjust the quality scores. If a base call is confirmed by another call that was sequenced on the opposite DNA strand or with a different chemistry, the quality score is increased by summing it with the score of the confirming call. During assembly by *Phrap*, base calls with low quality scores are virtually ignored, allowing use of full fragment reads without trimming.

The adjusted Phrap quality scores are also used to identify putative repeats in fragment reads. If mismatches occur in alignments where the quality scores are high, they are assumed to be due to a near-repeat. On the other hand, mismatches that occur in alignments where quality scores are low are purported to be due to base-calling errors. Subsequently, the reads are not overlapped if a repeat is suspected. A companion program to *Phrap*, *RepeatMasker* has also been developed to help avoid false overlaps of repeated regions. RepeatMasker screens sequences for known repeats. When found, known repeated regions are masked from use during assembly.

To determine a consensus sequence, *Phrap* selects the base with the highest adjusted quality score in each aligned column. The consensus is then a *Mosaic* of the highest quality parts of the alignment. Using the *Mosaic* consensus method can have a significant impact on the depth of coverage (number of aligned sequences) needed for accurate sequencing. In their tests, the developers of *Phrap* find that in a typical data set, about 25% of the base calls have predicted error rates of less than one error in 10 kb. This may eliminate the need for filling in low coverage areas when the error probability is nominal.

As researchers move to sequencing larger fragments and whole genomes, assembly will require software that can efficiently handle large amounts of data. As with most assembly packages, the *Phrap* package uses pairwise comparisons of all fragment reads in assembling contigs. As a rule, the execution time for an algorithm that performs pairwise comparisons is n^2. With n^2 algorithms, as the size of assemblies increases, so does the rate of increase in the amount of time needed for execution.

Using n^2 time algorithms for large-scale sequencing projects is time-consuming, perhaps even becoming impossibly so. As an example, consider an assembly of 22,000 fragment reads for a 2 mb fragment that takes about four hours using an n^2 algorithm. Extrapolating based on n^2, it would take over one and a half years to assemble random shotgun fragments of a chromosome of the human genome. In contrast, consider using a method that runs in time proportional to n. If four hours are required to assemble 22,000 fragment reads, a chromosome could conceivably be assembled in less than ten days.

8.2 The *SLIC Assembler*

I describe a fragment layout algorithm that offers a substantial gain in speed by running in time that is, in practice, linear in the number of fragment reads, n. I call the algorithm *SLIC*, for *Sequence Layout into Contigs*. The layout algorithm has been incorporated into a total package, called the *SLIC Assembler*, that takes ABI trace files as input and produces gapped and aligned contigs with corresponding consensus sequences. Four major steps are required for this process. First the reads must be preprocessed to remove both low-quality data and *vector* sequence (a fragment used to carry and replicate a fragment of interest.) Second, the *SLIC* layout method establishes the approximate offset of each read in a contig. Then the reads are gapped and aligned to produce an assembly. Finally, a consensus sequence for each contig is determined. The steps are summarized here.

SLIC Assembly

1. Preprocess to trim poor-quality ends and remove vector sequence.
2. Determine the layout of fragment reads into contigs using *SLIC.*
3. Align the layout of reads in each contig.
4. Compute the consensus sequence for each contig.

8.2.1 Integral Ancillary Methods

Next I briefly explain the integral ancillary methods used in preprocessing, alignment, and consensus-calling steps (steps 1, 2, and 4). I follow with a detailed description of my new linear-time layout method, *SLIC*.

End-Trimming

The first major step, preprocessing to trim low-quality ends and vector sequence are completed with methods implemented in DNASTAR Inc.'s *SeqManII*. For use with the *SLIC Assembler*, I have improved the *Trace-Class Trim* algorithm developed in earlier work as discussed in Chapter 4. The new algorithm is called *Trace-Quality Trim*.

As with the earlier *Trace-Class Trim* method, the new *Trace-Quality Trim* algorithm evaluates ABI traces to trim low-quality data from the ends of sequences. It proceeds by first assigning a quality value, Q, to each base in a sequence. The Q score for a base reflects the confidence that the base has been called correctly. (Base calls of N always have a Q of 0.) The steps in assigning Q are listed next.

Assign Quality Score

Let $I_{base\text{-}call}$ be the individual quality for the trace associated with the base call.

Let I_{base} be one of three quality values for traces not associated with the base call.

1. Assign individual quality values, I, to each of the A, C, G, and T traces.
2. Find the maximum I_{base} :

   $I_{max} = max\,(I_{base1}\,, I_{base2}\,, I_{base3})$
3. Q is the difference between $I_{base\text{-}call}$ and I_{max}. (0 if negative):

   $Q = max\,(0,\, I_{base\text{-}call} - I_{max})$

The first step of the calculation is to assign an individual quality value, I, to each of the four traces associated with a base call. The values reflect the shape and intensity of the traces. For a simple example of how the I values for a base call are calculated, consider a tall, sharp trace peak whose intensity starts at 0, increases to the maximum, and then returns to 0. The value for this peak is 100. At the other extreme, a totally flat trace whose intensity remains at 0 is assigned a score of 0. A trace peak whose intensity begins at 0, increases half way to the maximum, and returns to 0 gets an I value of 50. Any value from 0 to 100 is possible given the variation in shape and intensity of traces.

The calculation of the I value is nearly identical to the calculation of the evidence score used in consensus calling. Among several differences, the main one is that the I values are relative to the maximum height of a peak in the entire sequence, whereas the evidence scores are relative only to the heights of the peak for that base call.

After the four individual quality values, I, are assigned, the highest I value of the three traces not associated with the base call is identified and subtracted from the value for the base call trace. For example, consider the case in which the base call is A and I values for C, A, G, and T are 2, 56, 0, and 10, respectively. Excluding the I value for A, the highest value is 10. The resulting quality score, Q, score for the base call is 46 (56 - 10). Scores may range from 0 to 100. (In the case that the difference is negative, the score is 0.)

For each base call an averaged Q value, A, is also assigned. This is simply the average of the Q scores over a window of 21 bases. The averaging serves to smooth transitions between the values and provide a measure of the general quality of the trace in a region. The idea is the same as the one used to assign weights described in Chapter 3.

The averaged Q values, A, are used for the new *Trace-Quality Trim* algorithm. To use *Trace-Quality Trim*, a user specifies a trimming stringency threshold. Recommended thresholds are in the range of 8 to 16. In general, the largest contiguous section of the sequence with at or above threshold A scores is retained and the ends on either side are trimmed. *Trace-Quality Trim* pseudocode appears in Appendix C and an example of using A quality scores for trimming is contained in Figure 8-1.

I have implemented the *Trace-Quality Trim* method for end-trimming in the most recent commercially available version of the *SeqManII* fragment assembler, a part of the *Lasergene99* suite of applications developed by DNASTAR Inc.

Alignment

The second major step in the *SLIC Assembler* is to layout reads into contigs. This step is the major emphasis of this work and I will describe the *SLIC* algorithm later in Section 8.2.2 and in detailed pseudocode in Appendix D. Once the *SLIC* method has been used to determine the overall layout of the contigs, the reads must be aligned and gapped before the consensus can be computed. Two existing methods that accomplish this task are compared.

One alignment method is implemented in *SeqManII*; its steps are summarized next.

Align with SeqManII

Let *n* be the number of sequence reads in a contig.
Let the *n* reads be ordered by the approximate offset determined by *SLIC*.
Create a new contig with sequence 1.
For reads 2 to *n*

1. Overlap the read with the contig at its approximate offset.
2. Compute the consensus of the contig.
3. Find exact substring matches between the read and the consensus.
4. Align exact matches whose order is consistent between the read and the consensus.
5. Gap and align between exact matches.

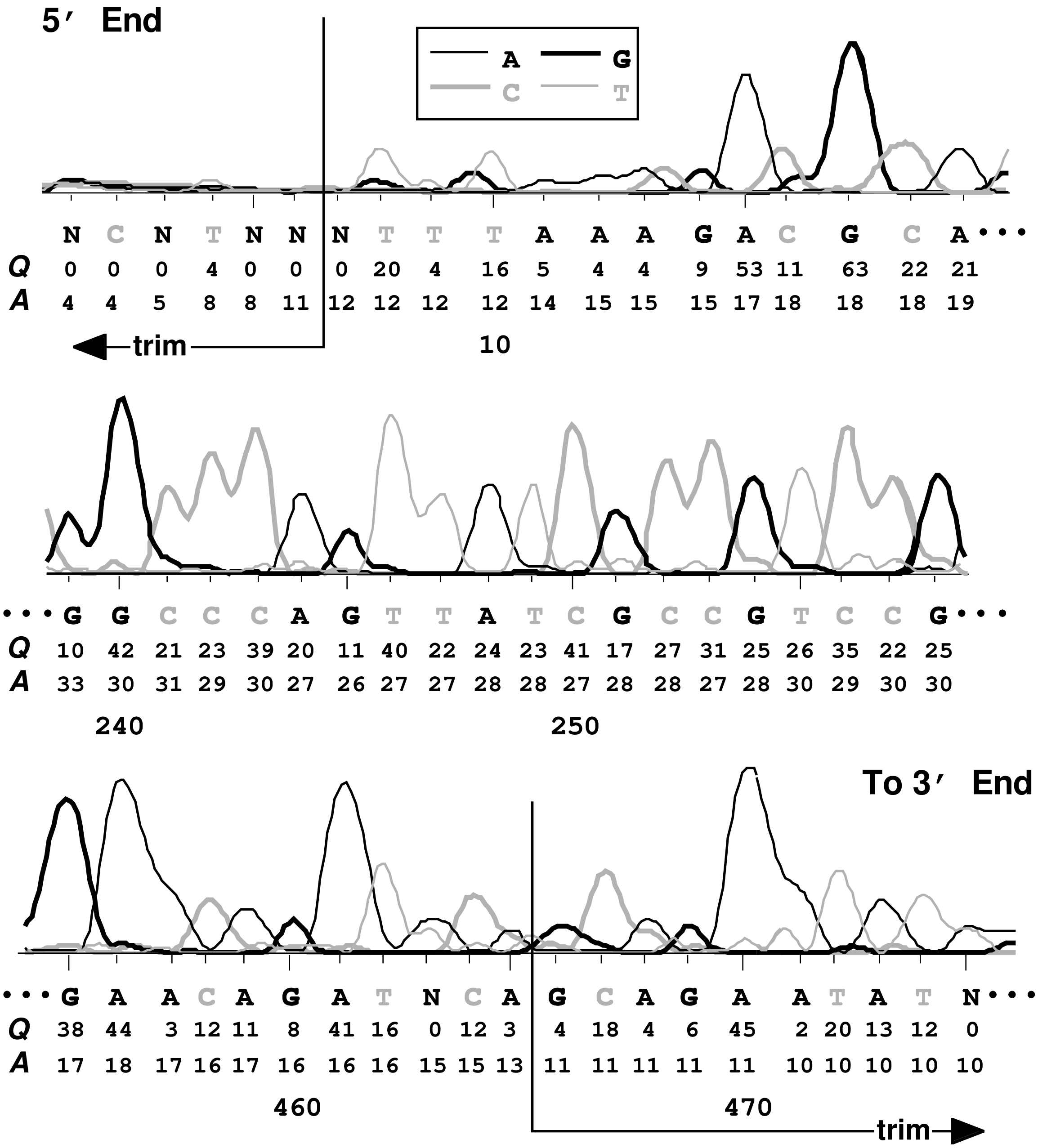


Figure 8-1. ***Trace-Quality Trim***. Beginning, middle and ending segments of a trace and base calls are shown. For each base, the quality score, *Q*, and average score, *A*, for a window of 21 bases are shown below the base. In this example, the trimming threshold is set to 12; the largest section of the read with an *A* score above 12 is retained and the ends are trimmed. The first six bases at the 5′ end of the read and bases past number 465 are trimmed since they are below threshold. The center of the figure shows high quality sequence in the middle of the read that is within the retained region. (Actual data shown.)

In the *SeqManII* approach, sequences are added one at a time, from left to right, to a growing contig. As each read is added, its sequence is aligned to the region of the contig consensus that it overlaps. The approximate offset of the read in the contig as determined by *SLIC* is used to determine the overlapping region. First *Martinez* (exact substring) matches (Martinez 1983) are found between the read and the consensus. Then *Needleman-Wunsch* (Needleman & Wunsch 1970) is used to align and gap the read with the consensus between consistent pairs of *Martinez* matches. Figure 8-2 (continued on the next page) steps through the alignment process using the *SeqManII* method.

Sequence		**Offset**
Read 1.	**TCGAAGTCCCTACGACCACTTTAGGGCGGG**	0
Read 2.	**AGTCCTACGACCTCTTACGGCGGGATCACGCATCCATTTAC**	6
Read 3.	**CTTAGGCGGGATCACGCATCCATTTTACGAAAT**	19

Create a new contig with read 1.

```
TCGAAGTCCCTACGACCACTTTAGGGCGGG
```

Add read 2 at offset 6.

```
TCGAAGTCCCTACGACCACTTTAGGGCGGG
      AGTCCTACGACCTCTTACGGCGGGATCACGCATCCATTTAC
```

Align exact matches between consensus and read 2 (boxed).

```
          TCGAAGTCCCTACGACCACTTTAGGGCGGG
              AGTCCTACGACCTCTTACGGCGGGATCACGCATCCATTTAC
consensus TCGAAGTCCCTACGACCACTTTAGGGCGGG
```

Gap and align between exact matches (grayed).

```
          TCGAAGTCCCTACGACCACTTTAGGGCGGG
             AGT-CCTACGACCTCTT-ACGGCGGGATCACGCATCCATTTAC
consensus TCGAAGTCCCTACGACCACTTTAGGGCGGG
```

Figure 8-2. ***SeqManII*** **Alignment** **(continued on next page).**

Add read 3 at offset 19.

```
TCGAAGTCCCTACGACCACTTTAGGGCGGG
    AGT-CCTACGACCTCTT-ACGGCGGGATCACGCATCCATTTAC
                    CTTAGGCGGGATCACGCATCCATTTTACGAAAT
```

Align exact matches between consensus and read 3 (boxed).

```
          TCGAAGTCCCTACGACCACTTTAGGGCGGG
              AGT-CCTACGACCTCTT-ACGGCGGGATCACGCATCCATTTAC
                              CTTAGGCGGGATCAGGCATCCATTTTACGAAAT
consensus TCGAAGTCCCTACGACCACTTTAGGGCGGGATCACGCATCCATTTAC
```

Gap and align between exact matches (grayed).

```
          TCGAAGTCCCTACGACCACTTTAGGGCGGG
              AGT-CCTACGACCTCTT-ACGGCGGGATCACGCATCCATTT-AC
                            CTT-A-GGCGGGATCAGGCATCCATTTTACGAAAT
consensus TCGAAGTCCCTACGACCACTTTAGGGCGGGATCACGCATCCATTT-AC
```

Figure 8-2. ***SeqManII*** **Alignment (continued from previous page).** The approximate offset for three sequence reads have been determined by *SLIC*. First, read 1 is used to create a new contig. The second read is added at its offset. Consistent exact substring matches between the read and the consensus are aligned, then the *Needleman-Wunsch* method is used to gap and align between the matches. The third read is added in the same manner as the second.

The other method, *ReAligner*, was developed to gap and align nearly aligned sequences (Anson & Myers 1997). The steps used in *ReAligner* are listed next.

Align with ReAligner

Create a near multiple-sequence alignment with all sequence reads.

While alignment improves do

1. Get next column in alignment.
2. Gap and align columns following current column.

First the approximate offsets as determined by the *SLIC* layout algorithm are used to create a near multiple sequence alignment. *ReAligner* then makes multiple passes processing one column at a time, improving the alignment with each iteration. When there is no improvement in the alignment, processing ceases. Figure 8-3 illustrates an alignment using *ReAligner*.

Sequence		Offset
Read 1.	TCGAAGTCCCTACGACCACTTTAGGGCGGG	0
Read 2.	AGTCCTACGACCTCTTACGGCGGGATCACGCATCCATTTAC	6
Read 3.	CTTAGGCGGGATCACGCATCCATTTTACGAAAT	20

Create a near multiple-sequence alignment with all 3 reads.

```
TCGAAGTCCCTACGACCACTTTAGGGCGGG
      AGTCCTACGACCTCTTACGGCGGGATCACGCATCCATTTAC
                    CTTAGGCGGGATCACGCATCCATTTTACGAAAT
```

Iterate over columns, improving the alignment.

```
TCGAAGTCCCTACGACCACTTTAGGGCGGG
    AGT-CCTACGACCTCTTACGGCGGGATCACGCATCCATTTAC
                    CTTAGGCGGGATCACGCATCCATTTTACGAAAT
```

Cease processing when the alignment fails to improve.

```
TCGAAGTCCCTACGACCACTTTAGGGCGGG
    AGT-CCTACGACCTCTT-ACGGCGGGATCACGCATCCATTT-AC
                  CTT-A-GGCGGGATCAGGCATCCATTTTACGAAAT
```

Figure 8-3. *ReAligner* Alignment. A near multiple-sequence alignment is created using the *SLIC* approximate offsets. The offsets were determined by the alignment of the boxed subsequence. *ReAligner* iterates over the aligned columns, gapping and aligning, until there is no improvement.

Consensus

The basic idea for the consensus calling algorithm used in the *SLIC Assembler* is the same as the *Trace-Evidence* method described in Chapter 6. Traces are examined and evidence for each base is summed. However, the calculation of the evidence scores and the steps in summing have been refined. I will refer to the refined method as *Trace-EvidenceII*. I incorporated this consensus calling method into DNASTAR Inc.'s *SeqManII* fragment assembly program. It is commercially available in the *Lasergene99* suite of applications.

With *Trace-EvidenceII*, *Trace-Class* scores are not used to determine evidence scores, rather a single score is computed. Nevertheless, the same trace characteristics that are examined for *Trace-Class* scores are evaluated for the refined calculation. Each set of trace data points (*A*, *C*, *G*, or *T*) associated with a single base call are scanned to find peaks (valleys are no longer used). Peaks may be as obvious as those that have changes in sign from positive to negative slope or may simply have a convex shape. The more defined the peak and the higher its intensity, the higher the score.

An additional change in assigning evidence scores is that peaks that occur in runs of identical bases are assigned scores that more accurately reflect their intensity. Before the refinement of the calculation, peaks in runs were given inappropriately low scores because within a run the trace does not descend as low on either side of the peak as it does when not in a run. The minimum intensity to either side of the peak was used for the right and left extremes in calculating the evidence score. The refined calculation in *Trace-EvidenceII* uses the minimum intensities on either side of the *run*, rather than either side of a *peak*, for the extremes, resulting in more accurate scores for peaks in runs. The extremes are found as follows.

Get Extremes

If the peak base call is in a run of identical bases then
    The left peak extreme, *L*, is the min of all trace points
        in the run to the left of the peak
    The right peak extreme, *R*, is the min of all trace points
        in the run to the right of the peak

After all evidence scores for a column of aligned bases have been determined, the algorithm for summing the score for *Trace-EvidenceII* is as previously described in Chapter 6 for *Trace-Evidence* with one difference. The change was made necessary by spurious peaks whose scores overwhelmed the scores for true peaks. The false peak may be the result of fluorescent-dye contamination or a *chimeric* read (a read that contains erroneously joined fragments from discontiguous sources); Figure 8-4 contains examples. In determining the consensus, a spurious peak is identified by its single occurrence among at least three other matching peaks in an aligned column. When a putative false peak is found, the summed score for the base associated with the false peak is reduced before determining the consensus. By adjusting the evidence sum in this way, far fewer ambiguous calls occur in the consensus. The algorithm for adjusting evidence scores follows.

Adjust Evidence Scores

For each sequence in a column
    If the maximum evidence is for *A*, increment $\texttt{count}_{\texttt{A}}$
    Else if maximum evidence is for *C*, increment $\texttt{count}_{\texttt{C}}$
    Else if maximum evidence is for *G*, increment $\texttt{count}_{\texttt{G}}$
    Else if maximum evidence is for *T*, increment $\texttt{count}_{\texttt{T}}$
Find the highest (`max_count`) and second highest (`next_count`) counts
If `max_count` is at least 3 and `next_count` is 1 then
    For each `i` = *A*, *C*, *G*, *T*
        $\texttt{evidence\_sum}_{\texttt{i}} = \texttt{evidence\_sum}_{\texttt{i}} * \texttt{count}_{\texttt{i}} / \texttt{max\_count}$

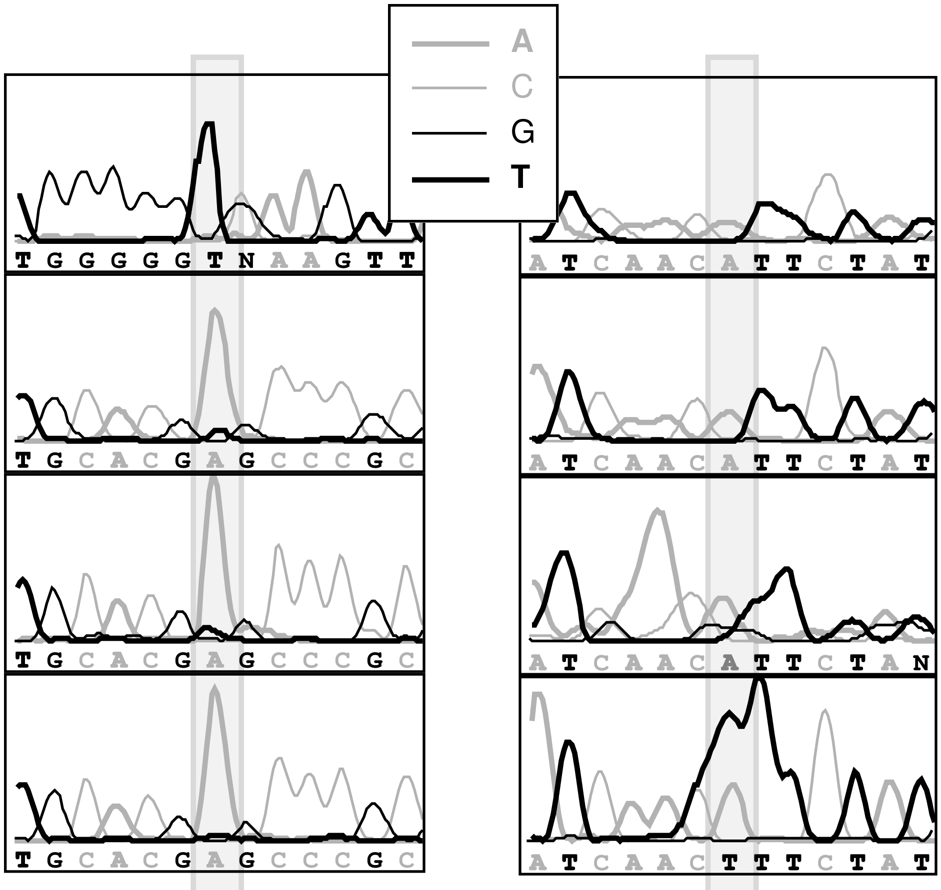


Figure 8-4. Spurious Peaks. In the alignment on the left, a chimeric read has been aligned with three other matching reads. The correct consensus call for the shaded column is an A, but with the original *Trace-Evidence* algorithm, the high T peak in the top sequence results in an ambiguous consensus call of W. On the right, the correct call is also A in the shaded column, but the unrefined *Trace-Evidence* returns a W due to the high T peak in the fourth sequence. When the sum of the evidence scores are adjusted using the *Trace-EvidenceII* method, the consensus call in both cases is an A, as desired. (Actual data shown.)

8.2.2 *SLIC* Algorithmic Details

The *SLIC* layout algorithm relies on subsequences of bases, or *mers*, that occur in overlapping regions of fragment reads. Mers that are common to two or more fragment reads are aligned to determine the overall layout of reads. The premise is that large DNA fragments contain many mers that occur only once (or infrequently) and that can be used to tag relative positions of fragment reads (Jain and Myers 1997). The idea of using mers to tag fragments and identify overlaps is illustrated in Figure 8-5.

Actual Sequence:

CGAATGTCATATGGCAGTACACGGCGTACGTTAGGTTTCTGAGGGATTTTCGAG

Fragment Reads:

1. **CGAATGTCATATGGCAGTA**
2. **TATGGCAGTACACGGCGTACGT**
3. **GGCGTACGTTAGGTTT**
4. **TTAGGTTTCTGAGGGATT**
5. **AGGTTTCTGAGGGATTTTCGAG**

Fragment Read Layout:

```
1. CGAATGTCATATGGCAGTA
2.          TATGGCAGTACACGGCGTACGT
3.                        GGCGTACGTTAGGTTT
4.                                TTAGGTTTCTGAGGGATT
5.                                  AGGTTTCTGAGGGATTTTCGAG
```

Figure 8-5. Using Mer Tags to Identify Overlaps. In this case, I have a 54 bp actual sequence that is covered by five overlapping fragment reads. The 6-mer tags for each fragment read are underlined. I align matching mer tags to determine the layout of the reads.

Since the *SLIC* algorithm relies on mer tags to identify overlapping regions of reads, this approach can work well only with data that is fairly error free. Fortunately, sequencing technology has now advanced to the point that at least several hundred consecutive base calls per fragment read are highly accurate. In addition, to remove noisy ends of the reads, I have developed methods that trim based on the quality of the traces. These both help to ensure that the data is sufficiently error-free to assemble successfully with *SLIC*.

The fundamental challenge of using mers to tag fragments lies in choosing tags that are most likely to be unique. In choosing tags, I consider two factors. The first is the length of mers; the longer the mer, the more probable that the mer is unique. In practice, the length of a mer tag is limited by the length of fragment read overlaps and is set before processing. The second factor is the number of occurrences of each mer in the data set. If a mer occurs more often than expected, I suspect that the mer is part of a repeated region of DNA and preferentially choose a mer with fewer occurrences.

A brief overview of *SLIC*, the linear-time layout algorithm, is listed next; detailed pseudocode appears in Appendix D. I make three linear scans through all of the base call sequences in the data set.

SLIC Overview

1. Initialize all variables and structures.
2. Read sequences.
3. Count occurrences of mers in all fragment reads.
4. For each fragment read, choose mer tags using mer counts.
5. For each read, if a mer is chosen as a tag for any previous read, choose it as a tag for the current read.
6. Make contigs.

In the initialization step, all variables and structures are cleared. In the second step, sequences are read and stored. In step three, I scan all of the reads keeping track of the counts of the occurrences of mers. In the fourth step, I choose mer tags for each read. The fifth step ensures that if a mer was previously chosen as a tag for any fragment read, it is chosen for all reads in which it occurs. In the last step, the lists of reads associated with mer tags are used to form contigs.

The goal of the first scan (step 3 in the overview) is to obtain a count the occurrences of each mer of fixed length k that appears in the data set. (I use the counts in the next scan to determine which k-mers to prefer as mers used to tag overlaps of reads.) I process each fragment read in turn, incrementing the total tally of counts for each k-mer that occurs. The counts are kept in a bucket-and-chain hash table to preserve the linear nature of the algorithm. In the table, the integer value of an encoding of the first first x bases of the mer specifies a bucket. In the encoding, two bits are used to represent each base call; A, G, T, and C are represented by 00, 01, 10, and 11, respectively. Since two bits encode each base in the x-mer, the length of the table is 4^x. The last y bases in the mer specify a mer record in the chain. Figure 8-6 gives an example of a mer hash table.

a)

Index	Mer (first x bases)	2-Bit Encoding
0	AAAAAAAA	0000000000000000
1	AAAAAAAG	0000000000000001
2	AAAAAAAT	0000000000000010
…	…	…
65,533	CCCCCCCG	1111111111111101
65,534	CCCCCCCT	1111111111111110
65,535	CCCCCCCC	1111111111111111

b)

Hash Index (first x bases)	Mer Chain (last y bases)		
	Mer 1	Mer 2	Mer 3
31,352 (GCTTGCTA)	mer GTCA count 1	mer TTAC count 6	mer GAAT count 3

Figure 8-6. Mer Bucket and Chain Hash Table and Lists. a) This is an example of bucket indices in a hash table of mers. In this example, x = 8 and the length of the table is 65,536. The integer values of the 2-bit encoding of the 8-mers indexes the table. b) I use the last y bases in the current mer to find the corresponding chain record for a particular mer. In this example, consider the following three mers where x = 8 and y = 4: GCTTGCTAGTCA, GCTTGCTATTAC, and GCTTGCTAGAAT. The first x bases are identical, so all hash into the table at index 31,352. The last y bases are different, so each has its own record. In the data set of fragment reads for this example, I have so far encountered one mer of GCTTGCTAGTCA, six of GCTTGCTATTAC, and three of GCTTGCTAGAAT. At the end of step 3, there is one record for each mer that occurs in the data set of fragment reads.

Before counting, a threshold is set that specifies a maximum number of identical mers. When the number of occurrences surpasses the threshold, the mer is marked as a repeat. My premise is that if the number of occurrences of a mer is significantly above expected, the mer is likely to be a repeated subsequence. Since my goal is to choose unique mers, I do not consider the putative repeats. The default setting for the threshold is 150% of expected redundancy. The expected number is simply the average coverage for the actual sequence:

$$\frac{\sum_{i=1}^{\text{number of reads}} \text{read length}_i}{\text{sequence length}}$$

The numerator can be estimated by multiplying the number of reads by the expected read length (usually about 500 bp).

A summary of step 3 (first of three read scans) follows.

Count Mers

For each read in a data set
- For each mer of length *k=x+y* in the read
  - Index the hash table using the first *x* bases in the mer
  - If no record exists for the last *y* bases in the mer then
    - Create a new record
  - Increment the mer count in the record
  - If the count exceeds the repeats threshold then
    - mark the mer as a repeat

In the second of the three scans of the fragment reads (step 4 of the overview), I choose the initial set of putative unique mers for each of the fragment reads. Theoretically, any number of mers can be chosen for each read, but I need to balance completeness with efficiency. If I choose too few mers, I risk missing some overlaps. In choosing more mers than are necessary, I waste storage and processing time. One obvious answer is to choose two mers per read – one at either end. With perfect data, this is sufficient, but given that the data near the ends is more error-prone, overlaps may be missed using this scheme. Figure 8-7 illustrates some potential problems with various numbers of mer tags. In practice, I choose three to five mers per read; one near either end with the others distributed between.

Actual Sequence

```
CGCATGCAAAAGTGATCGGGTATCACGCACGTATTCTTAGCAGAGTTATCCAACCA
```

Correct Fragment Read Layout

```
1. CGCATGCAAAAGTGATCGGGTATCACG
2.         AAAGTGATCGGGTATCACGCACGTATTC
3.                        CACGCACGTATTCTTAGCAGAGTT
4.                               GTATTCTTAGCAGAGTTATCCAACCA
```

a) First possible result

Contig 1

```
1. CGCATGCAAAAGTGATCGGGTATCACG
2.         AAAGTGATCGGGTATCACGCACGTAT
```

Contig 2

```
3.                        CACGCACGTATTCTTAGCAGAGTT
4.                               GTATTCTTAGCAGAGTTATCCAACCA
```

b) Second possible result

Contig 1

```
1. CGCATGCAAAAGTGATCGGGTATCACG
2.         AAAGTGATCGGGTATCACGCACGTATTC
3.                        CACGCACGTATTCTTAGCAGAGTT
4.                               GTATTCTTAGCAGAGTTATCCAACCA
```

Figure 8-7. Unidentified Overlaps (continued on next page).

c) Third possible result

Contig 1

```
1.   CGCATGCAAAAGTGATCGGGTATCACG
2e.            AAAGTGATCGGGTATCACGCACXTATTC
```

Contig 2

```
3.                                 CACGCACGTATTCTTAGCAGAGTT
4.                                          GTATTCTTAGCAGAGTTATCCAACCA
```

Figure 8-7. Unidentified Overlaps (continued from previous page). I have an actual sequence that is covered by four fragment reads. In a)-c), the mer tags chosen initially are singly underlined; their matches are marked with a double underline. a) I choose a single 6-mer tag in the center of each read. The tags identify overlaps between reads 1 and 2 and between reads 3 and 4, but miss the overlap of read 2 with 3. b) A mer tag is chosen for either end of each read. The tags correctly identify all read overlaps. c) Choosing one mer for either end of the read is not sufficient if base calling errors result in mismatched tags. A base calling error in the base sixth from the end in read 2 prevents finding its overlap with reads 3 and 4.

As I make the second scan, I first divide each read into as many partitions as the specified number of mers per read. Then, if possible, a mer tag is chosen in each of these partitions. I use a simple criterion to choose a mer; I prefer mers with the fewest number of occurrences. (The number must be at least two to identify overlapping fragments.) Again, my premise is that the fewer the number of occurrences, the more likely that the mer is unique in a given large DNA fragment or genome. Often there are ties in the number of occurrences and my choice of mer tag is dependent upon which partition I am processing. Ideally, I want to choose mer tags that are at either end of the fragment read and spaced evenly throughout the rest of the read. Given this, if I am choosing for the first partition, I choose the first mer with the fewest occurrences. Conversely, if I am choosing for the last partition, I choose the last mer with the fewest occurrences. For middle partitions, I choose the mer with the fewest occurrences that is nearest the center of the partition. Figure 8-8 illustrates breaking ties in a partition.

Actual Sequence

CGCATGCAAAAGTGATCGGGTATCACGCACGTATTCTTAGCAGAGTTATCCAACCATA

Fragment Read Layout

```
1. CGCATGCAAAAGTGATCGGGTATCACGCACGTATTC
2.     TGCAAAAGTGATCGGGTATCACGCACGTATTCTTAGCAGAGTTATC
3.                        CACGCACGTATTCTTAGCAGAGTTATCCAAC
4.                            CACGTATTCTTAGCAGAGTTATCCAACCATA
```

Read 1

Partitions:	**CGCA<u>TGCAAAAG</u>**	**TG<u>ATCGGGTA</u>TC**	**AC<u>GCACGTAT</u>TC**
Number of Occurrences:	**111122222222**	**222222222223**	**33344-------**

Figure 8-8. Breaking Mer Tag Ties. I show an actual sequence and three fragment reads. I want to choose three 8-mer tags per read. Before choosing mers for read 1, I divide it into three partitions. Then I examine the number of mer occurrences for each partition. In the first partition, I do not choose any of the first four mers since they occur only once and can not identify an overlap. Next, I have an eight-way tie with two occurrences each. I want a mer nearest the end, so I choose the first mer in the tie. In the middle partition, I want to choose one of the mers with two occurrences, since that is the fewest. I break the tie by choosing the mer nearest the center of the partition. In the last partition, I choose the last mer with three occurrences so that the mer will be as near to the end as possible. Each mer tag chosen for read 1 is underlined.

A summary of step 4 (second of three read scans) follows.

Choose Mer Tags

For each read in a data set
    Divide the read into *m* partitions.
            (*m* is the specified number of mers per read)
    For partition 1 (choose the first mer with the lowest count)
        While a mer tag is not chosen
            Get the next mer (in 5′ to 3′ order)
            If the mer is not a repeat then
                Get the count of the occurrence of the mer
                If the count is greater than 1 then choose the mer as a tag

For partitions 2 to m - 1 (choose the most central mer with the lowest count)
    For each mer in the partition
        If the mer is not a repeat then
            Get the count of the occurrence of the mer
            If the count is greater than 1,
                    less than or equal to the count of a previous mer tag,
                    and nearer to the center of the partition than a previous
                    mer tag then
                Choose the mer as a tag
For partition m (choose the last mer with the lowest count)
    While a mer tag is not chosen
        Get the next mer (in 3′ to 5′ order)
        If the mer is not a repeat then
            Get the count of the occurrence of the mer
            If the count is greater than 1 then choose the mer as a tag

The third scan through the data (step 5 in the overview) ensures that if a mer was chosen for any read, the mer is also chosen for all other reads containing that mer. The summary for this scan is listed next.

Choose Previously Chosen Mer Tags

For each read in a data set
    For each mer in the read
        If the mer is already chosen as a mer tag for any read then
            Choose the mer as a tag for the current read

By doing this, all fragment reads that align will be placed in the same contig. The necessity of this step is shown in an example in Figure 8-9 where a mer has been chosen as a tag by one fragment in the second scan, and is chosen in the third scan by another fragment that also contains the mer.

Actual Sequence

XXXXXXXXXXXXXXXAAAAAAAAABBBBBBBBCCCCCCCCYYYYYYYYYYYY

Fragment Reads

1. XXXXXXXXXXXXXXXAAAAAAAAABBBBBBBB
2. AAAAAAAAABBBBBBBBCCCCCCCC
3. CCCCCCCCYYYYYYYYYYYY

Scan 2

```
1. XXXXXXXXXXXXXXXAAAAAAAAABBBBBBBB
2.                AAAAAAAAABBBBBBBBCCCCCCCC
3.                                 CCCCCCCCYYYYYYYYYYYY
```

Scan 3

```
1. XXXXXXXXXXXXXXXAAAAAAAAABBBBBBBB
2.                AAAAAAAAABBBBBBBBCCCCCCCC
3.                                 CCCCCCCCYYYYYYYYYYYY
```

Figure 8-9. Choosing Mer Tags. The actual sequence has five regions designated by X, A, B, C, and Y. Regions A, B, and C represent the three mer tags that will be chosen in scans 2 and 3. In this example, I am going to choose two 8-mers per read; i.e. I divide each read into two parts and choose a mer in each partition whenever possible. Mer tags chosen in the second scan are single underlined and those chosen in the third scan are double underlined. There are three fragments reads in the data set. In the second scan, there is no mer to choose for the first half of read 1 since the mers occur only once. In the second half, I choose the the mer closest to the end of the read (BBBBBBBB). For the second read, I choose the mers closest to either end (AAAAAAAAA and CCCCCCCC). The mer tag chosen for the third read is CCCCCCCC. At this point I have established a relationship between reads 2 and 3, but not between reads 1 and 2. In the third scan, I additionally choose AAAAAAAAA for read 1 and BBBBBBBB for read 2. At this point I have accounted for all overlaps.

Note that the amount of storage and processing time can be reduced by incorporating a check for previously chosen mers in the second scan. In that scan, if a mer in the current partition has been previously chosen, the mer can be immediately chosen as the mer tag for the current fragment. However, checking for previously chosen mers in the second scan does not eliminate the need for the third scan, as shown in Figure 8-10.

Scan 2

```
1. XXXXXXXXXXXXXXXXAAAAAAAAAABBBBBBBB
2.         XXXXXYXXAAAAAAAAAABBBBBBBBCCCCCCCC
3.                                   CCCCCCCCYYYYYYYYYYYY
```

Scan 3

```
1. XXXXXXXXXXXXXXXXAAAAAAAAAABBBBBBBB
2.         XXXXXYXXAAAAAAAAAABBBBBBBBCCCCCCCC
3.                                   CCCCCCCCYYYYYYYYYYYY
```

Figure 8-10. Choosing Mer Tags More Efficiently. I have the same actual sequence, regions, reads, and number of mers as in Figure 8-9. Again, in the second scan, I choose the mer BBBBBBBB for read 1. For read 2, I choose BBBBBBBB since it is in the second half and was already chosen for read 1. Note that I do not choose CCCCCCCC for read 2 since I have already chosen a mer for the second half. The mer tag chosen for the third read is still CCCCCCCC. Even though I checked for previously chosen mers during the second scan, I have not established a relationship between reads 2 and 3. To identify the overlap, I still need the the third scan in which I additionally choose CCCCCCCC for read 2. I have, however, reduced the amount of mer information that I must store and process. Mer tags chosen in the second scan are single underlined and the mer chosen in the third scan is double underlined.

By the end of the third scan of the data, I have a list of mers that have been chosen as tags for reads. Associated with each mer tag list is a corresponding list of all fragment reads that contain the mer. This is the information I will use to determine the layout of fragment reads in contigs.

To form contigs, I iterate through the lists of chosen mer tags, checking the pairwise similarity of the fragment reads (The computational complexity analysis in Chapter 10 explains why this pairwise comparison does not compromise the linear nature of algorithm.) The check is necessary to ensure that the mers tags identify actual overlaps of fragments reads. Figure 8-11 illustrates this point.

Actual Sequence

AAAAAAAAAAABBBBBBBBCCCCCCCCCCCCCCCCCCBBBBBBBBDDDDDDDDDDDDDD

Fragment Reads

1. **AAAAAAAAAAA<u>BBBBBBBB</u>CCCCC**
2. **A<u>BBBBBBBB</u>CCCCCCCCCCCCCCCCCCBB**
3. **CCCCCCCCCCCCCCC<u>BBBBBBBB</u>DDD**
4. **CCCCCC<u>BBBBBBBB</u>DDDDDDDDDDDDD**

Overlaps (via mer <u>BBBBBBBB)</u>

```
1.         AAAAAAAAAAABBBBBBBBCCCCC
2.                   ABBBBBBBBCCCCCCCCCCCCCCCCCCBB
3. CCCCCCCCCCCCCCCBBBBBBBBDDD
4.              CCCCCCBBBBBBBBDDDDDDDDDDDDD
```

Figure 8-11. False Overlaps with Non-Unique Mer Tags. In the example, I have an actual sequence and four fragment reads. Sections of the fragment are designated by A, B, C, and D. Note that the read identified by B is repeated. When I align the B mer tag for the four reads and check the pairwise overlap similarity, I find that the four reads do not match and should not be overlapped. At the same time I recognize that the first two and the last two reads match so I divide the list into two, placing each matching pair in a new list.

Although in theory all mers chosen as tags are unique in the original large fragment of DNA, this is not the case in practice. Some mer tags chosen may not be unique; the pairwise comparisons help to identify these tags. For the comparison, a threshold is set that specifies the required amount of match similarity. The similarity is checked in a rolling window over the overlapping region so that an extremely good match in one region does not compensate for a poor match in another. See Figure 8-12 for an example of the need to checking match similarity in a rolling window. Pseudocode for checking match similarity in a rolling window follows.

ACCCATAGATGCGACTGGTAGACAGTGACACGATAGGCTAATTTACGGCAGCATTAAAGT
ACCCATAGATGCGACTGGTAGACAGTGACACGATAGGCTAATTTGATCGATCAGTCAGCC
????? ? ? ? ???

Figure 8-12. Pairwise Similarity in a Rolling Window. This example aligns two fragment reads ('?'s mark mismatches). The overall match similarity is 82% (49/60) which would exceed an 80% threshold. However, if the read is all good data, the first part is probably a repeat or chimeric and it is clear that the fragments should not be aligned. By doing a rolling similarity check with a window of size 20, I find that the last twelve windows have similarities ranging from 45-75%. These fall below an 80% threshold and result in marking the mer as a repeat.

Check Similarity

Set all positions in `window` to 0
Set `window_idx` to 1
Set `max_mismatches` to `window_size` * (1- `threshold`)
Set `num_mismatches` to 0

Align overlapping region
Start scanning with the first aligned bases in the overlap
While `num_mismatches` <= max_mismatches and more
          aligned bases in overlap
    Subtract `window` [`window_idx`] from `num_mismatches`
    If the aligned bases do not match then
        Increment `num_mismatches`
        Set `window` [`window_idx`] to 1 to record a mismatch
    Increment `window_idx`, wrapping when necessary
    Advance scan to next aligned bases
If `num_mismatches` <= `max_mismatches` then similarity is OK

After the pairwise similarity checks, all reads in a list have sufficient pairwise similarity in their overlaps. However, if any of the fragments are already in a contig, I must then also check the pairwise similarity of all fragments in the contig with any overlapping fragments in the list. (The fragments in a contig might not contain the current mer, but might still overlap some of the fragment reads in the current list.) An example in Figure 8-13 illustrates the need for checking all fragments in an existing contig with any overlapping fragment reads in the list.

Actual Sequence

AAAAAAAABBBBBBBBCCCCCCCCCCCCCCCCBBBBBBBBDDDDDDDDEEEEEEE

Fragment Reads

1. AAAAAAAABBBXBBBBCCCCCCCCCCCCCCC
2. AAAAAAAABBBBBBBB
3. BBBBBBBBDDDDDDDD
4. BBBBBBBBDDDDDDDDEEEEEEE

Contig 1 via mer AAAAAAAA
(fragments 1 and 2 are in the list of reads)

1. AAAAAAAABBBXBBBBCCCCCCCCCCCCCCC
2. AAAAAAAABBBBBBBB

Contig 2 via mer DDDDDDDD
(fragments 3 and 4 are in the list of reads)

3. BBBBBBBBDDDDDDDD
4. BBBBBBBBDDDDDDDDEEEEEEE

Overlap of Contigs 1 and 2 via mer BBBBBBBB
(fragments 2, 3, and 4 are in the list of reads)

```
1. AAAAAAAABBBXBBBBCCCCCCCCCCCCCCC
2. AAAAAAAABBBBBBBB
3.         BBBBBBBBDDDDDDDD
4.         BBBBBBBBDDDDDDDDEEEEEEE
```

Figure 8-13. False Overlaps with Sequencing Errors. I have an actual sequence and four fragment reads. Sections of the fragment are designated by A, B, C, D, and E. This time, in fragment read 1, the first B section contains a sequencing error in which the fourth base is called incorrectly. The result is that fragment read 1 does not occur in the fragment read list for BBBBBBBB. When I consider the merge of Contig 1 and 2, it is clear that I need to check the overlap similarity of fragment 1 with the others even though it does not appear in the list. To avoid the risk of incorrect alignments of fragments, I check all overlapping regions of both reads and contigs.

If not all overlapping regions in contigs and reads in a list have above-threshold similarity, I divide the list into new lists such that all the overlapping regions specified by each list do have sufficient similarity. I can then proceed to form contigs with the reads in the lists. A *contig* in this scenario is a list of reads and their approximate offsets. If no read in a list is yet in a contig, a new contig is made and all other reads in the same list are added to it. If at least one fragment is already in a contig, then other fragments in the same list not yet in a contig are added to it. If any other fragments in the list are already in a different contig, the fragments bridge a gap between contigs, and the contigs are merged. This procedure is listed next.

Make List into Contig

If no reads in the list are in a contig then
    Make a new contig
    Add all reads with their approximate offsets to the new contig
Else (at least one read in the list is already in a contig)
    Add to a contig each read in the list that is not in a contig
    If the reads in the list are in more than one contig then merge the contigs

Figure 8-14 gives a brief example of making a list into a contig and Figure 8-15 illustrates merging contigs.

Read Index	Contig
63	none
125	3
42	4
15	none

1. Add read 63 to contig 3
2. Add read 15 to contig 3
3. Merge contigs 3 and 4

Figure 8-14. Making a List into a Single Contig. A mer tag list contains four reads, two in contigs and two not. First the two reads not in a contig are added to contig 3. This leaves the four reads in two contigs so contigs 3 and 4 are merged.

Actual Sequence

```
CGCATGCAAAAGTGATCGGGTATCACGCACGTATTCTTAGCAGAGTT
```

Fragment Read Layout

```
1. CGCATGCAAAAGTGATCGG
2.         AAAGTGATCGGGTATCA
3.                 CGGGTATCACGCACGTATTC
4.                        CACGCACGTATTCTTAGCAGAGTT
```

a)

Contig 1 (via mer AAAGTGAT)

```
1. CGCATGCAAAAGTGATCGG
2.         AAAGTGATCGGGTATCA
```

Contig 2 (via mer CACGCACG)

```
3. CGGGTATCACGCACGTATTC
4.        CACGCACGTATTCTTAGCAGAGTT
```

Merged Contigs 1 and 2 (via mer GGGTATCA)

```
1. CGCATGCAAAAGTGATCGG
2.         AAAGTGATCGGGTATCA
3.                 CGGGTATCACGCACGTATTC
4.                        CACGCACGTATTCTTAGCAGAGTT
```

Figure 8-15. Merging Contigs. Two contigs have been formed using overlapping mers. The list of fragment reads for mer GGGTATCA indicates that the reads are already in different contigs so I merge the contigs.

At the completion of the iteration through the list of mer tags, I have a list of contigs. Each contig is represented as a list of information about the fragment reads that are contained in the contig. The information includes an identifier for the fragment read and the offset of the read in the contig. Figure 8-16 contains an example contig list and the implied layouts of fragment reads.

a) Contig Lists

Contig	Read List				
1	index 16 offset 0	index 6 offset 8	index 2 offset 14	index 14 offset 25	index 10 offset 34
2	index 1 offset 0	index 7 offset 9	index 15 offset 19		
3	index 12 offset 0	index 3 offset 6	index 13 offset 17	index 5 offset 33	

b) Layout

```
Contig 1
  Read 16: TAGGCTAGGCCCCATATGC
  Read  6:         GCCCCATATGCTGACGGCGCA
  Read  2:               TATGCTGACGGCGCATTTGAC
  Read 14:                          CGCATTTGACCCCAAAGTC
  Read 10:                                   CCCCAAAGTCCCCG

Contig 2
  Read  1: GATTGGGGACCAGCACCACCTTAGC
  Read  7:          CCAGCACCACCTTAGCAGGA
  Read 15:                    CTTAGCAGGATTGACACGGGTA

Contig 3
  Read 12: TTAGGATCGCGAGCTTA
  Read  3:       TCGCGAGCTTATCCAGAGTCGACCGG
  Read 13:                  TCCAGAGTCGACCGGTAGGGCTACACAAG
  Read  5:                                  AGGGCTACACAAGCCT
```

Figure 8-16. Contig Lists. Three contig lists are shown. Each list contains an identifier and offset for each read in the contig. The reads are ordered by offset in the lists.

I have described in detail the 6 steps required for determining the layout of sequence reads using the *SLIC* algorithm. First all variables and data structures are initialized. Then three linear scans of the reads are executed. The first counts the number of unique mers that occur in the data set. The second scan chooses mers (subsequences of bases) to tag the reads. In the third scan of the reads, any mer tags that were previously chosen for any read are chosen as tags for every read in which they occur. Finally, in the step 6, the mer tags are used to determine the layout of reads. Matching tags between reads are aligned and a rolling similarity check is used to determine if indeed the sequences should be overlapped. All reads that have sufficient similarity are joined in a contig. Information for each contig includes an identifier and offset for each read included in the contig.

8.3 Summary

As the speed of producing sequencing data increases, computational methods for assembling large amounts of data in a time-efficient manner must be developed. Fragment assembly programs currently favored by large genome sequencing centers execute pairwise comparisons of fragment reads, resulting in assembly times that are proportional to n^2 (where n is the number of reads). I have developed an algorithm for fragment layout, *SLIC* (*Sequence Layout into Contigs*), that avoids explicit pairwise comparisons of all reads. By using a hash table to store and retrieve information on the subsequences that occur in reads, the *SLIC* algorithm is, in practice, a function of n. This represents a dramatic decrease in assembly time as the number of reads in an assembly increases.

The *SLIC* layout algorithm is incorporated into a total package for fragment assembly, the *SLIC Assembler*. The assembly steps included in the package are: 1) preprocess to trim poor-quality ends and remove vector sequence, 2) determine the layout of fragment reads using *SLIC*, 3) align the layout of reads, and 4) compute the consensus sequence. I have incorporated two of the revised methods described in this chapter, *Trace-Quality Trim* and *Trace-EvidenceII*, into the latest commercial version of *SeqManII*. It is available as part of DNASTAR Inc.'s suite of applications, *Lasergene99*.

A comparative evaluation of the *SLIC Assembler* with *Phrap* and DNASTAR Inc.'s *SeqManII* is described in the next chapter.

Chapter 9

Evaluation of *SLIC*

I compare assembly performance of the *SLIC* (*Sequence Layout into Contigs*) *Assembler* to *Phrap* from the University of Washington and DNASTAR Inc.'s *SeqManII*. In comparisons, *Phrap* is used because many researchers, especially those in large genome centers, consider it to be the assembly package of choice. *SeqManII* is included since my earlier work is incorporated into this program. All tests were conducted on a Hewlett Packard *Kayak XU* with a 450 Mhz *Pentium II* and 256 MB RAM running *Windows NT 4.0*. In my evaluation, I find that *SLIC* compares favorably to *Phrap* and is superior to *SeqManII*.

9.1 Data Sets

I used four data sets provided by the *E. coli* Genome Project at the University of Wisconsin to compare the performance of *SLIC*. The data sets contain ABI 377 reads from *E. coli* segments H, J, K, and L from the *E. Coli* Genome Project (Blattner *et al.* 1997). The segments are from 211 to 265 kb long and approximately 3000 fragments reads comprise each data set. The data sets are summarized in Table 9-1. The inputs for the tests are ABI 377 data files. Output for each test includes assembled contigs in which the reads have been aligned and gapped.

Table 9-1. Fragment Assembly Test Data Sets. *E. coli* segments H, J, K, and L are used to compare assemblies of *SLIC* with *SeqManII* and *Phrap*. Each fragment is over 200 kb long and the expected depth of coverage is about six to seven sequences.

***E. coli* Segment**	**Fragment Length (kb)**	**Number of Reads**
H	223	2925
J	240	3381
K	265	3443
L	211	3077

9.2 Time

Each of the three programs require processing of the data before assembly to promote optimal results. In the case of *SeqManII*, this includes trimming low-quality data, vector removal, and optimizing the entry order of the reads. *SLIC* requires trimming low-quality data and vector removal. *Phrap* performs best when base calls and error probabilities have been assigned by *Phred* (Ewing & Green 1998). In addition, another companion program, *CrossMatch* is used to remove vector from reads before assembly by *Phrap*. I report the running time of each of the packages in terms of the time needed for both assembly and preprocessing.

On the four data sets, *SLIC* runs in about two thirds the time it takes to run *SeqManII* and from 52% to 93% of the time it takes to assemble with *Phrap*. Figure 9-1 shows the results of timing tests in a bar graph with stacked preprocessing and assembly times. Note that overall, when preprocessing steps are included, *SLIC* assembles reads faster than either of the other programs for all four segments. The reduction from the *Phrap* and *SeqManII* assembly times to the *SLIC* assembly time is statistically significant at the 95% confidence level using a paired one-tailed t-test.

Since I propose that the linear run-time of the *SLIC* layout algorithm makes it especially well-suited to assembling large numbers of sequences, it is important to establish its scalability. To do so, I used *SLIC* to determine the layouts of data sets with numbers of sequences ranging from 2925 to 17,233. For comparison, I also assembled the same data sets with *Phrap*. A graph comparing number of sequences to time of execution are shown in Figure 9-2. In the graph, evidence for a linear relationship with *SLIC* is strong and the relationship as *Phrap* scales is inconclusive.

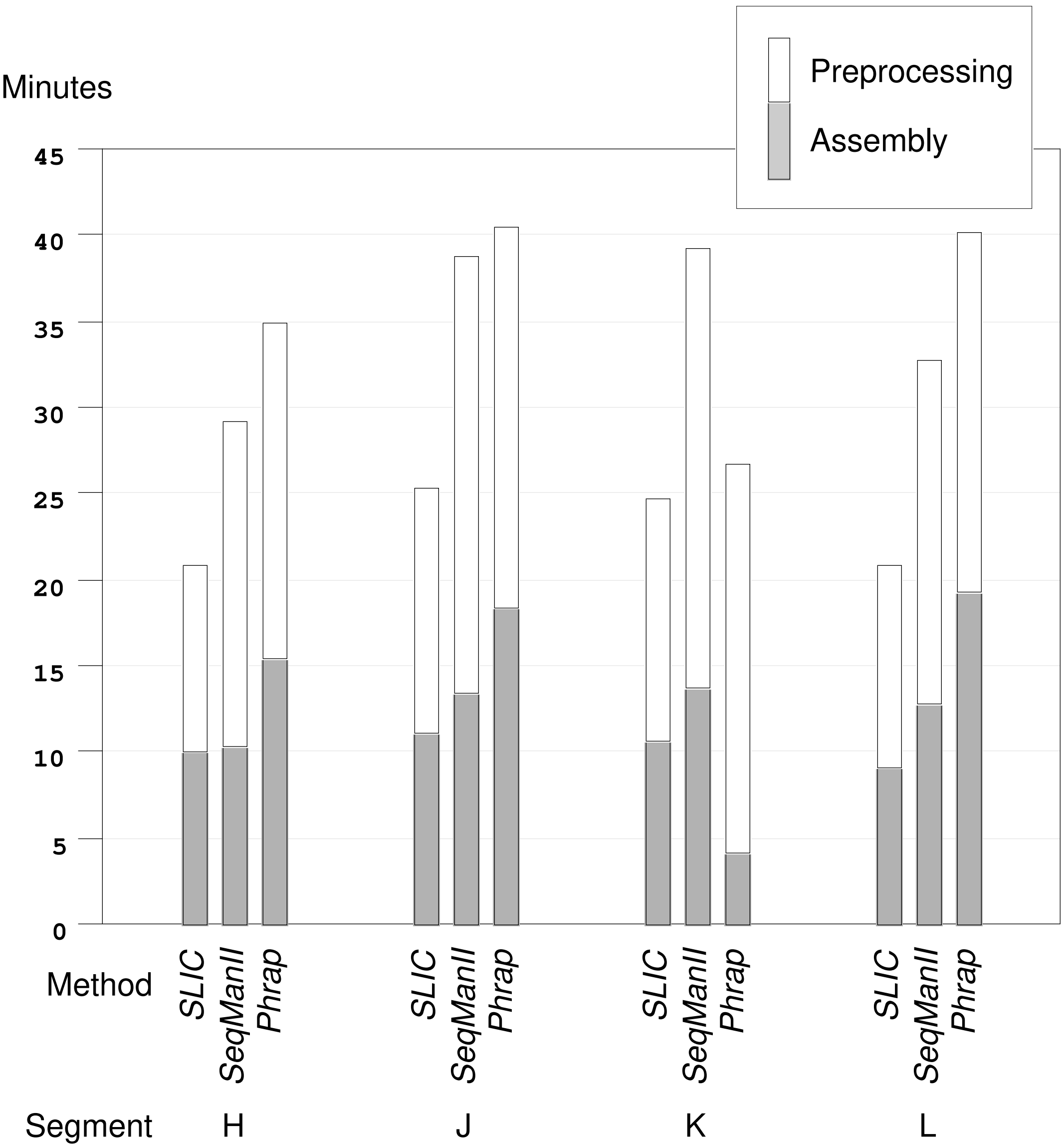


Figure 9-1. Assembly Timing Results. The time to preprocess and assemble the four *E. coli* segments with the three methods is shown. In one segment, K, the time to assemble using *Phrap* is the minimum among the three methods, although the overall time to preprocess and assemble with *SLIC* is still less. In the other three segments, *SLIC* has the lowest assembly and overall times.

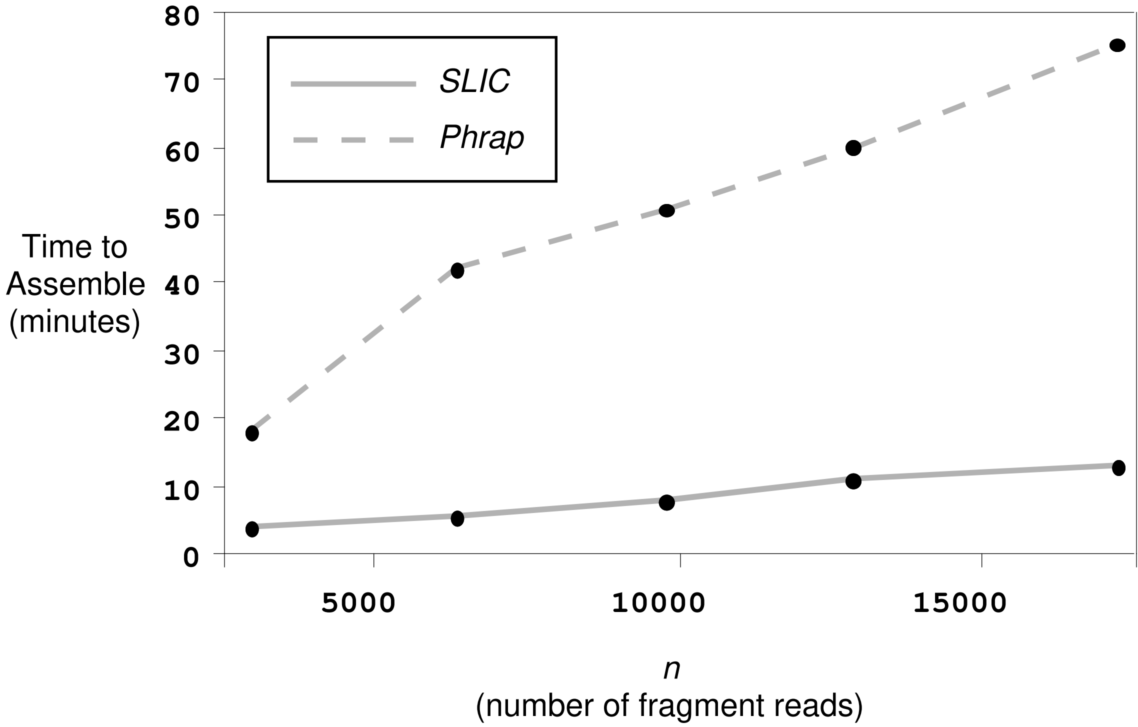


Figure 9-2. Scalability of *SLIC*. The execution times for executing *SLIC* compared to *Phrap* with a range of numbers of sequences is shown. With *SLIC* the relationship remains linear with increasing numbers of sequences. The relationship with *Phrap* is unclear.

9.3 Layout

The layouts produced by *SLIC*, *SeqManII*, and *Phrap* differ most significantly in the placement of repeated regions and in the number of contigs produced. A layout of fragment reads in a contig is *correct* when all overlapping reads are correctly placed relative to each other. I check the correctness of the layouts by aligning the contigs with the GenBank entry for the *E. coli* sequences. In the layouts, some contigs contain *false joins* – reads that are erroneously overlapped due to repeated, or nearly repeated sequences. A false join indicates that an incorrect layout has been produced.

Figures 9-3 through 9-6 show the layouts produced by the three programs. In the figures, horizontal black bars represent individual contigs. To display the relative positions of the contigs, the bars are aligned with the GenBank entry for each segment (the horizontal gray bar). Contigs containing false joins have been split and correctly placed relative to the GenBank sequence. Scissors indicate where contigs have been split and thin lines connect the split contigs so that the original layout can be deduced.

The *SeqManII* layout for segment K contains three false joins and for segment H contains one. Both *SeqManII* and *SLIC* fold a tandem repeat in segment L. In addition, *SLIC* and *Phrap* make one false join in segment K. All three programs correctly assemble segment J.

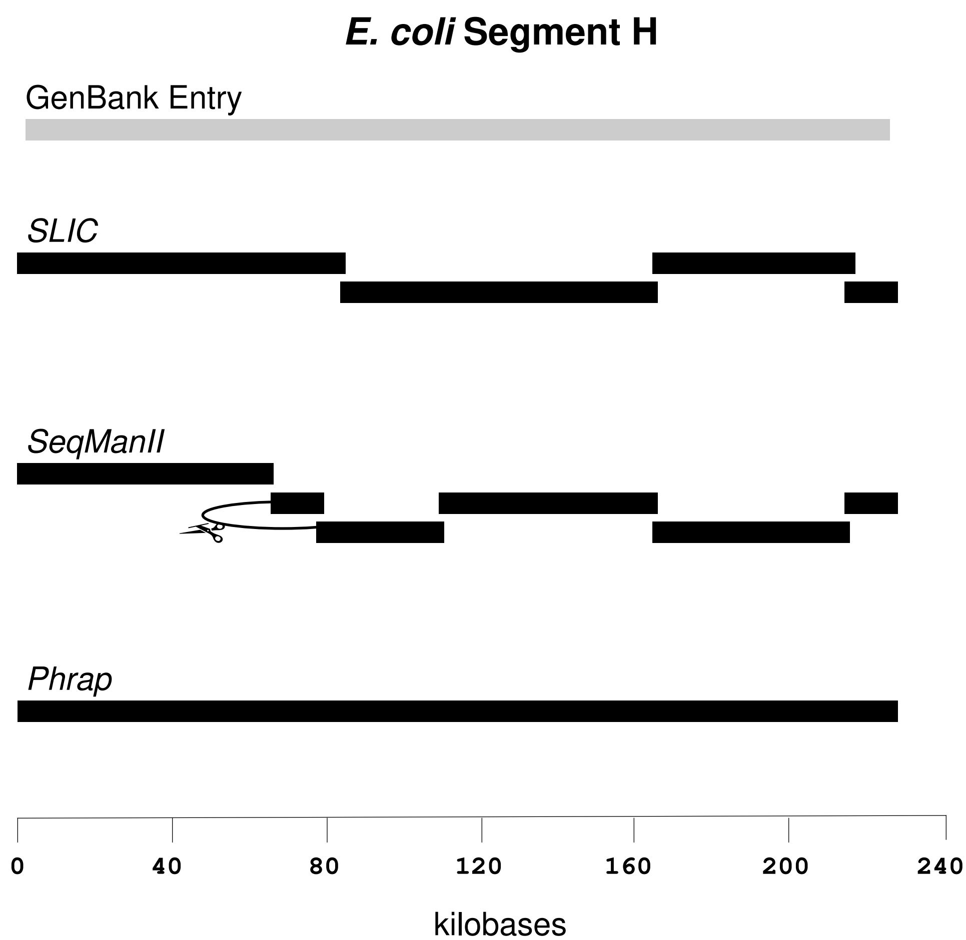


Figure 9-3. Layout of Segment H. *SLIC* and *Phrap* produce correct layouts while the *SeqManII* layout contains one false join.

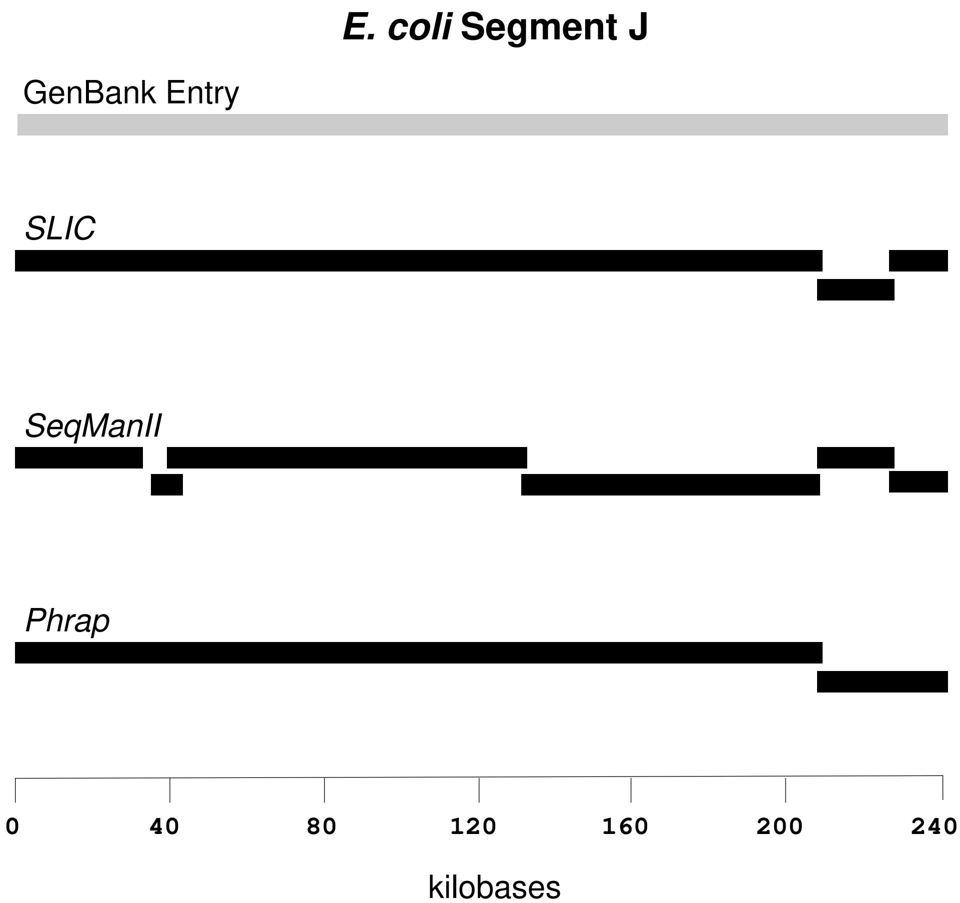


Figure 9-4. Layout of Segment J. All three methods produce correct layouts.

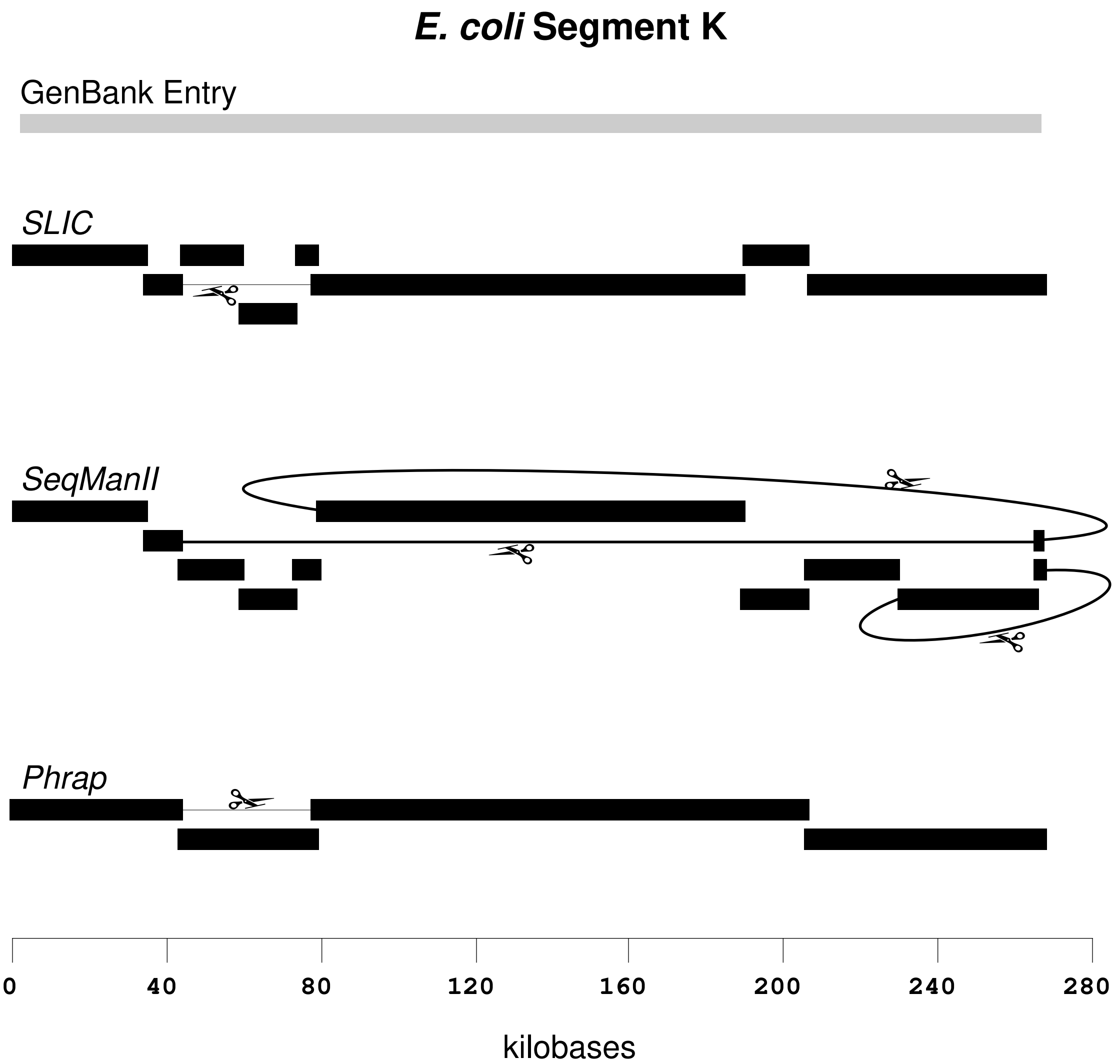


Figure 9-5. Layout of Segment K. All three methods make false joins: one each for *SLIC* and *Phrap*, and three for *SeqManII*.

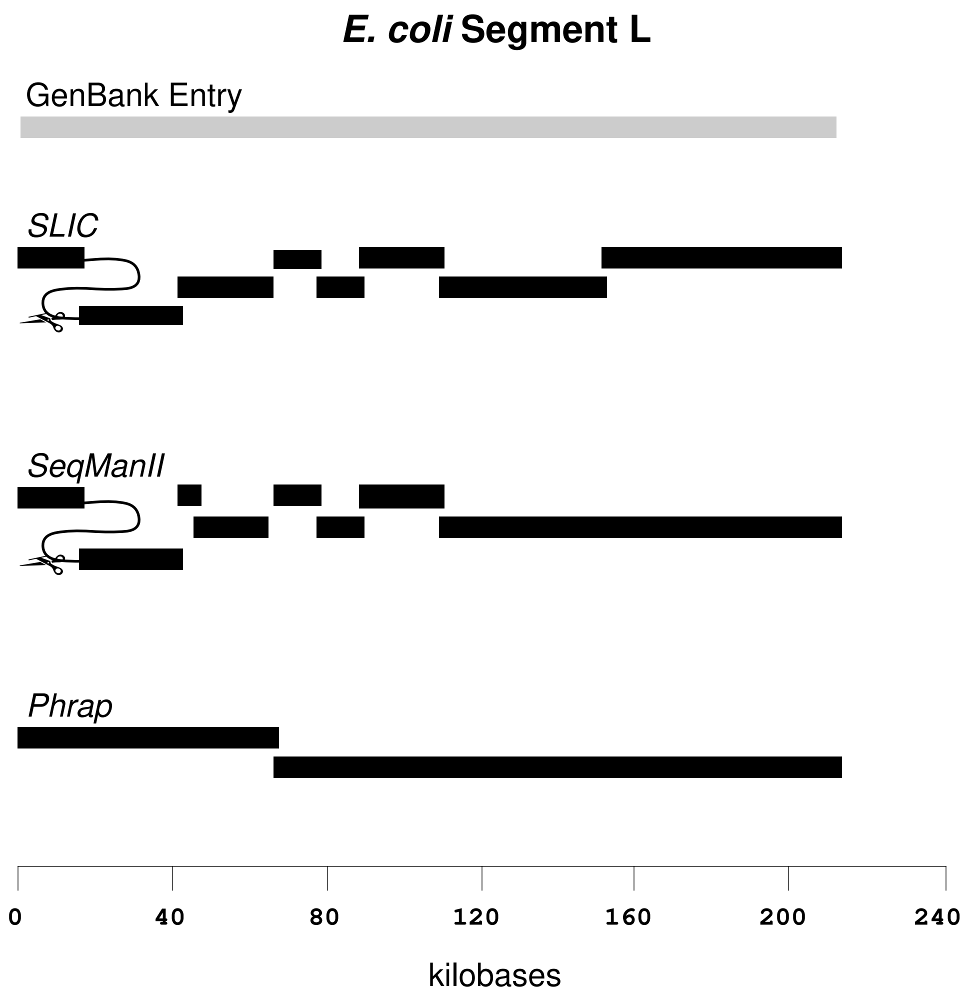


Figure 9-6. Layout of Segment L. *SLIC* and *SeqManII* produce a fold of a tandem repeat that *Phrap* avoids.

Figure 9-7 graphs the number of contigs (larger than 2 kb) generated by each of the three programs for the four data sets. The number of contigs initially produced as well as after splitting the false joins are graphed. In general, *Phrap* produces fewer contigs than either *SLIC* or *SeqManII*. Since *Phrap* does not require trimming of poor quality data before assembly, it makes full use of consistent fragment reads. The trimming that is required for *SLIC* and *SeqManII* results in some gaps between contigs. In addition, overlaps that could have merged contigs may not be recognized if they are short, if there are no mers in common, or if there is insufficient similarity in the overlapping region of the reads. Some instances of insufficient similarity may be due to trimming that is too conservative, leaving significantly noisy ends. When using *SLIC*, the problem is ameliorated by a step I added that makes a post pass using a smaller mer size to check for overlaps between contigs.

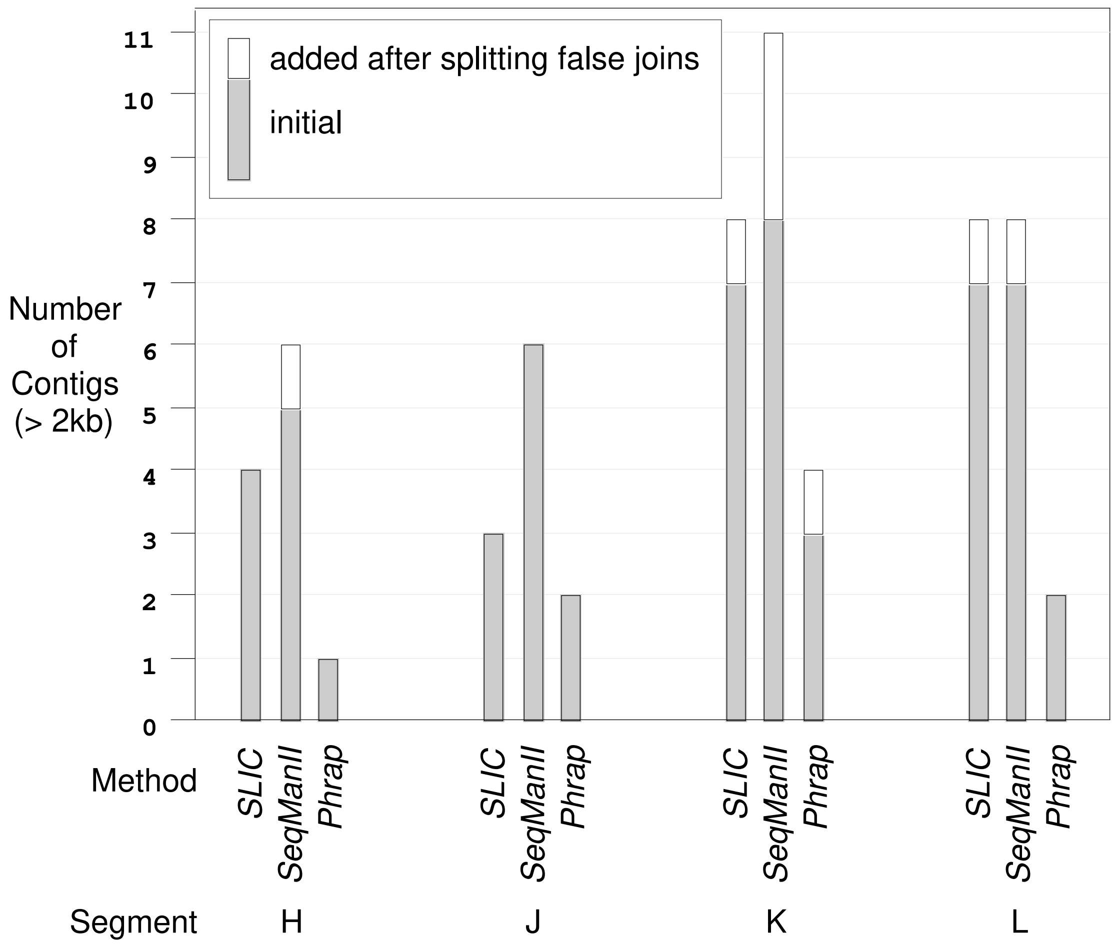


Figure 9-7. Number of Contigs. The number of contigs greater than 2 kb in length produced, both initially and after breaking false joins is shown for the three methods. *Phrap* has the least for all segments. *SLIC* produces fewer contigs than *SeqManII* for three of the segments and ties for the fourth.

It is possible that the *SLIC* algorithm produces different layouts depending upon the input order of fragment reads. Recall that in step 4 of the algorithm (Section 8.2.2), efficiency is increased by incorporating a check for previously chosen mers in the second scan. In that scan, if a mer in the current partition has been previously chosen, the mer can be immediately chosen as the mer tag for the current fragment. This check introduces order-dependency into the algorithm. To evaluate the stability of *SLIC* relative to the order of reads, I assembled each of the four *E. coli* segments five times, each time using a different random order of reads. In general, the contigs produced are essentially the same; the majority contain exactly the same number of sequences in the five assemblies. Of the remaining, the maximum difference in the number of sequences for the same contig is 0.7%. For segments K and L, the same overall layout was produced for all five assemblies. For the other two segments, H and J, the overall layouts are the same in four of five assemblies. In the fifth, one contigs in each was split into two separate contigs. Note that since *SeqManII* orders fragment reads during preprocessing, varying the input order should have no effect on resulting assemblies. Tests on *Phrap* using varying orders of fragment reads are future work.

9.4 Consensus

As part of my testing, I compared the accuracy of three consensus-calling methods: my refined *Trace-EvidenceII*, *Mosaic*, and *Majority*. The *Trace-Evidence* method is detailed in Chapter 6 and its refinements into *Trace-EvidenceII* are described in Chapter 8. The *Majority* method was also covered earlier, in Chapter 5. Recall that in a *Phrap* contig alignment, one base call chosen per aligned column forms a consensus *Mosaic*. The base call chosen is the one with the highest quality score as initially assigned by *Phred* (Ewing *et al.* 1998, Ewing & Green 1998) and adjusted by *Phrap* (Green 1997b, Phrap source code documentation).

All three methods have a system for identifying low-confidence consensus calls that warrant manual examination. To indicate low confidence, *Majority* and *Trace-EvidenceII* return ambiguous calls, and *Phrap* returns lower-case calls. In reporting the accuracy of *Mosaic* consensus sequences, although the obvious interpretation of lower-case calls is that they must be examined individually, lower-case calls can also be interpreted as definitive consensus calls. I report results based on both interpretations of the calls. For one set of tests, I interpret lower-case *Phrap* calls as definitive and set thresholds for *Trace-EvidenceII* and *Majority* such that no ambiguous calls are made. For other tests, I interpret lower-case *Mosaic* calls as ambiguous and set the thresholds for the other methods to default values that allow ambiguous calls.

The accuracy of consensus calls made using the *Majority* and *Trace-EvidenceII* methods are based on an assembly produced by the *SLIC Assembler* using ABI data and the *ReAligner* gapping and alignment method. The accuracy of calls made using the *Mosaic* method are based on a *Phrap* assembly that takes *Phred* base calls and quality scores as input. For each method, I report the accuracy of about 680k consensus calls that are aligned with the *E. coli* GenBank entry.

Test results when no ambiguous calls are made by *Trace-EvidenceII* or *Majority* and lower-case *Mosaic* calls are interpreted as definitive calls are contained in Figure 9-8 and Table 9-2. In Figure 9-8, the number of correct calls per kb are graphed by amounts of coverage from *two to ten or more* aligned sequences. *Trace-EvidenceII* returns the same or higher accuracy than the other two methods at all coverages above two.

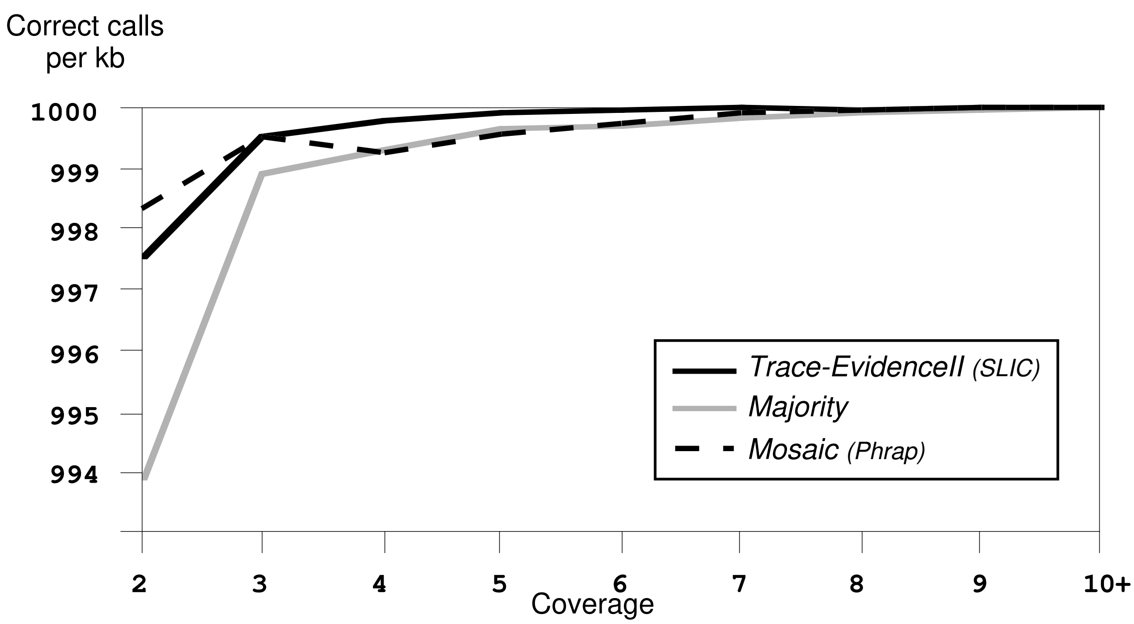


Figure 9-8. Consensus Accuracy with No Ambiguities. The accuracy of the *Trace-EvidenceII* consensus is equal to or higher than the other two methods for all coverages of three or more.

Table 9-2 lists a summary of the consensus accuracies for aligned columns with coverages of *four or more* reads and no ambiguous calls. The *Trace-EvidenceII* method produces one incorrect consensus call per 20 kb, compared to about one in 13 kb using the *Mosaic* method and about one per 4 kb for the *Majority* method. The differences between the results of *Trace-EvidenceII* and the *Mosaic* method with four or more sequences are statistically significant at the 95% confidence level using a paired one-tailed t-test.

Table 9-2. Consensus Accuracy with No Ambiguities Summary. The number of correct consensus calls per kb are listed for the *Trace-EvidenceII*, *Majority*, and *Mosaic* consensus calling methods. The accuracies listed are for columns with a coverage of four or more aligned reads.

Method	Correct per kb
Trace-EvidenceII	999.95
Majority	999.77
Mosaic	999.92

Test results when thresholds in *Trace-EvidenceII* and *Majority* allow ambiguities and lower-case *Mosaic* calls are considered ambiguous are contained in Figure 9-9 and Table 9-2. In Figure 9-9, results of the accuracy tests are graphed by amounts of coverage from *two to ten or more* aligned sequences. The graphs reveal a dramatically greater number of ambiguous calls made by *Mosaic*, especially at lower coverages. The number of incorrect is very similar for *Trace-EvidenceII* and *Mosaic* for coverages of three or more. At the 95% confidence level, the differences between correct calls and ambiguous calls made by *Trace-EvidenceII* and *Mosaic* are statistically significant using a paired one-tailed t-test.

Table 9-3 lists a summary of the consensus accuracies for aligned columns with coverages of *four or more* reads when ambiguous calls are allowed. The error rate for *Trace-EvidenceII* and *Mosaic* are extremely low; both make three or fewer errors per 100 kb. However, the number of ambiguous calls is far less for *Trace-EvidenceII*. Respectively, the *Majority* and *Mosaic* methods output 377 and 472 ambiguous calls per 100 kb, compared to 8 for *Trace-EvidenceII*. This is a significant reduction in the number of calls that must be examined manually when using *Trace-EvidenceII*. In addition, note that when ambiguous calls are allowed, *Trace-EvidenceII* is the only method that produces an accuracy of 99.98% that all but meets the National Human Genome Research Institute accuracy standard of 99.99% without hand editing (NHGRI 1998). In the results of using four or more aligned sequences with ambiguous calls, the differences between using *Trace-EvidenceII* and the *Mosaic* method are statistically significant using a paired one-tailed t-test at the 95% confidence level.

In Table 9-3, although the number of ambiguities is far less with *Trace-EvidenceII* than with *Mosaic*, the amount of error is slightly higher. This result suggests that there may be a trade-off between accuracy and ambiguities. However, when I sum results for coverages of eight or more, the error rate for both the methods is 0.01 per kb. Even with the equal error rate, *Mosaic* still outputs many more ambiguous calls; 120 per 100 kb compared to 1 per 100 kb for *Trace-EvidenceII*.

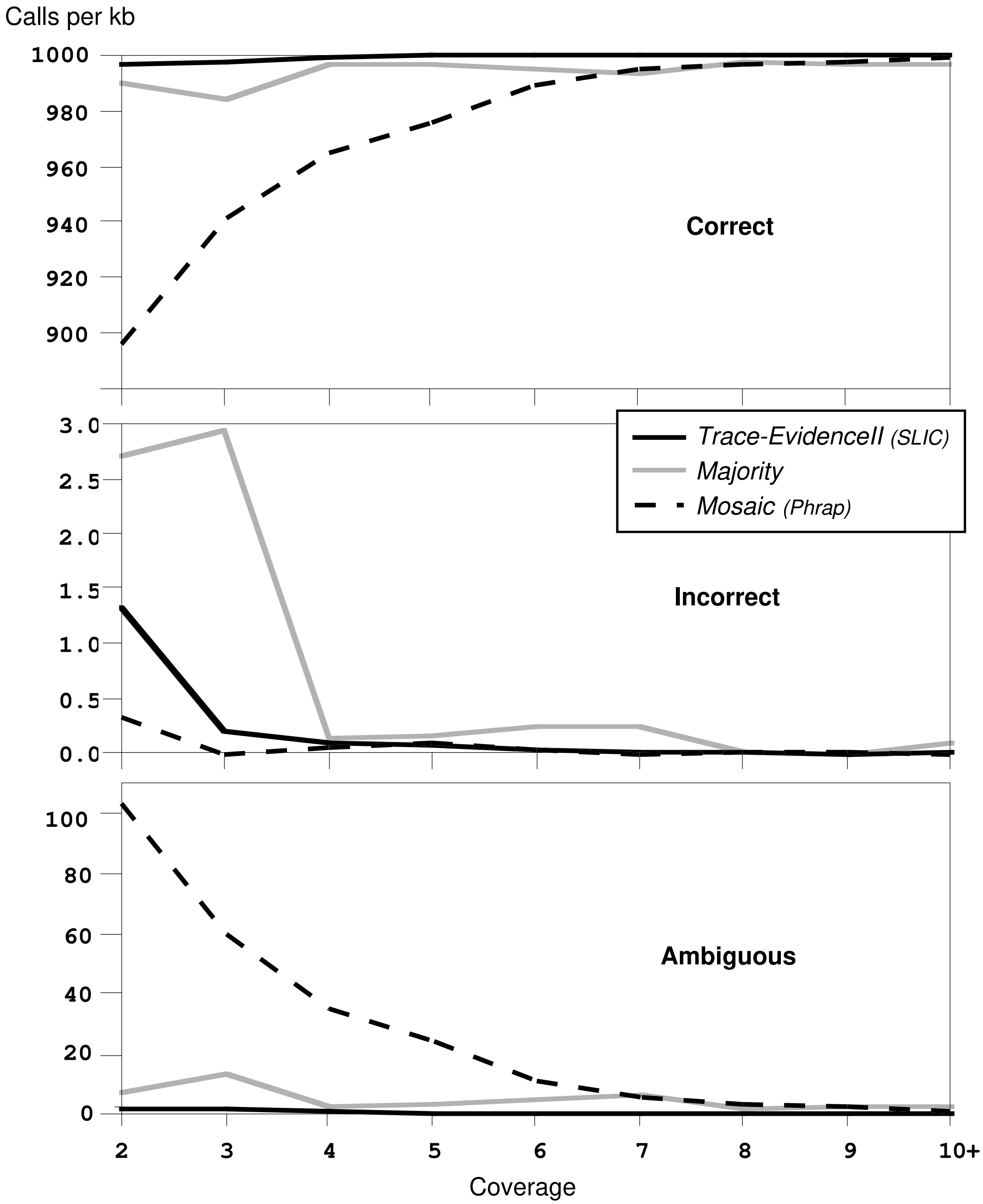


Figure 9-9. Consensus Accuracy with Ambiguities. *Phrap* returns far more ambiguities than either of the other two methods. At coverages of four or more, the number of incorrect consensus calls is nearly equal for all three methods.

Table 9-3. Consensus Accuracy with Ambiguities Summary. The number of correct, incorrect, and ambiguous consensus calls per kb are listed for the *Trace-EvidenceII*, *Majority*, and *Mosaic* consensus calling methods. The accuracies listed are for columns with a coverage of four or more aligned reads.

Method	Correct per kb	Incorrect per kb	Ambiguous per kb
Trace-EvidenceII	999.89	0.03	0.08
Majority	996.08	0.14	3.77
Mosaic	995.27	0.01	4.72

I examined *Phrap* consensus calls that are not in agreement with GenBank and find that errors usually occur due to the interaction of the *Mosaic* scheme with inaccurate base calls. In an aligned column, even though one or even the majority of the base calls is correct, an incorrect consensus call is made if the highest quality score is associated with a miscalled base. Figure 9-10 contains two examples, one in which the erroneous call is a gap and the other in which it is a base.

Many *Trace-EvidenceII* consensus errors occur when a nearly equal number of gaps and bases are in the aligned column. In general, if the sum of the weights of the bases exceeds the sum of the weights, the consensus is called as a base, and is called as a gap if the gap weight sum is greater than the base weight sum. Other errors are made by *Trace-EvidenceII* when the evidence for a spurious peak dominates the sum of the evidence for true peaks. This problem has been diminished by the refinement described earlier in Section 8.2.1, but has not been eliminated. Examples of gap and spurious peak errors are shown in Figure 9-11.

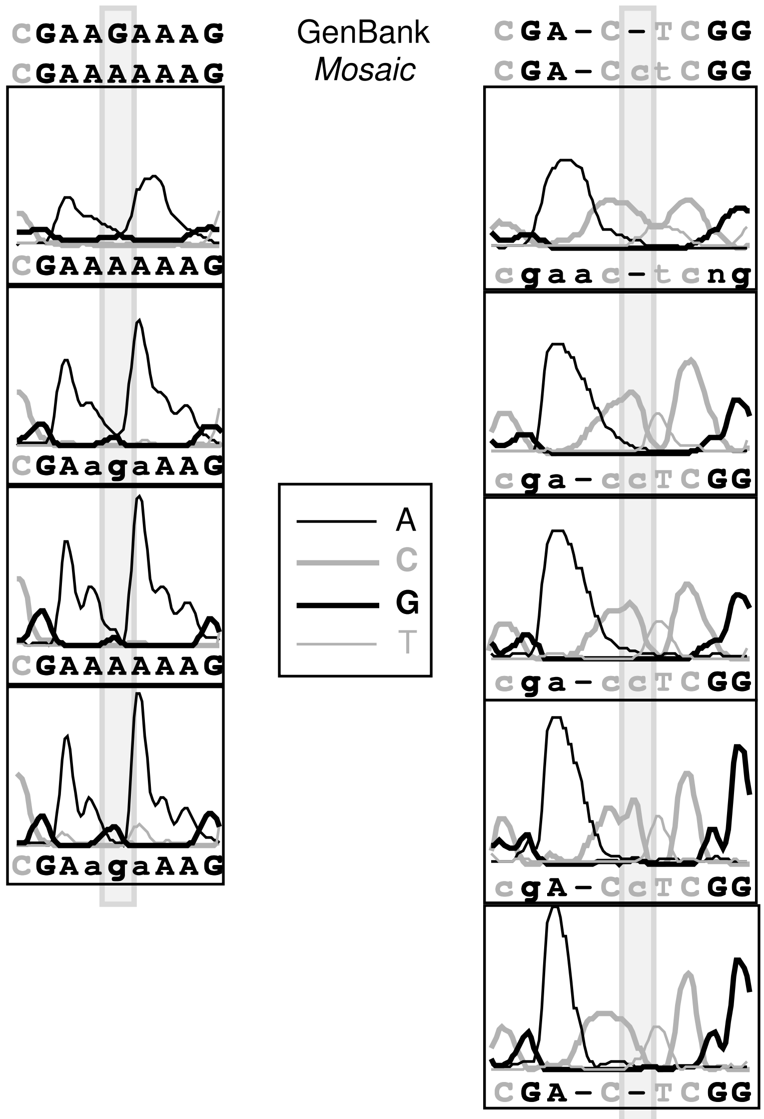


Figure 9-10. ***Mosaic*** **Consensus Errors.** *Mosaic* returns incorrect consensus calls when the highest quality base call is erroneous. (Actual data shown.)

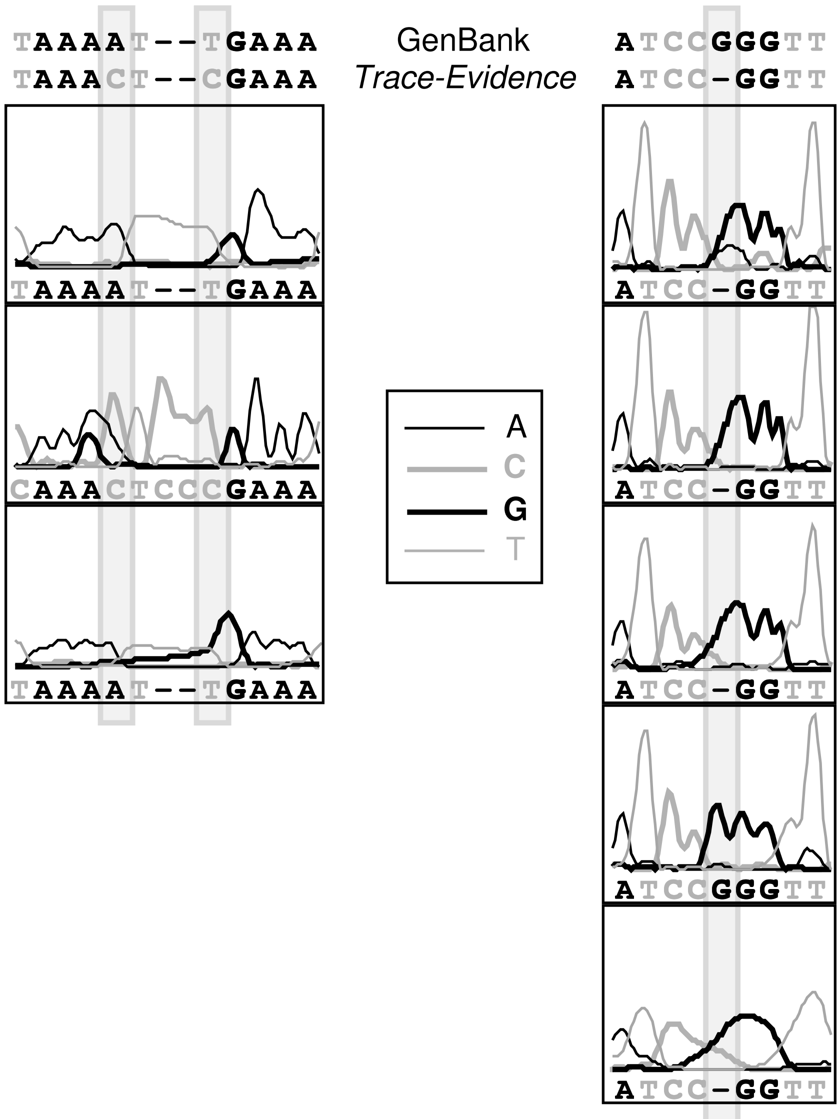


Figure 9-11. *Trace-EvidenceII* Consensus Errors. Spurious peaks in the left alignment and a missing base call in the right result in errors in *Trace-EvidenceII* consensus calls. (Actual data shown.)

9.5 Alignment

To compare the two *SLIC Assembler* alignment methods, *SeqManII* and *ReAligner* (Anson & Myers 1997), I report the amount of time to align, the number of conflicts in an alignment, and the consensus accuracy of alignments. First, I compare the time to align of the *SeqManII* and *ReAligner* methods. As can be seen in Table 9-4, on average over four *E. coli* segment assemblies, the *ReAligner* method takes just over two thirds the time it takes for *SeqManII*.

Table 9-4. Alignment Time. The time to align *E. coli* segments H, J, K, and L in minutes are listed in along with average times for the two alignment methods.

Method	H	J	K	L	Average
ReAligner	6.1	8.9	8.3	7.1	7.6
SeqManII	9	12.8	12.3	10.1	11.1

For the second comparison, I count the number of base calls that are in conflict with the consensus in the four *E. coli* segment assemblies. A higher-quality alignment should have fewer conflicts. Table 9-5 lists the number of conflicting and total base calls found in the comparison. (Gaps added by the methods are included in the base call counts.) The percent of conflicts is very similar for the two methods: 1.45% for *ReAligner* and 1.48% for *SeqManII*. I hypothesized that with the low error rates produced by the *SLIC Assembler*, even this small difference could have some effect on consensus accuracy. To confirm this belief, I compared consensus sequence accuracies of the two methods (Figure 9-12.) I find that the although the *ReAligner* consensus appears to contain fewer errors, the difference is not statistically significant.

Table 9-5. Alignment Conflicts. The total and conflicting number of base calls are shown for the two methods. Although the percent of conflicts is lower for *ReAligner*, the percentages are close.

Method	Number of Base Conflicts	Total Number of Bases	Percent Conflicts
ReAligner	89,153	6,141,734	1.45
SeqManII	91,075	6,135,870	1.48

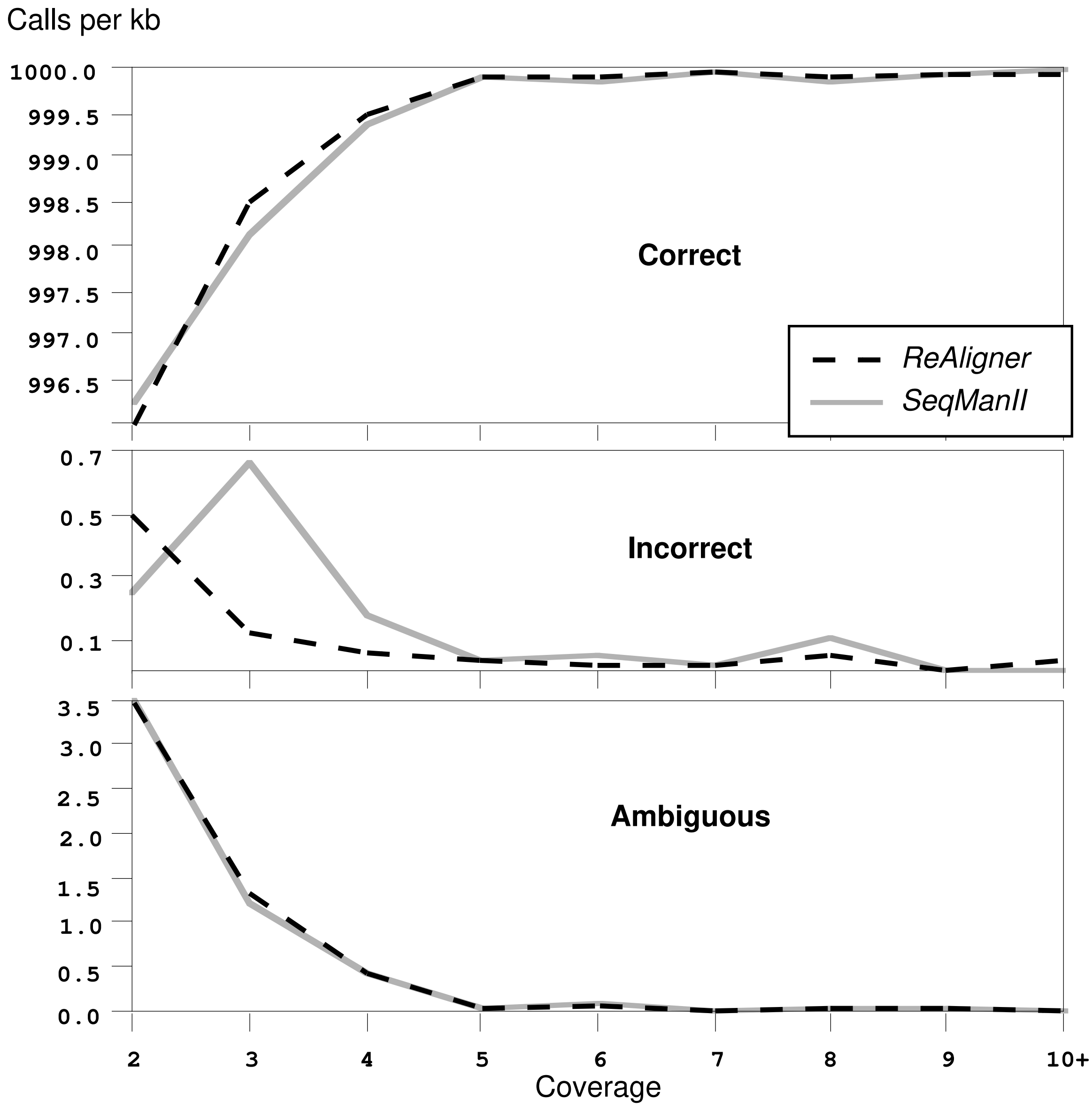


Figure 9-12. *ReAligner* and *SeqManII* Alignment Consensus Accuracy. The accuracy of 242 kb of consensus calls for segment J is graphed. The differences in the accuracies are not statistically significant.

I also observe that *ReAligner* tends to add more gaps than *SeqManII* in areas of high disagreement, resulting in longer alignments. Figure 9-13 illustrates this with a comparison of alignments produced by *ReAligner* and *SeqManII* for the same region of sequence. There is no disadvantage with using either method by this criterion since the consensus for both alignments is accurate compared to the GenBank entry.

a)

```
Consensus  ...CTTGGTGCTGGCGGTCA-G-AT------AG--CCCGCCAT...

           ...CTTGGTGCTGGCGGTCA-G-AT------AG--CCCGCCAT...
           ...CTTGGTGCTGGCGGTCA-G-AT------AG--CCCGCCAT...
           ...CTTGGTGCTGGCGGTCA-G-AT------NG--CCCGCCAA...
           ...TTNGGTGCTGGCGGTCA-G-AT------AG--CCCGCCAT...
                            ACATGAATACGCCAAGCTCCCGCCAT...
```

b)

```
Consensus  ...CTTGGTGCTGGCGGTCAGATAGCCCGCCAT...

           ...CTTGGTGCTGGCGGTCAGATAGCCCGCCAT...
           ...CTTGGTGCTGGCGGTCAGATAGCCCGCCAT...
           ...CTTGGTGCTGGCGGTCAGATNGCCCGCCAA...
           ...CTTGGTGCTGGCGGTCAGATAGCCCGCCAT...
                ACATGAATACGC-CA-AGCTCCCGCCAT...
```

Figure 9-13. *ReAligner* and *SeqManII* Alignments. Five sequences are aligned; the first four match well throughout their length, but the fifth one has a noisy 5′ end. a) *ReAligner* adds multiple gaps to the matching sequences to align them with the noisy end. b) *SeqManII* aligns by allowing more mismatches among the sequences. In either case, the consensus sequence generated is correct. (Actual data shown.)

Overall, since there does not seem to be any significant difference between the consensus accuracies of alignments produced by *ReAligner* and *SeqManII*, the preferred alignment method is *ReAligner* due to the advantage it realizes in shortened execution time.

9.6 Quality Scores

Accurate trimming is critical to the effectiveness of the *SLIC Assembler*. If trimming is too conservative, noisy ends may prevent elongation of a contig and trimming that is too aggressive will result in a loss of overlaps that could merge contigs. The *Trace-Quality Trim* algorithm employed in the preprocessing steps of the *SLIC Assembler* is based on quality scores associated with individual base calls. The most effective scores should have a strong correlation with the accuracy of base calls. To investigate the existence of such a correlation, I measured accuracy as a function of quality score. Results are graphed in Figure 9-14. The percent of correct base calls obviously rises along with the quality scores for scores up to about 20. Above 20, although the percent of correct base calls slowly increases, the range of scores is not significantly indicative of the expected accuracy of base calls.

Ideally, I would like to obtain a linear relationship between quality score and base-call correctness for the entire spectrum of quality scores. However, the low correlation for scores above 20 is not a problem for the *SLIC Assembler* since averaged quality scores are used only for thresholds in trimming. The thresholds I recommend are in the 8 to 16 range, well within the discriminating region of quality scores.

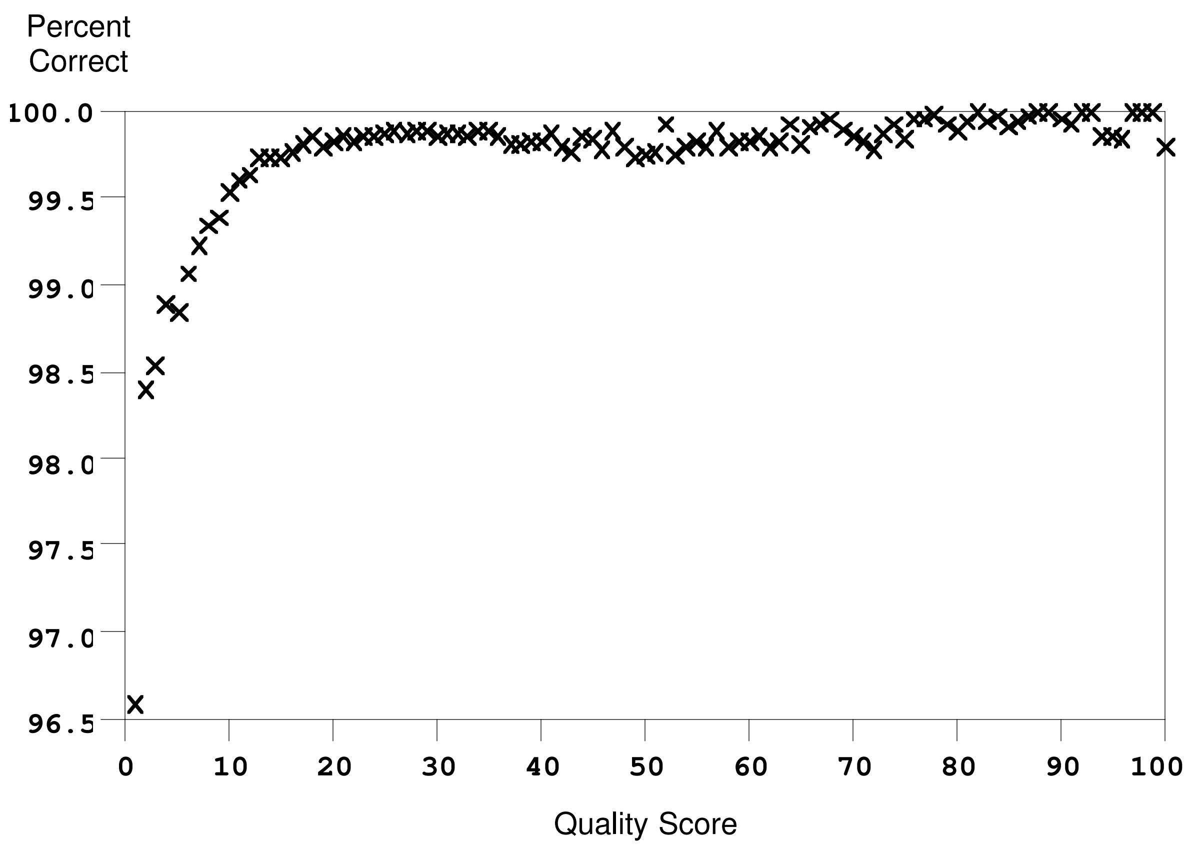


Figure 9-14. Accuracy as a Function of Quality Score. A definite correlation can be seen between quality scores and the percent of correct base calls for scores up to about 20.

9.7 Discussion and Summary

Overall, the *SLIC Assembler* produces high-quality fragment assemblies. It outperforms *SeqManII* in every aspect of this evaluation, and produces results similar or superior to *Phrap* in most ways. In particular, the assembly time and consensus accuracy are a proven strength of the *SLIC Assembler*. Not only does the consensus sequences produced by the *SLIC Assembler* have low error rates that compete with *Phrap*, they also contain far fewer low-confidence calls that must be examined manually.

The *SLIC Assembler* takes a simple approach to detecting repeats. The approach is to eliminate mer overlaps when the number of identical mers exceeds some fraction above the expected coverage. This method resulted in fewer false joins than *SeqManII*, but one more than *Phrap*. An important next step is to develop more sophisticated methods for avoiding false joins due to repeated regions.

Arguably, the most significant weakness of the *SLIC Assembler* is in its need to use trimmed fragment reads, resulting in fragmentation into multiple contigs. In unreported experiments using a lower trimming stringency with *SLIC*, even more contigs are produced since mismatches in noisy ends cause the pairwise similarity to fall below threshold. The *Phrap* assembler does do some virtual trimming, but only by adjusting base call quality scores in the context of all other reads in an assembly project. This allows the confirmation and utilization of all reasonable data. The *Trace-Quality Trim* used in preprocessing for the *SLIC Assembler* trims sequences without regard for other reads in the project. This is a significant drawback and is a problem for future work.

As part of the *SLIC Assembler* evaluation, I compared the *SeqManII* and *ReAligner* alignment algorithms. Although the number of conflicts in a *ReAligner* alignment shows a small decrease over the number in a *SeqManII* alignment, the decrease makes no significant difference in consensus accuracy as shown in Figure 9-12. However, due to its relatively shorter execution time, *ReAligner* is a better choice than *SeqManII* for aligning layouts in the *SLIC Assembler*.

Even though the quality scores produced by the *SLIC Assembler* are adequate for their use in the trimming, refinement of the score calculation is another subject for future work. In the analysis, only scores below about 20 are highly correlated to base call correctness. Since the preferred trimming thresholds are in that range, the quality scores work well for trimming. However, quality scores are also useful to apply to other problems in DNA sequencing. For example, many researchers want a reliable measure of the overall quality of their reads so that they can adjust laboratory procedures to increase the quality of data they produce.

The *SLIC Assembler* provides a fast and accurate system for assembling fragment reads. Improvements in the system will be realized by refining it to trim in the context of all sequence reads in a project, handle repeats in a more sophisticated manner, and produce quality scores that correlate linearly with base-call correctness.

Chapter 10

Computational Complexity of *SLIC*

In practice, the computational complexity of the *SLIC* layout algorithm is linear with respect to n, the number of fragment reads. The complexity analysis relies on the assumption that there is a practical upper bound to the size of the mer tags that depends neither on the number of fragment reads nor on the length of the actual sequence. In theory, the complexity may be analyzed as $O(n \log n)$, but in practice, the assumption is valid and the complexity of the algorithm is $O(n)$. I first discuss why I assume that the mer size may be considered constant.

I define the following symbols:

k	size of mer tag
n	number of fragment reads
s	length of the actual sequence
f	average length of a fragment read
c	average coverage
m	number of mer tags to choose per f bases
t	total number of mer tags to choose

I state the following:

$$n \equiv \frac{sc}{f}$$

$$s \equiv \frac{nf}{c}$$

$$t \equiv \frac{sm}{f}$$

A mer tag size of k provides 4^k possible unique tags since there are four DNA bases. I must have at least as many possible mer tags as the total number of tags I choose. Given this and a series of substitutions I find the following:

$$4^k \geq t$$

$$4^k \geq \frac{sm}{f}$$

$$4^k \geq \frac{nm}{c}$$

$$k \geq \log_4 \frac{nm}{c}$$

$$k \geq \log_4 n + \log_4 m - \log_4 c$$

$$k \geq \log_4 n + constant$$

At this point, it appears that the size of the mer tag, k, is a function of n, the number of fragments. Although this is true in theory, in practice there is an upper bound to k. The upper bound can be approximated by using a Poisson distribution to estimate the probability that a random k-mer occurs more than once in a sequence (Studier 1989). The probability that a random mer has exactly p occurrences is estimated by

$$P(p) = \frac{x^p e^{-x}}{p!} \text{ where } x = \left(\frac{s}{4^k}\right)$$

and the probability that a randomly chosen k-mer occurs two or more times is

$$P(\geq 2) = 1 - \left(P(0) + P(1)\right).$$

When I consider a genome ten times the size of *Human* and graph the probabilities for various sizes of mers (Figure 10-1) I see that the probability of two or more mer occurrences approaches zero near a mer size of 19. This can be used to estimate a reasonable upper bound on k.

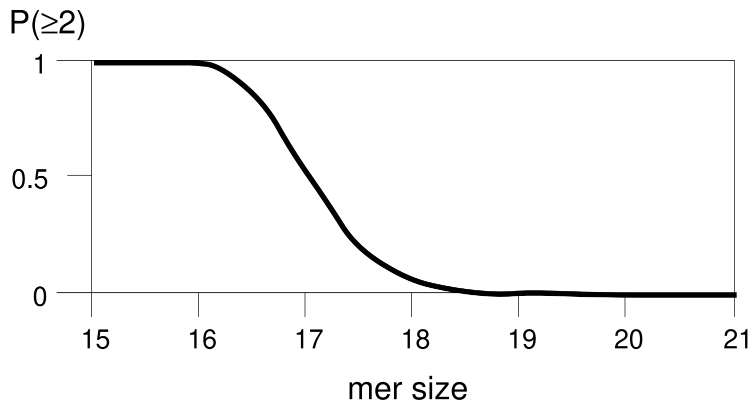


Figure 10-1. Mer Sizes for a Large Genome. The probability that a randomly chosen mer of a given size will occur two or more times in a genome containing 3×10^{10} base pairs. Mer sizes from 15 to 21 are graphed. The probability approaches zero at a mer size of 19.

An absolute upper bound on k is set by the length of fragment reads. At present, although some technologies produce reads in excess of 1000 base pairs, the usable portion is usually about 500 base pairs long. Clearly, the mer tag size must be less than this or the algorithm is useless for identifying overlaps. I can state that the absolute upper bound on the mer size is not set by the length of the actual sequence nor of the number of fragment reads, but rather on the length of fragment overlaps. In the degenerate case, the bound is approximately 500. I can therefore consider k, the size of mer tags, to be a constant in my analysis of the complexity of the algorithm. Given this assumption, an analysis of the complexity of individual steps as listed in Figure 10-2 is provided next.

Complexity	Step
O(constant)	1. Initialize all variables and structures
O(n)	2. Read sequences
O(n)	3. Count occurrences of mers in all fragment reads
O(n)	4. For each read, choose mer tags using mer counts
O(n)	5. For each read, if a mer is chosen as a tag for any read, choose it as a tag for the current read
O(n)	6. Make contigs

Figure 10-2. *SLIC* Computational Complexity. The computational complexity of each major step in the implementation of the layout algorithm is listed.

Step 1 is completed in constant time. The three major data structures used are the hash table for the mers, the list of fragment reads, and the list of contigs. Each of the entries in the hash table is an empty list of mers. The time to accomplish its initialization is therefore constant since the length of the table is dependent upon the size of the mer, which is constant. The fragment read and contig lists are both empty at this point, so the time to clear them is also constant.

In step 2, I read once through each of the fragments' base-call sequences determining its length. This iteration depends on the number of fragments and takes time proportional to n.

Step 3 requires a single read through all of the base-call sequences, counting occurrences of mers. To read the sequences takes time proportional to n. To count the mers, it is necessary to access a bucket and chain hash table. The size of a mer, k, is $x + y$. Recall that in the table, the integer value of an encoding of the first first x bases of the mer specifies a bucket. In the encoding, two bits are used to represent each base call; A, G, T, and C are represented by 00, 01, 10, and 11, respectively. Since two bits encode each base in the x-mer, the length of the table is 4^x. The last y bases in the mer specify a mer record in the chain. Indexing the hash table therefore takes place in constant time since I use the integer value of the encoding of the first x bases of the mer as the index, and the last y bases to specify a record in the chain. The length of the mer lists that form the chain of the hash table can be at most 4^y. Since y is constant, accessing a mer in the list is also constant. The overall complexity of this step is proportional to n.

During each of steps 4 and 5, I again scan through the base-call sequence of each of the fragment reads, accessing the hash table once for each mer that is contained in the sequence. The scans require time proportional to n and accessing the table is constant. In addition, for each new mer tag, I add onto the mer's list of associated fragment reads. I always add to the end of the list, so this step takes constant time. The time complexity for each of steps 4 and 5 is $O(n)$.

In step 6, I iterate through each of the lists in the hash table of mers, possibly making contigs with each of the fragment lists associated with mer tags. Since the greatest possible number of mer tags that can be chosen is 4^{x+y}, and $x + y$ is a constant, the number of mer tag lists that must be processed is constant. (Note that the number chosen is actually much less than 4^{x+y}). Making a contig with each of the mer tags lists is also constant. Recall that I set a threshold for the maximum count of mer occurrences allowed before a mer is marked as a repeat. This keeps the maximum length of any list at less than the threshold, so the lists are constant in length. (Recall that the threshold is a function of the depth of coverage of sequences, which is a constant usually about 6 to 10 sequences.) Since the lists' lengths are constant, the number of pairwise comparisons required for the similarity check is dependent on a constant. Making a new contig is constant in time and adding to or merging contigs is at most proportional to n. Overall, step 6 completes in time proportional to n.

I have shown that, in practice, the computational complexity of the *SLIC* algorithm is proportional to n, the number of fragment reads. The first step is completed in constant time, a series of scans through the reads in steps 2 to 5 each require linear time with respect to n, and constructing the contigs in step 6 is also linear. The analysis is based on the assumption that there is a practical upper bound to the size of mer that can be used. Note that although base-calling errors in the data set may result in production of an inferior assembly, errors do not affect the computational complexity of the algorithm.

Chapter 11

Additional Related Research

Research into computational methods for DNA sequencing has been ongoing since the advent of sequencing methods. Starting as early as the late 1970s, Rodger Staden published a series of papers describing software programs designed to analyze and manipulate sequence data (Staden 1986, 1984a, 1984b, 1982a, 1982b, 1980, 1979, 1978, 1977). Recent work by various researchers has progressed into methods that use artificial intelligence, genetic algorithms, and examination of fluorescent trace data. In this chapter I review research including methods for sequence assembly, base calling, and quality assessments. I also report observations about patterns found in trace data that may be useful.

11.1 Fragment Assembly

The *Phrap* assembly program was described in Chapter 8. Here I describe five other assembly methods: *TIGR*, *GAP*, *CAP2*, *Alewife*, and an approach that uses genetic algorithms.

11.1.1 *TIGR*

The *TIGR Assembler*, developed at the *Institute for Genomic Research*, was used to assemble the 1.8 mb *Haemophilus infuenzae* genome (Sutton *et al.* 1995). In the *TIGR* assembler, first a pairwise comparison of all reads in a data set identifies potential overlaps between reads. The pairwise comparison for n reads is a function of n^2. To speed up this step, rather than executing a full *Smith-Waterman* alignment (Smith & Waterman 1980) on each pair, an evaluation of the number of substrings common to both reads is performed. Only those pairs with a sufficient number of substrings in common are fully aligned and checked for similarity.

During the pairwise comparisons, putative repeated regions are identified in reads that have an overly abundant number of potential overlaps. Assembly of these reads is deferred until last and is carried out with a higher match stringency. In addition, distance constraints are used to help properly place sequences that have been identified as potential repeats. The distance between some pairs of reads containing a repeat may be known. In that case, if a read containing a repeat already has its paired read in a contig, then only regions at the given distance from the paired read are considered for overlap with the repeat read.

The consensus sequence is generated by examining the base calls in an aligned column. A *profile* is produced that indicates the total number of reads in the column with calls of *A*, *C*, *G*, *T*, and *gap*. Allowable consensus calls include: upper and lower-case bases (*A*, *C*, *G*, *T*, *a*, *c*, *g*, and *t*), *gap*, two-base ambiguity codes (*r*, *k*, *s*, *w*, *m*, and *y*), and *n*. A small set of rules determines the consensus call based on the profile. If the largest component in the profile is greater than two-thirds the total, an upper-case base or *gap* is called. If two non-gap components are significant, a lower-case ambiguity is called. If the largest component is between one-half and two-thirds the total, a lower-case base or *gap* is called. In all remaining cases, a lower-case *n* is called. Lower case letters indicate when the confidence in the call is not high. In addition, if the confidence in a *gap* call is not high, the call following the gap is lower-case. Using this method, lower-case letters in the consensus sequence pinpoint calls that require examination by human editors.

11.1.2 *GAP*

The *Genome Assembly Program* (*GAP*) uses a greedy approach to fragment assembly (Bonfield, Smith, & Staden 1995). First a companion program, *PREGAP* assigns quality values to base calls and trims reads to remove poor quality data and vector. Then, one at a time, each read is added to a contig if it is sufficiently similar. First the read is compared against all other reads for matching subsequences. An alignment is then made between the read and each other read for which it has a match, and the quality of each alignment is noted. The read is overlapped with the read with which it has the highest-quality alignment (over some minimum threshold). If the read aligns sufficiently well with more than one read, then after it is overlapped with the best-aligning sequence, it is used to join the contigs of the two matching reads.

The *GAP* program can also use distance constraints. To use the constraints, the placement of a read may be restricted to an approximate given distance from an *anchor* read. If an above threshold alignment can be found for the read within the specified region, the read is added to the contig. The program also allows for a variety of *tags* to be used. One tag is used to label repeated regions in reads. A region that is tagged as having a repeat is not used in searching for subsequence matches but is aligned during assembly.

11.1.3 *CAP2*

CAP2 is an improved version of the *Contig Assembly Program* (*CAP*) (Huang 1996). The assembly methods developed for *CAP* are at the core of the *ABI Prism AutoAssembler*. This program assembles reads in three phases: 1) overlap detection, 2) contig formation, and 3) consensus sequence determination. In the first phase, a filter identifies which pairs of reads are likely to overlap. Between all such pairs, the match similarity is computed using a variant of the *Smith-Waterman* alignment algorithm (Smith & Waterman 1980). Error vectors are computed and used to evaluate the strength of overlaps and to identify chimeric reads. In the second phase, a preliminary assembly is formed by joining pairs of reads in decreasing order of pairwise similarity. Inconsistent overlaps in the preliminary assembly are used to identify and partition repeated reads. The partitions are used to establish the final assembly. In phase three, the contigs are fully aligned and the consensus is computed.

11.1.4 *Alewife*

A method called *Alewife* that is reported to run in time less than n^2 is described on the *Whitehead Institute/MIT Genome Sequencing Project* web site (MIT 1998). The basic idea of this assembler is similar to the idea I use in my layout algorithm. The *Alewife* method uses 25-

mers as tags to identify overlaps in fragment reads. By using a hash table, *Alewife* is reported to assemble fragments at a rate that is proportional to $n \log n$. It thus appears to be based on an algorithm capable of performing sequence assembly more quickly than programs using n^2 algorithms. However, the linear algorithm I have developed, *SLIC*, performs sequence assembly proportional to n, and is theoretically faster than *Alewife*. Since the details of the MIT algorithm are not yet published, other possible differences between my layout algorithm and *Alewife* are not known.

11.1.5 Genetic Algorithms

A unique approach to sequence assembly is to use genetic algorithms (Parsons & Johnson 1995, Parsons 1993). In this work, the layout of a data set with n fragments is represented as a bit string with $n * k$ bits (k bits per fragment) where $n \leq 2^k$. For each fragment, the integer value of its k bits identifies the position of the read in the overall layout. Added to the $n * k$ bit string is an additional k bits used to identify the starting fragment in the layout. The total length of the bit string is then $(n + 1) * k$. Standard operators are used while the mapping from bit strings to layouts ensures that legal layouts are generated. The fitness function evaluates the strength of overlaps between adjacent reads in the layout. Two variants of the fitness function were developed, one is proportional to n and the other to n^2. Depending on which function is chosen, the overall time to run the algorithm is then proportional to n^2 or n^3.

In initial work, although the genetic algorithm approach produced good layouts for small data sets under 20 kb, the large search space often led to the failure to find good solutions for larger data sets. Later work produced acceptable results with a 35 kb data set, but this size is still far from the size of sequencing projects undertaken by large sequencing centers and is certainly far less than most whole genomes. Another drawback of the method is its failure to deal with highly conserved repeat regions. With the presence of repeats, the resulting consensus tends to be shorter than expected, indicating a compression of the repeated regions.

11.2 Base Calling

Commercial companies, such as ABI, that perform one of the most commonly used base calling methods for fluorescently labeled sequences have not disclosed their algorithms. However, some research for base calling on fluorescent data has been performed and reported. A system that is rapidly gaining popularity, challenging ABI base calling, is *Phred* (Ewing *et al.* 1998). Tibbetts, Bowling, and Golden (1994) have studied using neural networks for base calling. Giddings *et al.* (1993) describe a system that uses an object-oriented filtering system.

A base caller that reports error rates lower than ABI software is *Phred* (Ewing *et al.* 1998). Given ABI trace files, *Phred* calls bases and assigns error probabilities to each base call. Base calls are determined with a four-phase method: 1) determine peak locations, 2) identify observed peaks, 3) match locations to observed peaks, and 4) call unmatched observed peaks.

The first phase is based on the premise that in a local area, base call peaks are fairly evenly spaced. The expected spacing of the peaks is used to predict the number and locations of base calls. In phase two, observed peaks are found by scanning each of the four traces, looking for concave down regions. The area of each concave down region found is computed and compared to preceding observed peaks. If the area is within 10% of the last ten observed peaks and within 5% of the immediately preceding peak, it is identified as an observed peak.

As observed peaks are matched with predicted locations in phase three, some peaks may be eliminated and others may be broken into two or more identical base calls. This phase is the most complicated and progresses through three stages. In the first stage, obvious matches are made. In the second stage dynamic programming is used to align other peaks with locations. Finally, unmatched observed peaks that appear to be real are matched to locations. At the end of phase three, there may still be peaks in troublesome regions that seem to be actual peaks but still have not been assigned locations. In phase four these peaks are assigned locations if they meet specific criteria.

Tibbetts, Bowling, and Golden build their neural network base calling system on previous work that conditions the raw data output from sequencers (Golden, Torgersen & Tibbetts 1993). In the conditioned traces, peaks are narrower, better separated, and have less crossover. This conditioned data is used to form the inputs for a neural-network base caller.

The base caller takes three inputs: the primary sequence determinant (related to the intensity and probable identity of the peak), intensity relative to the 5' base, and separation relative to the 5' base. There are four outputs, one for each of the four possible bases. They tested their system on data generated by a Du Pont *Genesis 200 U*; the base calling output by the Du Pont system is 90-95% accurate. The neural network base caller achieved 95% accuracy without using relative intensities and separations, and reached 99% accuracy using these features. The limitation in the use of these features is that they are specific to chemistries and conditions.

Giddings *et al.* (1993) also preprocess raw trace data before base calling to make peaks more distinct and disjoint. Base calling, using this processed data, consists of three major steps: 1) identify peaks, 2) determine which peaks are likely to represent fragments, and 3) assign confidence values to base calls. The base calling system iterates over steps 2 and 3, removing unlikely peaks after each iteration. This cycle continues for a given number of iterations, or until all confidence values are adequate or unchanged. For step 2, filtering likely bases, they suggest that multiple characteristics of trace data may be used. They chose three: peak height, peak spacing, and peak width. (For peak spacing, they use the value as computed globally over an entire run.)

Gidding's system requires a substantial number of parameters. Some are automatically set according to characteristics of the run – this is an appealing component of this system. Many other (more than ten) parameters must be set by a user. This is ordinarily an undesirable feature in a system, but the authors claim that it is not difficult to find a reasonable combination, and that once set, the parameters need not be changed unless experimental conditions change. One of the strengths of the system is that the filters in step 2 are completely modular; additional filters may be developed and added readily.

11.3 Quality Assessment

Through the last decade, interest in quality assessments for DNA sequencing has been increasing (Ewing & Green 1998, Richterich 1998, Li *et al.* 1997, Bonfield & Staden 1995, Naeve *et al.* 1995, Lawrence & Solovyev 1994, Lipshutz *et al.* 1994, Khurshid & Beck 1993, and Chen & Hunkapiller 1992).

Currently, the most widely accepted quality standard is that which is produced by *Phred* (Ewing & Green 1998). Each base call made by *Phred* is assigned a quality score, Q, that reflects the estimated probability of error, $P(e)$, of the base call. The interpretation of the quality score is $Q = n10 \log_{10}(P(e))$. For example, with $Q = 20$, the *Phred* quality score estimates that the probability of error is 1 in 100. The most effective criteria used in assigning quality scores examine data in a window surrounding the base call of interest. The four most influential measurements used are: the ratio of the largest to the smallest peak spacing in a window of seven, the ratio of the highest to the lowest peak intensity in a window of seven, the ratio of the highest to the lowest peak intensity in a window of three, and the number of bases separating the base of interest from the nearest base call of N.

The method used to combine the measurements into a quality score must be calibrated by using a known sequence and a training set of reads. This constrains the usefulness of the quality scores since the method must be calibrated for each machine that produces traces with different characteristics. For example, ABI has recently released its new sequencer, the 3700, and the developer of *Phred* predicts it will be months before their quality assessment method is calibrated to work with the 3700 (Wade 1999).

Another method that uses a known sequence and training reads in assigning error probabilities was developed by Lawrence and Solovyev (1994). This method uses discriminant analysis to combine 25 trace characteristics into an error probability for a base call. Models for three types of base calling errors are developed: miscalls, insertions, and deletions. For each of the three models, an iterative process adds characteristics one at a time in decreasing order of significance until the discriminating power of the model fails to increase.

11.4 Patterns in Trace Data

With dye-terminating chemistries, the intensity signal reflects the number of dideoxynucleotides that are inserted during replication. The likelihood of inserting a chain-terminating dideoxynucleotide rather than a chain-elongating deoxynucleotide is influenced by adjacent 5' bases in the growing fragment. The result is that patterns in intensities are seen. This observation led to some of the (unreported) experiments I performed using neural networks for consensus calling (Chapter 7). In some experiments I input information about one or more bases to either side of the base of interest in hopes of increasing accuracy. In these experiments, I found no increase in accuracy over input representations that include only single-column data. Here I relate some research involving the discovery of patterns in DNA trace data.

Perkin-Elmer (1995) reports patterns they have detected in fluorescent-dye labeled trace data. Parker *et al.* (1995) systematically varied pairs of neighboring bases to discover patterns in peak intensities. They found that peak height can often be predicted by 5' neighboring bases. They report predictions based on one, two, and three bases 5' to the base of interest. Golden, Torgersen, and Tibbetts (1993) used neural networks to extract patterns in peak intensities. They report that different proteins seem to have complex systems of rules dependent on 5' bases that determine whether a dideoxynucleotide or a deoxynucleotide will be added to a growing fragment. They found that the bases one, two, three, and ten bases 5' to the base of interest have the most influence on the dideoxynucleotide and deoxynucleotide competition.

Golden, Torgersen, and Tibbetts (1993) have also found patterns in the separations (rate of migration) of adjacent bases. They show that separations are dependent on a dideoxynucleotide and its 5' neighboring bases. Particular patterns in neighboring bases may form secondary structures that result in the predictable variance in rate of migration of adjacent bases.

Some researchers have reported that the separations also vary globally over an entire run of fragments. Giddings *et al.* (1993) established that the separations are best fit by a negative quadratic function – the fragments tend to migrate progressively faster before slowing in the latter part of the run. Bouriakov and Mayhew (1995) have found that the rate of migration over a run steadily increases and then drops off sharply after about the last 100 bases.

11.5 Summary

Research into computational methods for DNA sequencing is a growing and exciting endeavor. Some approaches to problems are well-studied and mature, while other, more innovative approaches are also under investigation. For example, research into applying AI techniques to problems in molecular biology has become fairly common in academic settings. One example of an AI approach that is widely used is GRAIL, a system that uses neural networks to search for genes (Uberbacher & Mural 1991). Unfortunately, many solutions have yet to make a strong appearance in freely available or commercial software. In the future, more novel solutions should make their way into common usage as research moves away from conventional methods.

Chapter 12

Conclusions

The goal of my work is to develop computational methods for increasing the speed and accuracy of DNA fragment assembly. As advances in technology result in the production of increasing amounts of sequencing data in decreasing amounts of time, it is imperative that computational methods are developed that allow data analysis to keep pace. In this dissertation, I presented several methods that improve the speed and accuracy of fragment assembly and that lay a foundation for further research.

12.1 Contributions

I contribute effective computational methods for DNA fragment assembly in two primary directions. One thrust is in the development and application of a descriptive representation of fluorescent traces. The representation, *Trace-Class*, is useful for trimming poor-quality data from the ends of fragment reads, in methods for determining the consensus sequence of an assembly of aligned reads, and in assessing data quality. The other thrust is in developing an algorithm for fragment read layout, *SLIC* (*Sequence Layout into Contigs*), that in practice runs in time linear with the number of reads. I incorporated *SLIC* into a total package for fragment assembly, the *SLIC Assembler*, that includes refinements of trimming and consensus algorithms.

12.1.1 *Trace-Class* Representation

Fluorescent traces output by sequencing machines are the key to modern DNA sequencing. The traces are scanned to call the bases for individual reads. Previous representations of the traces include: a raw representation as a sequence of dye intensities, the collapse of the intensities into a base call, and the visual representation of a 2-D graph. I defined a new representation, *Trace-Class*, that captures the shape and intensity of the traces (Chapter 3). In particular, the representation recognizes the height and definition of peaks and valleys in traces. In assembly processes, it is useful to have access to these characteristics. If not, any decisions that require examination of the traces must be made by hand; incorporating characteristics into computational methods allows many decisions to be made without human intervention.

12.1.2 *Trace-Class Trim*

Most assembly programs require that the poor quality data be trimmed from the ends of fragment reads before assembly. Without trimming, inclusion of the poor data may result in fragmented contigs. When attempting to align overlapping reads, the poor data on the ends of a read may have insufficient similarity to allow overlapping and aligning the read in a contig.

One common previous method for end-trimming, *N-Trim*, merely counts the number of no-calls (*N*s) in a window of bases. Ends with above-threshold number of *N*s in a window are trimmed from the read. The new method I introduced, *Trace-Class Trim* examines the number of high quality *characteristic classes*, as defined by the *Trace-Class* representation, in a window (Chapter 4). Assemblies produced from reads trimmed with *Trace-Class Trim* are of higher quality than those produced after *N-Trim* or no trimming. The key to the success of *Trace-Class Trim* is in its use of descriptive trace information via the *Trace-Class* representation.

12.1.3 *Trace-Evidence* Consensus

The specific goal of DNA sequencing is to determine the sequence of bases in a fragment of DNA. In a fragment assembly, this sequence is the consensus of the aligned reads in the assembly. As such, the consensus must be accurate if the specific goal of sequencing is to be attained. Inaccurate sequences can have significantly adverse effects on sequence analysis as miscalled, inserted, or deleted bases change the characteristics of predicted proteins.

Automatically generated consensus sequences usually contain some errors and a number of ambiguous (low-confidence) calls. Ambiguous calls add to the work of sequencing since these calls must be examined and resolved by hand. Reducing the number of ambiguous calls and eliminating errors are eminently worthwhile objectives for developers of automatic methods for consensus calling. My new method for consensus calling, *Trace-Evidence*, makes substantial progress toward these objectives (Chapter 6). A standard previous method, *Majority*, simply counts the number of each kind of base call in an aligned column and applies a threshold in calling a base, gap, or ambiguity. In contrast, the *Trace-Evidence* method takes into account underlying trace characteristics by summing evidence as supplied by the *Trace-Class* representation.

When compared to the previous *Majority* method for consensus calling, *Trace-Evidence* consensus sequences are substantially more accurate while making significantly fewer ambiguous calls, especially at low coverages (numbers of aligned sequences). Reducing the needed coverage means a decrease in costs, since every step in sequencing adds to the overall expense. A typical coverage of 6 to 10 may not be required when consensus calls are highly accurate with fewer aligned sequences. As with *Trace-Class Trim*, the strength of *Trace-Evidence* consensus is in its use of trace information provided by the *Trace-Class* representation.

12.1.4 Neural-Network Consensus

I contribute to research in the application of neural networks to problems in molecular biology by training neural networks to make consensus calls (Chapter 7). In my work, I trained neural networks with five different input representations to output one of the four bases or a gap as a consensus call. One network uses only base call information in its input representation, while the other four include trace characteristics using the *Trace-Class* representation, trace peak intensities, or both. Again I find that using trace characteristics significantly improves consensus accuracy. The network that uses only base calls as inputs produces lower consensus accuracies.

12.1.5 *SLIC Assembler*

One major contribution of my work is in the development of an algorithm for fragment layout, *SLIC* (*Sequence Layout into Contigs*), that, in practice, runs in linear time with the number of fragment reads (Chapter 8). In addition, building on the layout algorithm, I have developed a comprehensive package for fragment assembly, the *SLIC Assembler*. The package is still evolving and is expected to be available commercially through DNASTAR Inc. in the future.

Quality Scores

Used in various methods in assembling with the *SLIC Assembler* are quality scores (Chapter 8). I assign a single trace quality score to each base call in a read based on the shape and intensity of the underlying trace data. In some cases, the quality score is relative to the intensity of the entire trace, while in others it is appropriate to make the score relative only to the portion of the trace that underlies a base call.

The purpose of the quality score is to define a measure of the confidence of a base call. For some applications, the scores are averaged over a window centered on the base of interest so that the local quality of the traces can be evaluated. The quality scores may range from 0 to 100. The scores up to about 20 show a strong correlation with correctness of base calls. This relationship indicates that using scores under 20 to evaluate quality is reasonable and justified.

Trace-Quality Trim

For use in the *SLIC Assembler*, I developed a second approach to end trimming, *Trace-Quality Trim*, that also examines trace characteristics (Chapter 8). Rather than using *Trace-Class* scores, I use the averaged quality scores (with intensities relative to the entire trace). A threshold of the averaged quality values specifies where trimming occurs; the largest portion of the read at or above threshold is retained while the rest is trimmed. This trimming method uses more information than *Trace-Class Trim*, making trimming decisions less arbitrary.

Trace-EvidenceII Consensus

I refined the *Trace-Evidence* consensus for incorporation into the *SLIC Assembler*. The refined consensus method, *Trace-EvidenceII*, does not use the *Trace-Class* scores for evidence as *Trace-Evidence* does (Chapter 8). Instead, it uses the single quality score based on the shape and intensity of traces as is used in the refined *Trace-Quality Trim*. However, the intensities used for trimming are relative to an entire trace, and the intensities used for consensus calling are relative only to the local area of the base call. Before summing, the evidence scores are weighted by the quality of the trace according the the averaged quality values (identical to those used in trimming).

Another difference between *Trace-Evidence* and *Trace-EvidenceII* is that while summing the evidence based on the quality scores, adjustments are made to reflect the detection of a spurious peak in a column of bases. This helps to prevent spurious peak evidence from overwhelming the evidence for true peaks and results in fewer ambiguous calls.

The improvements in the *Trace-EvidenceII* consensus method result in consensus sequences with extremely high accuracies and substantially fewer ambiguous calls. The accuracy of consensus calls made when the coverage is four or more sequences is 99.989%. This all but meets the standard of 99.99% set by the National Human Genome Research Institute (NHGRI 1998).

SLIC Algorithm

Most existing assemblers perform pairwise comparisons of reads, resulting in layout times proportional to n^2, where n is the number of reads. An important contribution of my work is in the development of an algorithm, *SLIC*, that in practice executes in time proportional to n (Chapter 8). No other existing assembler claims a linear layout time. As scientists move toward sequencing larger DNA fragments and whole genomes, execution times may represent the difference between a possible and impossible task. With a large number of fragment reads, an n^2 algorithm may take years to execute, compared to days for a linear time algorithm.

In general, the layouts produced by *SLIC* are of good quality. In evaluative tests, the times to assemble and the layouts produced using *SLIC* compare favorably. Observations of repeat handling capabilities show that *SLIC* is superior or similar to other methods.

12.1.6 Commercial Availability

I have implemented several of my new methods into commercially available versions of DNASTAR Inc.'s *SeqManII* fragment assembly program. The original version of *SeqMan* used the *N-Trim* approach to end trimming and the *Majority* method to make consensus calls. For *SeqManII*, I replaced *N-Trim* with *Trace-Class Trim* for trimming, and for consensus calling I replaced the *Majority* method with *Trace-Evidence*. For the latest release of DNASTAR Inc.'s suite of applications, *Lasergene99*, I implemented *Trace-Quality Trim* and enhanced the *Trace-Evidence* method with the *Trace-EvidenceII* improvements. Also available as a feature of *SeqManII* in *Lasergene99* is a visual display of quality values and averaged quality values as described in Chapter 8. The *SLIC Assembler* is expected to be released by DNASTAR Inc. as commercial software in the future.

12.2 Limitations and Future Work

The work I have completed represents a real improvement in methods for DNA fragment assembly. Although the work has limitations, it still provides a solid ground for building more sophisticated solutions to unsolved problems.

12.2.1 Quality Scores

The *Trace-Class* representation was originally developed for use with neural networks. I wanted to give the network a good description of the traces that captured visual characteristics. As it turned out, when I applied the scores to other problems, the scores needed to be combined into a single score. Although evaluated empirically, the method for combining scores is fairly arbitrary. It is much more straightforward to assign a single score at the outset. In later work, this is what I did in determining the quality scores used in the *SLIC Assembler* ancillary methods. The glaring weakness in the quality scores is their inability to discriminate among the correctness of base calls when their values exceed about 20. There is a great deal of interest in the development of quality scores that accurately reflect base call correctness and I believe that this is an important direction for my work to take in the future.

12.2.2 Neural-Network Consensus

One major limitation with the use of neural network is that they must be trained on a data set with characteristics similar to those of the data that they will later process. The problem is in forming a training set; a training set contains not only instances of the problem, but must also include the correct classification for the instance. A researcher may not have access to data that allows them to form training sets with both an adequate number of instances as well as the correct classification. Alternately, using networks trained by developers who do have access to such data is not useful unless the characteristics of the data are the same as that which is to be analyzed. An undertaking for future work is to develop a method for training accurate neural networks using only small amounts of training data for which the correct classification may be only an estimate.

12.2.3 *SLIC Assembler*

Evaluations of the current implementation of the *SLIC Assembler* dictate several paths for improvements. Limitations that must be addressed include issues with end trimming, repeat handling, consensus accuracy, and memory use.

End Trimming

One limitation of the *SLIC Assembler* is that it requires relatively error-free data for successful assembly. I help to ensure the use of accurate data by trimming poor quality data from the ends of sequences before assembly. However, the ends in a read are trimmed without regard to other reads in the data set. The result is that the trimming is arbitrary in the context of the whole data set. Some ends are trimmed too excessively and overlaps with other reads are lost. Some are trimmed too conservatively and noisy ends prevent aligning with other reads due to below threshold similarity in overlapping regions.

To address this problem, I first plan to use data sets with varying amounts of artificially introduced noise to characterize how base-calling errors affect assembly. I can then investigate possible solutions. One idea is to use the mer counts made in the first pass of the *SLIC* algorithm as a possible source of information for trimming in the context of other reads. Regions of sequences that have no mers in common with other reads are are likely to be noisy, chimeric, or non-overlapping with other reads. In any case, it may be safe to ignore these regions of reads during assembly. I plan to try this approach is in future work.

Repeat Handling

Repeat handling in the *SLIC Assembler* is primitive; it assumes a mer is in a repeat if its occurrence is excessive given the expected coverage. This approach is used by other existing assemblers, but it is not sufficient to distinguish between repetitive and non-repetitive regions. The amount of coverage can vary dramatically in an assembly due to inconsistencies inherent in the fragment preparation and sequencing process. The repeat handling approach also does not address how the correct placement for the putative repeats might be determined. Correct handling of sequences containing repeats is a significant undertaking for future work. I plan to research graph-traversal and other algorithmic solutions for cases in which the *SLIC* algorithm fails when assembling repeated regions.

Consensus Accuracy

Errors in the consensus of a *SLIC* assembly appear when the evidence supplied by the reads is not in total agreement. I believe that many errors can be avoided by not just examining a single column of trace information, but also the information surrounding columns. If there is good agreement among most reads aligned in a local area, but a minority do not correspond, reads associated with the minority should be discounted. Although the error rate is already quite low with *Trace-EvidenceII*, I think it is worthwhile to pursue this idea for increasing accuracy.

Memory Use

In the present implementation of the *SLIC* algorithm, all reads and ancillary information are kept in memory. Clearly, this is a significant detriment to assembling large fragments and whole genomes. Fluorescent sequencing machines produce individual fragment reads of about 500 usable base pairs. Assuming a coverage of five, the number of fragment reads is almost 10,000 to cover a small bacterium, is about 120,000 for the yeast genome, and numbers over one million for *C. elegans*. To execute *SLIC* on these numbers of reads would require about 40 MB, 500 MB, and 5 GB of memory, respectively. While to many users, 40 MB, and even

500 MB is not a problem, others will not have access to sufficient RAM.

My aim for future work on memory usage is that, given the length of the overall fragment and the number of sequences, *SLIC* will automatically segment processing to work within memory and disk space limitations. *SLIC* requires space not only for the sequences and ancillary information, but also for several large tables of information collected during processing. Both of these space requirements will be addressed through appropriate segmenting of the *SLIC* algorithm. The basic idea is that layout can be accomplished though iterative accumulation of interim results that can be saved on disk and merged later.

12.3 Final Remarks

Computational methods for DNA fragment assembly have been evolving for a number of years. Through a great deal of dedicated research over time, considerable improvements in the effectiveness of the methods have been made. The work I have presented here provides additional improvements and can serve as a foundation for further study in developing better solutions to problems in fragment assembly.

Appendix A

Glossary of Biological Terms

This appendix contains a glossary of biological terms as used in this dissertation. Further details may be found in textbooks such as *Modern Genetic Analysis* (Griffiths et al. 1998).

5′ end

The end of a DNA molecule that originates with the sugar ring containing a 5′ carbon atom. Often used to refer to the start of a fragment read.

3′ end

The end of a DNA molecule that terminates with the sugar ring containing a 3′ carbon atom. Often used to refer to the end of a fragment read.

ABI

Applied Biosystems Inc. A division of Perkin-Elmer that produces the most widely used DNA sequencing machines.

adenine

See **base**.

ambiguous

A base call that is any combination of A, C, G, and/or T.

assembly

Determining the layout of fragment reads by aligning their overlapping regions of base calls.

base

One of four molecules, (A), cytosine (C), guanine (G), and thymine (T), that when bonded to phosphate and sugar make up a deoxynucleotide.

base call

The base associated with a particular sequence of fluorescent-dye intensities.

base calling

Interpreting output from sequencing machines to call the sequence of bases for a fragment of DNA.

base pair

A pair of complementary bases. A is complementary to T and C is complementary to G.

chimera

Two erroneously joined fragments of DNA from discontiguous sources.

complementary bases

Bases that bond in a pair. A is complementary to T and C is complementary to G.

consensus

Refers to either a consensus call or a consensus sequence.

consensus call

The most likely base given an aligned column of base calls.

consensus calling

Determining the most likely base given an aligned column of base calls.

consensus sequence

The most likely sequence of bases given an alignment of sequences.

contaminant sequence

A sequence that is not from the organism of interest.

contig

Contiguous segments of DNA formed by aligning overlapping regions of reads.

coverage

The number of aligned sequences.

cytosine

See **base**.

deoxyribonucleic acid

A molecule composed of a chain of deoxynucleotides. Commonly called DNA.

deoxynucleotide

A base bonded to phosphate and sugar.

dideoxynucleotides

A modified deoxynucleotide that terminates elongation during DNA replication.

DNA

see **deoxyribonucleic acid**.

DNA sequencing

Determining the sequence of bases in a fragment of DNA.

dye contamination

Excess dye that migrates with fragments during sequencing resulting in a spurious high fluctuation in the trace.

electrophoresis

See **gel electrophoresis.**

false join

Reads erroneously overlapped due to repeated or near-repeated sequences.

fluorescent dye

Dyes used to label fragments of DNA. Each of four dyes labels one of the four dideoxynucleotides. When excited by a laser, each of the four dyes emits a distinct spectrum of light.

fluorescent-dye traces

See **traces**.

fragment assembly

See **assembly**.

fragment layout

See **layout**.

fragment read

See **read**.

gap

Used in an alignment to indicate that either a base is missing from the sequence of base calls (a *deletion*) or that a false base has been called in one or more of the aligned sequences (an *insertion*).

gel electrophoresis

A process in which fragments of DNA migrate through a gel when a voltage is applied.

genes

Regions in DNA that enocde proteins and other products.

genome

A molecule of DNA that is the genetic material for an organism.

guanine

See **base**.

Human Genome Project

A project to find all the genes in human DNA and to discover the functions of their proteins.

kb

See **kilobase.**

kilobase

A measure of the length of DNA fragments; 1000 bases.

layout

The order and offset of reads in a fragment assembly.

no-call

The call made by a base caller when it must make a call, but the correct base cannot be determined.

polyacrylamide gel

A gel that allows electropheretic separation of DNA fragments.

primer

A short fragment of DNA used to prime replication.

read

The sequence of base calls for a fragment of DNA.

reading a trace

The detection and recording of the intensities of fluorescent dyes by a sequencing machine as fragments pass a detector.

repeat

A subsequence of DNA that occurs more than once in a DNA fragment or genome.

repeated sequence

See **repeat**.

sequence alignment

Aligning two or more sequences of bases such that the number of mismatches is minimized.

sequence assembly

See **assembly**.

sequence layout

See **layout**.

sequence read

See **read**.

shotgun sequencing

A strategy for sequencing DNA that involves first creating a random (shotgun) library of small fragments from a whole genome and then sequencing the small fragments.

thymine

See **base**.

traces

Sequences of the intensities (amounts) of fluorescent dyes advancing through time.

vector sequence

A fragment of DNA used to carry and replicate a fragment of interest.

whole-genome shotgun sequencing

See **shotgun sequencing**.

Appendix B

Trace-Class Score Pseudocode

In this appendix I present pseudocode for calculating the *Trace-Class* scores described in Chapter 3. Four functions are included:

Assign_Trace-Class_Scores,

Assign_Strong_Med_Scores,

Assign_Med_Weak_Scores, and

Assign_Weak_Scores.

Assign_Trace-Class_Scores

Parameters

```
int   pt_array[];        /* Array of intensity data points */
int   num_pts;           /* Number of data points in pt_array */
int   base_pt;           /* pt_array index of base call location */
int   max_pt;            /* Max intensity of all four traces */
int   *strong_peak;      /* Pointer to Strong peak score */
int   *med_peak;         /* Pointer to Medium peak score */
int   *weak_peak;        /* Pointer to Weak peak score */
int   *strong_valley;    /* Pointer to Strong valley score */
```

```
int   *med_valley;        /* Pointer to Medium valley score */
int   *weak_valley;       /* Pointer to Weak valley score */
```

Algorithm

```
if (Assign_Strong_Med_Scores(pt_array, num_pts, base_pt,
            max_pt, &strong_peak, &med_peak, &strong_valley,
            &med_valley) == false) {

   if (Assign_Med_Weak_Scores(pt_array, num_pts, base_pt,
            max_pt, &med_peak, &weak_peak, &med_valley,
            &weak_valley) == false) {

   Assign_Weak_Scores(pt_array, num_pts base_pt, max_pt, &weak_peak,
            &weak_valley);
   }
}
```

Assign_Strong_Med_Scores

```
/* Returns true if strong peak or valley found */
```

Parameters

```
int   pt_array[];         /* Array of intensity data points */
int   num_pts;            /* Number of data points in pt_array */
int   base_pt;            /* pt_array index of base call location */
int   max_pt;             /* Max intensity of all four traces */
int   *strong_peak;       /* Pointer to strong peak score */
int   *med_peak;          /* Pointer to medium peak score */
int   *strong_valley;     /* Pointer to strong valley score */
int   *med_valley;        /* Pointer to medium valley score */
```

Local Variables

```
int   curr_pt;            /* pt_array index of current point */
int   prev_slope;         /* Slope between previous point and curr_pt */
int   next_slope;         /* Slope between curr_pt and next point */
int   peak_pt;            /* pt_array index of peak */
int   valley_pt;          /* pt_array index of valley */
int   left_extreme_pt;    /* pt_array index of change to left of
                             peak_pt or valley_pt */
int   right_extreme_pt;   /* pt_array index of change to right of
                             peak_pt or valley_pt */
int   extreme_array[];    /* Indices in pt_array where slope sign
                             changes; used to find left_extreme_pt and
                             right_extreme_pt */
int   idx;                /* Index into extreme_array */
int   extreme_peak_idx;   /* Index of peak_pt pt_array index in
                             extreme_array */
int   extreme_valley_idx; /* Index of valley_pt  pt_array in
                             extreme_array */
float distance_adj;       /* Score multiplier based on distance of
                             peak_pt and valley_pt from base_pt */
float height_adj;         /* Score multiplier based on relative
                             intensities of peak_pt and valley_pt with
                             max_pt */
```

Algorithm

```
idx = 1;
extreme_array[idx] = 1;       /* use first point if no changes */
idx = idx + 1;
peak_pt = -1;
valley_pt = -1;
```

```
for curr_pt = 2 to num_pts - 1 {
   prev_slope = pt_array[curr_pt] - pt_array[curr_pt - 1];
   next_slope = pt_array[curr_pt + 1] - pt_array[curr_pt];

   /* if change in sign of slope, peak or valley found */
   if (prev_slope * next_slope < 0) {
      if (prev_slope > 0) {            /* peak found */
         if (abs_value(curr_pt - base_pt) < abs_value(peak_pt - base_pt)) {
            peak_pt = curr_pt;         /* closer peak found */
            extreme_peak_idx = idx;
         }
      }
      else {                           /* valley found */
         if (abs_value(curr_pt - base_pt) <
                                    abs_value(valley_pt - base_pt)) {
            valley_pt = curr_pt;       /* closer valley found */
            extreme_valley_idx = idx;
         }
      }
      extreme_array[idx] = curr_pt;
      idx = idx + 1;
   }
}

extreme_array[idx] = num_pts; /* use last point if no changes */
idx = idx + 1;
```

```
if (peak_pt <> -1) {
   left_extreme_pt = extreme_array[extreme_peak_idx - 1];
   right_extreme_pt = extreme_array[extreme_peak_idx + 1];

   *strong_peak = 100 * (pt_array[peak_pt] - (pt_array[left_extreme_pt]
               + pt_array[right_extreme_pt]) / 2) / pt_array[peak_pt];
   *med_peak = 100 - *strong_peak;

   distance_adj = (num_pts - abs_value(peak_pt - base_pt)) / num_pts;
   height_adj = pt_array[peak_pt] / max_pt;

   *strong_peak *= distance_adj * height_adj;
   *med_peak *= distance_adj * height_adj;
}

if (valley_pt <> -1) {
   left_extreme_pt = extreme_array[extreme_valley_idx - 1];
   right_extreme_pt = extreme_array[extreme_valley_idx + 1];

   *strong_valley = 100 * (pt_array[valley_pt] - (pt_array[left_extreme_pt]
               + pt_array[right_extreme_pt]) / 2) / pt_array[valley_pt];
   *med_valley = 100 - *strong_valley;

   distance_adj = (num_pts - abs_value(valley_pt - base_pt)) / num_pts;
   height_adj = (1 - pt_array[valley_pt] / max_pt);

   *strong_valley *= distance_adj * height_adj;
   *med_valley *= distance_adj * height_adj;
}

return (peak_pt <> -1 or valley_pt <> -1);
```

Assign_Med_Weak_Scores

```
/* Returns true if medium peak or valley found */
```

Parameters

```
int   pt_array[];        /* Array of intensity data points */
int   num_pts;           /* Number of data points in pt_array */
int   base_pt;           /* pt_array index of base call location */
int   max_pt;            /* Max intensity of all four traces */
int   *med_peak;         /* Pointer to medium peak score */
int   *weak_peak;        /* Pointer to weak peak score */
int   *med_valley;       /* Pointer to medium valley score */
int   *weak_valley;      /* Pointer to weak valley score */
```

Local Variables

```
int   curr_pt;           /* pt_array index of current point */
int   prev_slope;        /* Slope between previous point and curr_pt */
int   next_slope;        /* Slope between curr_pt and next point */
int   max_change;        /* Amount of max change between slopes */
int   max_change_pt;     /* Data point where max_change occurs */
Bool  in_peak;           /* True when in slopes decreasing */
Bool  in_valley;         /* True when in slopes increasing */
int   peak_pt;           /* pt_array index of peak */
int   valley_pt;         /* pt_array index of valley */
int   left_extreme_pt;   /* pt_array index of change to left of
                            peak_pt or valley_pt */
int   right_extreme_pt;  /* pt_array index of change to right of
                            peak_pt or valley_pt */
int   extreme_array[];   /* Indices of data points with curvature
                            changes; used to find left_extreme_pt and
                            right_extreme_pt */
```

```
 int   idx;                /* Index into extreme_array */
 int   extreme_peak_idx;   /* Index of peak_pt in extreme_array */
 int   extreme_valley_idx; /* Index of valley_pt in extreme_array */
 float distance_adj;       /* Score multiplier based on distance of
                              peak_pt and valley_pt from base_pt */
 float height_adj;         /* Score multiplier based on relative
                              intensities of peak_pt and valley_pt with
                              max_pt */
```

Algorithm

```
idx = 1;
extreme_array[idx] = 1;       /* first point */
idx = idx + 1;
max_change = 0;
max_change_pt = 1;
in_peak = false;
in_valley = false;
peak_pt = -1;
valley_pt = -1;

for curr_pt = 2 to num_pts - 1 {
   prev_slope = pt_array[curr_pt] - pt_array[curr_pt - 1];
   next_slope = pt_array[curr_pt + 1] - pt_array[curr_pt];
   if (prev_slope > next_slope) {      /* peak, decreasing slope */
      if (in_peak) {                   /* still in peak */
         if (prev_slope - next_slope > max_change) {
            max_change = prev_slope - next_slope; /* change is greater */
            max_change_pt = curr_pt;
         }
      }
```

```
    else if (in_valley) {            /* were in peak, now in valley */
      if (abs_value(max_change_pt - base_pt) <
                     abs_value(peak_pt - base_pt)) {
        peak_pt = max_change_pt;     /* save closer peak */
        extreme_peak_idx = idx;
      }
      extreme_array[idx++] = max_change_pt;
      max_change = 0;
      in_valley = false;             /* no longer in valley */
      in_peak = true;                /* now in peak */
    }
  }
  else if (prev_slope > next_slope) {    /* valley, increasing slope */
    if (in_valley) {                     /* still in valley */
      if (next_slope - prev_slope > max_change) {
        max_change = next_slope - prev_slope; /* change is greater */
        max_change_pt = curr_pt;
      }
    }
    else if (in_peak) {              /* were in valley, now in peak */
      if (abs_value(max_change_pt - base_pt) <
                     abs_value(peak_pt - base_pt)) {
        valley_pt = max_change_pt;     /* save closer valley */
        extreme_valley_idx = idx;
      }
      extreme_array[idx] = max_change_pt;
      idx = idx + 1;
      max_change = 0;
      in_peak = false;               /* no longer in peak */
      in_valley = true;              /* now in valley */
```

```
      }
   }
}

if (in_peak) {
   if (valley_pt <> -1) {
      if (abs_value(max_change_pt - base_pt) <
                           abs_value(peak_pt - base_pt)) {
         peak_pt = max_change_pt;      /* closer valley found */
         extreme_peak_idx = idx;
      }
      extreme_array[idx] = max_change_pt;
      idx = idx + 1;
   }
}
else if (in_valley) {
   if (peak_pt <> -1) {
      if (abs_value(max_change_pt - base_pt) <
                           abs_value(valley_pt - base_pt)) {
         valley_pt = max_change_pt;    /* closer valley found */
         extreme_valley_idx = idx;
      }
      extreme_array[idx] = max_change_pt;
      idx = idx + 1;
   }
}

extreme_array[idx] = num_pts; /* last point */
idx = idx + 1;
```

```
if (peak_pt <> -1) {
   left_extreme_pt = extreme_array[extreme_peak_idx - 1];
   right_extreme_pt = extreme_array[extreme_peak_idx + 1];

   *weak_peak = 100 * (max(pt_array[left_extreme_pt],
               pt_array[right_extreme_pt]) - pt_array[peak_pt]) /
               max(pt_array[left_extreme_pt], pt_array[right_extreme_pt]);
   *med_peak = 100 - *med_peak;

   distance_adj = (num_pts - abs_value(peak_pt - base_pt)) / num_pts;
   height_adj = pt_array[peak_pt] / max_pt;

   *strong_peak *= distance_adj * height_adj;
   *med_peak *= distance_adj * height_adj;
}
if (valley_pt <> -1) {
   left_extreme_pt = extreme_array[extreme_valley_idx - 1];
   right_extreme_pt = extreme_array[extreme_valley_idx + 1];

   *weak_valley = 100 * (pt_array[valley_pt] -
               min(pt_array[left_extreme_pt],
               pt_array[right_extreme_pt])) / pt_array[valley_pt];
   *med_valley = 100 - *med_valley;

   distance_adj = (num_pts - abs_value(valley_pt - base_pt)) / num_pts;
   height_adj = (1 - pt_array[valley_pt] / max_pt);

   *strong_valley *= distance_adj * height_adj;
   *med_valley *= distance_adj * height_adj;
}

return (peak_pt <> -1 or valley_pt <> -1);
```

Assign_Weak_Scores

Parameters

```
int    pt_array[];         /* Array of intensity data points */
int    base_pt;            /* pt_array index of base call location */
int    max_pt;             /* Max intensity of all four traces */
int    *weak_peak;         /* Pointer to weak peak score */
int    *weak_valley;       /* Pointer to weak valley score */
```

Local Variables

```
int    prev_slope;         /* Slope between previous point and curr_pt */
int    next_slope;         /* Slope between curr_pt and next point */
float  height_adj;         /* Score multiplier based on relative
                              intensities of peak_pt and valley_pt with
                              max_pt */
```

Algorithm

```
prev_slope = pt_array[base_pt] - pt_array[base_pt - 2];
next_slope = pt_array[base_pt + 2] - pt_array[base_pt];

if (prev_slope > next_slope) {          /* peak, decreasing slope */
   *weak_peak = 100;

   height_adj = (1 - pt_array[base_pt] / max_pt);
   *weak_peak *= height_adj;
}
else if (prev_slope < next_slope) {    /* peak, increasing slope */
   *weak_valley = 100;

   height_adj = (1 - pt_array[base_pt] / max_pt);
   *weak_valley *= height_adj;
}
```

Appendix C

End-Trimming Pseudocode

In this appendix I present pseudocode for trimming poor data with *Trace-Class Trim* as described in Chapter 4 and the *Trace-Quality Trim* described in Chapter 8.

Trace-Class Trim

```
/* Trims the 3' end of a sequence read. */

/* Returns base_array index of first base to trim. */
```

Parameters

```
 char  base_array[];     /* Array of base calls */
 int   num_bases;        /* Number of bases in base_array */
 int   trace_array[4][]; /* Array of traces for each of the 4 bases */
 int   max_poor;         /* Max number of poor_classes allowed in window
*/
 Bool  poor_classes[];   /* Array position 1=SP, 2=MP, 3=WP, 4=WV, 5=MV,
                              6=SV, true if defined as poor_class */
 int   window_size;      /* Number of bases in a window */
```

Local Variables

```
int   base_idx;         /* Index into base_array */
int   curr_class;       /* Characteristic class for trace of curr_base,
                               1=SP, 2=MP, 3=WP, 4=WV, 5=MV, 6=SV */
int   window_array[];   /* Rolling window array of poor classes, set to 1
                             if classified as a poor class & 0 otherwise */
int   num_poor;         /* Number of poor_classes in current window */
int   window_idx;       /* Index into window_array */
```

Algorithm

```
base_idx = num_bases;   /* start at 3' end */
num_poor = 0;
clear(window_array);    /* set all window positions to 0 */

/* Fill the first window, then continue counting number of poor classes
   until num_poor <= max_poor */
While (base_idx > 0 && (num_poor > max_poor or   /* too many poor */
         base_idx >= num_bases - window_size)) { /* filling 1st window */
   num_poor= num_poor - window_array[window_idx];
   curr_class = get_class(base_idx, bases_array, trace_array);
   If (poor_classes[curr_class] == true) {
      window_array[window_idx] = 1; /* record occurrence of poor class */
      num_poor = num_poor + 1;
   }
   Else window_array[window_idx] = 0;

   window_idx = (window_idx + 1) mod window_size;
   --base_idx;
}
base_index = base_idx + window_size + 1;
return base_index;
```

Trace-Quality Trim

```
/* Trims 3' end and 5' ends of a sequence read. */
```

Parameters

```
char bases_array[];    /* Array of base calls */
int  num_bases;        /* Number of bases in base_array */
int  trace_array[4][]; /* Array of traces for each of the 4 bases */
int  threshold;        /* Regions above this quality threshold are ok */
int  *end3_base;       /* Pointer to base_array index of 3' trim
                          location */
int  *end5_base;       /* Pointer to base_array index of 5' trim
                          location */
```

Local Variables

```
int  Q;               /* Quality score for curr_base */
int  curr_base;       /* Index of current base in base_array */
int  curr_seq_length; /* Length of potential ok subsequence */
int  curr_end3_base;  /* base_array index of potential 3' trim location */
int  curr_end5_base;  /* base_array index of potential 5' trim location */
int  seq_length;      /* Length of longest ok subsequence so far */
```

Algorithm

```
seq_length = 0
curr_seq_length = 0

/* Scan bases from 3' to 5', keeping track of the longest ok subsequence
found so far */
For curr_base = 1 to num_bases {
   Q = get_quality(curr_base, traces);
   If Q >= threshold {        /* base Q meets threshold */
```

```
      curr_seq_length = curr_seq_length + 1;
      curr_end3_base = curr_base;
   }
   Else {                    /* found a base with Q below threshold */
      /* potential ok subsequence is longer */
      If (curr_seq_length > seq_length) {
         end5_base = curr_end3_base - curr_seq_length + 1;
         end3_base = curr_end3_base;
         seq_length = curr_seq_length;
      }
      curr_seq_length = 0;  /* reset length of potential ok subsequence */
   }
}

If (curr_seq_length > seq_length) {    /* ok subsequence goes to 3' end */
   *end5_base = curr_end3_base - curr_seq_length + 1;
   *end3_base = curr_end3_base;
}
```

Appendix D

SLIC Pseudocode

In this appendix I present type definitions, variables, data structures, function prototypes, and pseudocode followed by a detailed description of the *SLIC* layout algorithm.

Type Definitions

Name	Type		Description
`StreamT`	`string`		Base call sequence, encoded with 2 bits per base (A = 00, C = 11, G = 01, T = 10)
`MerT`	`StreamT`		Encoded mer
`FrgRec`	`Record`		Fragment read information
	`int`	`length`	Read length
	`StreamT`	`stream`	Base call sequence of read
	`int`	`ctgListIdx`	Index in a `CtgListT`
	`End`		
`FrgListT`	`FrgRec List`		List of `FrgRec`
`MerFrgRec`	`Record`		Information on a fragment read that contains a given mer
	`int`	`frgIdx`	Index of a `FrgRec` in a `FrgListT`
	`int`	`merOffset`	Offset of mer in a `FrgRec.stream`

	`End`	
`MerFrgListT`	`MerFrgRec List`	List of `MerFrgRec`
`MatchListT`	`MerFrgListT List`	List of `MerFrgListT`
`MerRec`	`Record`	Mer information
	`MerT mer`	Mer
	`int count`	Count of `mer` occurrences in all fragment reads
	`Boolean repeat`	`True` if the mer is a putative repeat
	`MerFrgListT`	`merFrgList` List of `MerFrgRec`
	`End`	
`MerListT`	`MerRec List`	List of `MerRec`
`MerTableT`	`MerListT List`	Hash table of mers, hashed on first *x* bases in mer, chained by last *y* bases in mer; chains are of type `MerListT`
`ScoreRec`	`Record`	Score information
	`MerT mer`	Encoded last *y* bases of mer
	`int merOffset`	Offset of mer in a `FrgRec.stream`
	`int score`	Mer score
	`End`	
`CtgFrgRec`	`Record`	Information on a fragment read that is in a contig
	`int frgIdx`	Index of a fragment read in a `FrgListT`
	`int offset`	Offset of fragment read in a contig
	`End`	
`CtgFrgListT`	`CtgFrgRec List`	List of `CtgFrgRec`, ordered by offset
`CtgListT`	`CtgFrgListT List`	List of `CtgFrgListT`, one `CtgFrgListT` per contig

Variables and Data Structures

Name	Type	Description
newCtgFrg	CtgFrgRec	New CtgFrgRec
ctgList	CtgListT	List of contigs
ctgListIdx	int	Current index of a contig in a CtgListT
found	Boolean	True if found
frgList	FrgListT	List of FrgListRec for fragment reads in the dataset
frgRec	FrgRec	Current FrgRec
maxCount	int	Count threshold for the number of occurrences of a mer in the dataset
matchListT	MatchFrgListT	List of MerFrgListTs.
mer	MerT	Current mer
newMerFrg	MerFrgRec	New MerFrgRec
merFrgRec	MerFrgRec	Current MerFrgRec
merRec	MerRec	Current MerRec
merTable	MerTableT	Hash table of all mers of length $x + y$ that occur in the dataset
numFrgs	int	Number of fragment reads in frgList
numMers	int	Number of mer tags to choose for each fragment read
merOffset	int	Offset of a mer in a fragment read
scoreRec	ScoreRec	Current ScoreRec
tagScoreRec	ScoreRec	ScoreRec of highest scoring mer for current partition

Functions

Prototype	Returns	Description
addToList(List, Item)	List	Adds Item to List
assignScore(MerT, int)	ScoreRec	Assigns score for MerT
betterScore(ScoreRec,	Boolean	Returns true if 1st ScoreRec is better than 2nd for given partition
getCtgListIdx(MerFrgListT)	int	Returns index of any contig in the list, 0 if

		none
`getMerRec(MerTableT, MerT)`	`MerRec`	Returns `MerRec` for `MerT` from `MerTableT`
`getNextFrgRead()`	`FrgRec`	Returns next fragment read, stored in a `FrgRec`
`getNextMer(FrgRec, int)`	`MerT`	Returns next mer from `FrgRec` for current partition; if `int` is 0, partition is entire 5' to 3' read
`inMerFrgList(MerFrgListT, int)`	`Boolean`	Returns `true` if the read indexed by `int` is in `MerFrgListT`
`listLen(List)`	`int`	Returns length of `List`
`makeMatchFrgLists(MerFrgListT)`	`MatchFrgListT`	Divides a `MerFrgListT` into separate lists in `MatchFrgListT` such that all fragment overlaps within each list have above-threshold similarity
`mergeCtgs(CtgListT, int, int)`	`CtgListT`	Merges contigs indexed by the two `ints`
`newCtgFrgRec(CtgFrgListT, int)`	`CtgFrgRec`	Makes a new `CtgFrgRec`
`newCtg(CtgListT, int)`	`int`	Makes a new contig, returns its index in a `CtgListT`
`newMerFrgRec(int, int)`	`MerFrgRec`	Makes a new `MerFrgRec`
`oneCtg(MerFrgListT)`	`Boolean`	Returns `true` if all in the `MerFrgListT` are in the same contig

Figure D-1 charts the three main data structures and their relationships.

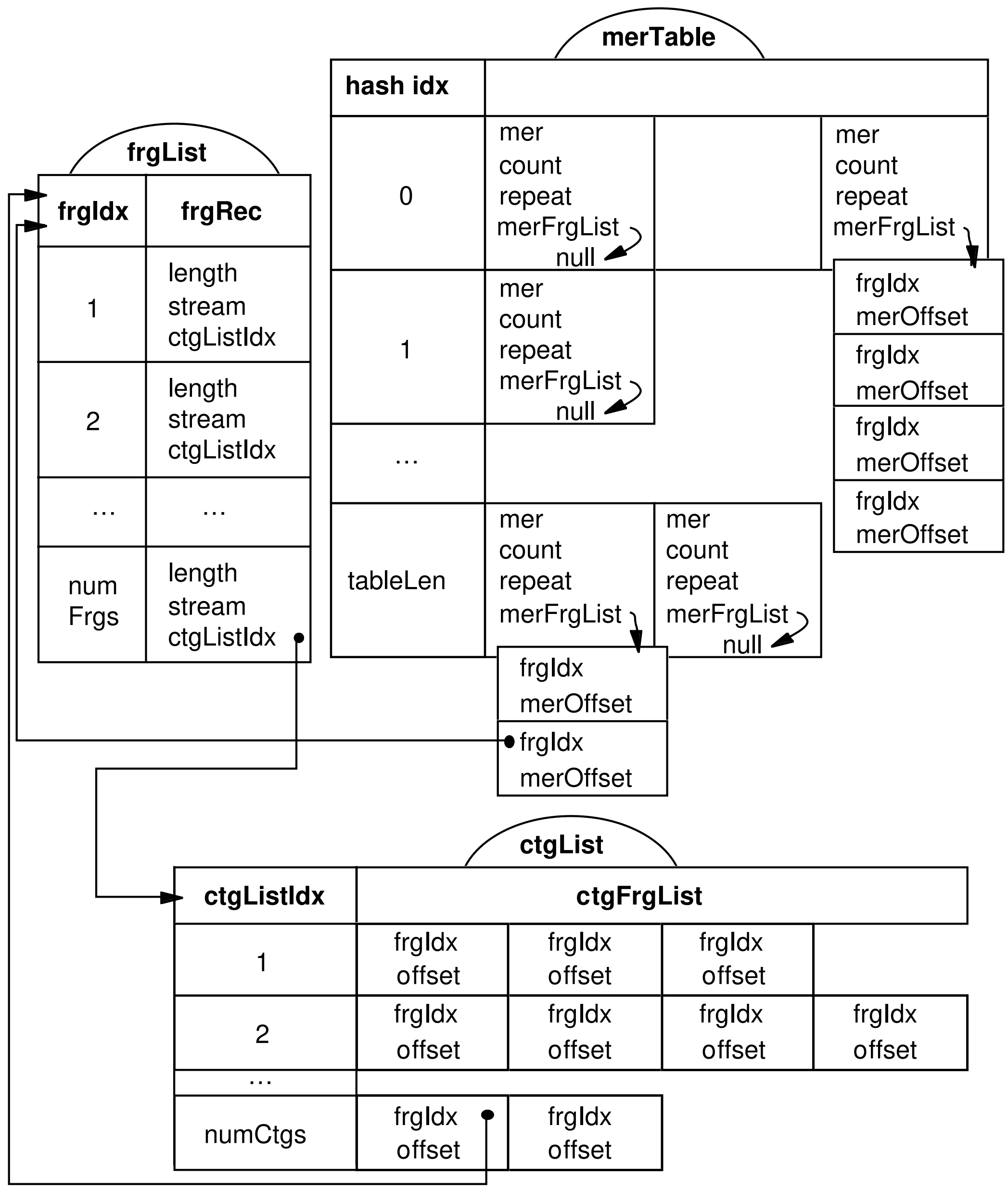


Figure D-1. *SLIC* Data Structures. The `frgList` contains a `FrgRec` for each fragment read in the dataset. In `FrgRec`, `ctgListIdx` is an index into the `ctgList` (the list of all contigs). The `merTable` is a bucket-and-chain hash table. The first x bases of a mer are used to index into the table and the last y bases are in `MerRec` records in the `MerRecList`. Each `MerRec` may have a `merFrgList` associated with it. A `merFrgList` is a list of `MerFrgRec`. In a `MerFrgRec`, the `frgIdx` indexes a fragment read record in the `frgList`. The `merOffset` is the offset of the mer in the fragment read. The `ctgList` is a list of all contigs for the dataset. Each contig is represented by a list of `CtgFrgRec`. The `frgIdx` in a `CtgFrgRec` indexes a `frgRec` in the `frgList`. The `offset` specifies the position of the fragment read in the contig.

Pseudocode

1. Initialize all variables and structures

/* Get fragment reads */

2. While `frgRec = getNextFrgRead()`
   2.1 `frgList = addToList(frgList, frgRec)`
   2.2 `numFrgs = numFrgs + 1`

/* Count occurrences of mers */

3. For `i = 1` to `numFrgs`
   3.1 While `mer = getNextMer(frgList[i], 0)`
      3.1.1 `merRec = getMerRec(merTable, mer)`
      3.1.2 If `merRec.count < maxCount`
         3.1.2.1 `merRec.count = merRec.count + 1`
      3.1.3 Else `merRec.repeat = true`

/* Choose mer tags in each fragment read */

4. For `i = 1` to `numFrgs`
   4.1 `merOffset = 0`
   4.2 For `j = 1` to `numMers`
      4.3.1 `found = false`
      4.3.2 While `mer = getNextMer(frgList[i], j)`
         4.3.2.1 `scoreRec = assignScore(mer, merOffset++)`
         4.3.2.2 If `betterScore(scoreRec, tagScoreRec, j)`
            4.3.2.2.1 `tagScoreRec = scoreRec`
            4.3.2.2.2 `found = true`
      4.3.3 If `found`
         4.3.3.1 `newMerFrg = newMerFrgRec(i, tagScoreRec.merOffset)`
         4.3.3.2 `merRec = getMerRec(merTable, tagScoreRec.mer)`

4.3.3.3 `merRec.merFrgList = addToList(merRec.merFrgList, newMerFrg)`

/* If not in the list, add fragment reads that contain a previously chosen mer tag to the mer's `merFrgList`*/

5. For `i = 1` to `numFrgs`
   - 5.1 `merOffset = 0`
   - 5.2 While `mer = getNextMer(frgList[i], 0)`
     - 5.2.1 `merRec = getMerRec(merTable, mer)`
     - 5.2.2 If `merRec.merFrgList` and `not inMerFrgList(merRec.merFrgList, i)`
       - 5.2.2.1 `newMerFrg = newMerFrgRec(i, merOffset++)`
       - 5.2.2.2 `merRec.merFrgList = addToList(merRec.merFrgList, newMerFrg)`

/* Make, add to, and merge contigs */

6. For `i = 1` to `listLen(merTable)`
   - 6.1 For `j = 0` to `listLen(merTable[i])`
     - 6.1.1 `merRec = merTable[i][j]`
     - 6.1.2 If `merRec.merFrgList` and `not oneCtg(merRec.merFrgList)`
       - 6.1.2.1 `makeMatchFrgLists(merRec.merFrgList, matchFrgList)`
       - 6.1.2.2 For `j = 0` to `listLen(matchingMerFrgLists)`
         - 6.1.2.2.1 `ctgListIdx = getCtgListIdx(matchFrgList[k])`
         - 6.1.2.2.2 If `ctgListIdx = 0`
           - 6.1.2.2.2.1 `ctgListIdx = newCtg(ctgList, matchFrgList[k][1])`
         - 6.1.2.2.3 For `m = 1` to `listLen(matchFrgList[k])`
           - 6.1.2.2.3.1 `merFrgRec = matchFrgList[k][m]`
           - 6.1.2.2.3.2 If `frgList[merFrgRec.frgIdx].ctgListIdx = 0`
             - 6.1.2.2.3.2.1 `newCtgFrg = newCtgFrgRec(ctgList[ctgListIdx], merFrgRec.frgIdx)`
             - 6.1.2.2.3.2.2 `ctgList[ctgListIdx] = addToList(ctgList[ctgListIdx], newCtgFrg)`
             - 6.1.2.2.3.2.3 `frgList[merFrgRec.frgIdx].ctgListIdx = ctgListIdx`

6.1.2.2.3.3 Else If `frgList[merFrgRec.frgIdx].ctgListIdx <> ctgListIdx`

6.1.2.2.3.3.1 `ctgList = mergeCtgs(ctgList, frgList[merFrgRec.frgIdx].ctgListIdx, ctgListIdx)`

Detailed Description

1. Initialize all variables and structures.

2. While `frgRec = getNextFrgRead()`

Sequences of base calls for fragment reads are read and information is stored in a record of type `FrgRec`. Information includes:

1) `length`, the total number of base calls;
2) `stream`, the sequence of base calls;
3) `ctgListIdx`, an index into `ctgList` that specifies which contig includes the fragment (initialized to 0).

The base calls are encoded in `stream` such that two bits are used to represent each base call; *A*, *G*, *T*, and *C* are represented by 00, 01, 10, and 11, respectively.

2.1 `frgList = addToList(frgList, frgRec)`

At this point, reads that are too short to be useful can be excluded from the dataset. Otherwise, I add the new `FrgRec` record to `frgList` (the list of all fragment reads in the dataset).

2.2 `numFrgs = numFrgs + 1`

3. For `i = 1` to `numFrgs`

Iterate through each of the reads in `frgList`, counting the total occurrences of individual mers.

3.1 While `mer = getNextMer(frgList[i], 0)`

Get the next mer for the current fragment read. (The '0' in the function call indicates that I am scanning the entire read, 5' to 3'.) Only the mer or its reverse complement needs to be processed; I arbitrarily choose the one with the smaller integer value. For example, with 2-bit encoding, the bit stream for the sequence GATT is 01001010, yielding an integer value of 74. The reverse complement and its corresponding bit stream are AATC and 00001011, yielding an integer value of 11. I return the reverse complement of the mer since its value (11) is less than that of the mer (74). Null is returned after the last mer in the sequence stream has been returned.

3.1.1 `merRec = getMerRec(merTable, mer)`

Get the `merRec` corresponding the the current mer. First, the integer value of the 2-bit encoding of the first x bases of the mer is used as the index into `merTable`. Abstractly, the `merTable` is a bucket-and-chain hash table where the first x bases specify the buckets. Since two bits encode each base in the x-mer, the length of the table is 4^x. Figure D-2 gives an example of `merTable` indexing and length.

If no `merRec` exists for the mer when `getMerRec()` is called, a new `merRec` is created and added to the `merRecList`. Since the records are dynamically allocated, only mers that occur in the dataset have an associated record. Note that all `merFrgList` fields remain `null` until step 4.

3.1.2 If `merRec.count < maxCount`

Increment the count for the current `merRec` if it is less than the `maxCount` threshold. A threshold is set that specifies the maximum number of identical mers that occur in the dataset. (The default setting is 150% of expected redundancy.) When the number of occurrences reaches the threshold, the mer is marked as a repeat. As an example, consider a dataset with an expected redundancy of six. If a mer occurs more than nine times, I presume that it is repeated in the original DNA fragment. In that case, the `repeat` field of the `MerRec` is set to `true`.

3.1.2.1 `merRec.count = merRec.count + 1`

3.1.3 Else `merRec.repeat = true`

4. For `i = 1` to `numFrgs`

Iterate through the fragment reads, choosing mer tags.

4.1 `merOffset = 0`

Initialize `merOffset` to 0. The offset of the current mer in a `FrgRec.stream` is needed for the `merOffset` field in a `MerFrgRec`.

4.2 For `j = 1` to `numMers`

Choose a mer tag for each partition of the fragment read. The user specifies how many mers to choose per fragment read. The read is divided into that many partitions and a mer is chosen for each partition. Multiple mers are usually needed to help ensure that each overlap has a chosen mer tag. It is especially important to have mers near either end of the fragment reads.

4.3.1 `found = false`

`found` remains `false` if no mer tag is found the the partition.

4.3.2 While `mer = getNextMer(frgList[i], j)`

Get the next mer for the current partition and fragment read. I am only interested in finding a mer in the current partition of the fragment read, so I only return mers in the that partition. For example, consider the case where the number of partitions is three and I have a fragment read of 440 bases. The three partitions will include bases 1 to 147, bases 148 to 294, and bases 295 to 440. If possible, a mer tag is chosen in each of these partitions. Null is returned after I have returned the last mer in the partition.

4.3.2.1 `scoreRec = assignScore(mer, merOffset++)`

Assign a score for the current mer by preferring mers with the fewest number of occurrences (greater than one). The score for a mer is `maxCount - merRec.count`. The `merOffset` is incremented at the completion of the function call. If the mer is marked as a repeat, a score of 0 is assigned.

a)

Index	Mer (first *x* bases)	2-Bit Encoding
0	**AAAAAAAA**	0000000000000000
1	**AAAAAAAG**	0000000000000001
2	**AAAAAAAT**	0000000000000010
…	…	…
65,533	**CCCCCCCG**	1111111111111101
65,534	**CCCCCCCT**	1111111111111110
65,535	**CCCCCCCC**	1111111111111111

b)

hash index (first *x* bases)	**merList** (last *y* bases)		
	merRec 1	**merRec 2**	**merRec 3**
31,352 (**GCTTGCTA**)	**mer** **GTCA**	**mer** **TTAC**	**mer** **GAAT**
	count 1	**count** 6	**count** 3
	repeat false	**repeat** false	**repeat** false
	merFrgList null	**merFrgList** null	**merFrgList** null

Figure D-2. Mer Table and Lists. a) This is an example `merTable`. In this example, x = 8 and the length of the table is 65,536. The integer values of the 2-bit encoding of the 8-mers indexes the table. I use the last y bases in the current mer to find the corresponding `merRec`. Each entry in the `merTable` is a `merRecList`; the lists form the chains for the hash table. In the list, there is one `merRec` for each mer that occurs in the dataset of fragment reads. b) This is an example of a `merRecList`. In this example, consider the following three mers where x = 8 and y = 4: `GCTTGCTAGTCA`, `GCTTGCTATTAC`, and `GCTTGCTAGAAT`. The first x bases are identical, so all hash into `merTable` at index 31,352. The last y bases are different, so each has its own record in the `merRecList`. In the dataset of fragment reads for this example, I have so far encountered one mer of `GCTTGCTAGTCA`, six of `GCTTGCTATTAC`, and three of `GCTTGCTAGAAT`.

4.3.2.2 If `betterScore(scoreRec, tagScoreRec, j)`

Check if the current score is better than the best score so far. There are often ties in scores and the identification of the better score is dependent upon which partition I am processing. I want to choose mer tags that are near either end of the fragment read and spaced as evenly as possible throughout the rest of the read. If I am scoring the first partition, I choose the first mer with the highest score. Conversely, if I am scoring the last partition, I choose the mer with the last occurrence of the highest score. For middle partitions, I choose the mer with the highest score that is nearest the center of the partition. If the score in `scoreRec` is 0, `false` is returned.

4.3.2.2.1 `tagScoreRec = scoreRec`

Update `tagScoreRec` if the current score in `scoreRec` is better than the score in `tagScoreRec`.

4.3.2.2.1 `found = true`

Indicate that at least one possible mer tag has been found.

4.3.3 `If found`

4.3.3.1 `newMerFrg = newMerFrgRec(i, tagScoreRec.merOffset)`

Make a new `MerFrgRec` using the fragment index and the direction and offset of the mer tag.

4.3.3.2 `merRec = getMerRec(merTable, tagScoreRec.mer)`

Find the `merRec` associated with the mer tag (as in step 3.1.1).

4.3.3.3 `merRec.merFrgList = addToList(merRec.merFrgList, newMerFrg)`

Add the new `MerFrgRec` to the mer tag's `merFrgList`. The `merFrgList` is the list of all fragment reads for which the mer has been chosen as a mer tag.

5. For `i = 1` to `numFrgs`

This iteration through the data ensures that if a mer was chosen for one read, all other reads with that mer are in the mer's associated `merFrgList`. For each mer, the existence of a `merFrgList` indicates that the mer has been chosen as a tag for at least one fragment. (The list remains `null` until the mer has been chosen by a fragment.)

The amount of storage and processing time can be reduced by incorporating a check for existing `merFrgList's` in step 4. In that step, if a `merFrgList` exists for a mer in the current partition, the mer can be immediately chosen as the mer tag for the current fragment. Checking for previously chosen mers in step 4 does not eliminate the need for step 5.

5.1 `merOffset` = 0

Set `merOffset` to 0. As in step 4.1, the offset of the current mer in a `FrgRec.stream` is needed for the `merOffset` field in a `MerFrgRec`.

5.2 While `mer` = `getNextMer(frgList[i], 0)`

Get the next mer for the current fragment. (The '0' in the function call indicates that I am scanning the entire read, 5' to 3'.)

5.2.1 `merRec` = `getMerRec(merTable, mer)`

Find the `merRec` associated with the mer (as in step 3.1.1).

5.2.2 If `merRec.merFrgList` and `not inMerFrgList(merRec.merFrgList, i)`

Check if a `merFrgList` exists and if the fragment is already in the list. If there is no `merFrgList` for the current mer, then the mer has never been chosen as a tag and I ignore it. Otherwise, I check if the current fragment read already has an entry in the `MerfrgList`. If not, I add a new `MerFrgRec` containing the fragment information to the list.

5.2.2.1 `newMerFrg` = `newMerFrgRec(i, merOffset++)`

Make a new `MerFrgRec` using the fragment index and position of the mer. The `merOffset` is incremented at the completion of the function call.

5.2.2.2 `merRec.merFrgList` = `addToList(merRec.merFrgList, newMerFrg)`

Add the new `MerFrgRec` to the `merRec.merFrgList`.

6. For `i` = 1 to `listLen(merTable)`

Iterate through the `merTable`, making, adding to, and merging contigs. At the completion of this step I have a list of contigs in `ctgList`. Each contig is represented as a list of `ctgFrgRec` records that each contain an index into the `frgList` and the offset of the fragment read in the contig. Each contig list of `ctgFrgRec` are ordered by offset to simplify further processing of the contig layouts. Figure D-3 contains an example of a completed `ctgList` and its associated layout.

6.1 For `j = 0` to `listLen(merTable[i])`

Each entry in the `merTable` is a list of `merRec's`. I check each entry in the list to see if the fragment reads in the `merFrgList` should be overlapped in a contig.

6.1.1 `merRec = merTable[i][j]`

Get the next `merRec` in the `merRecList`.

6.1.2 If `merRec.merFrgList` and `not oneCtg(merRec.merFrgList)`

First I check if the `merRec` has a `merFrgList`, indicating that the mer has been chosen as a tag. Then I check if all the fragments in the list are already in the same contig

6.1.2.1 `makeMatchFrgLists(merRec.merFrgList, matchFrgList)`

I check for pairwise overlap similarity between all fragment reads in `merRec.merFrgList`. In addition, if any of the fragments are already in a contig, I check the pairwise similarity of all fragments in the contig with any overlapping fragments in the `merRec.merFrgList`. (The fragments in a contig might not contain the current mer, but might still overlap some of the fragment reads in the current `merRec.merFrgList`.) A threshold is set that specifies the required amount of match similarity. If not all the fragments and contigs in a list have above-threshold pairwise similarity, then the list is divided by `makeMatchFrgLists` into multiple lists such that the similarity within each list is above threshold.

6.1.2.2 For `j = 0` to `listLen(matchingMerFrgLists)`

Iterate through the fragment read lists in the `matchingMerFrgLists`.

6.1.2.2.1 `ctgListIdx = getCtgListIdx(matchFrgList[k])`

Check if any of the fragments in the `matchFrgList[k]` is in a contig. If a `merRec.ctgListIdx` is 0, the fragment is not yet in a contig; otherwise, it identifies an index into the `ctgList`. In this step, if any of the fragments in `merRec.merFrgList` is in a contig, I return its `ctgListIdx`; otherwise I return 0. If none of the fragments is in a contig, I make a new one and add all other fragments to the same contig. If any fragments are in other contigs, I add them to the same contig also.

6.1.2.2.2 If `ctgListIdx = 0`

If the ctgListIdx is 0, none of the fragments is in a contig. If the ctgListIdx is greater than 0, I enter all other fragment reads in the `merRec.merFrgList` into the contig indexed by ctgListIdx.

6.1.2.2.2.1 `ctgListIdx = newCtg(ctgList, matchFrgList[k][1])`

I arbitrarily make a new contig with the `CtgFrgRec` of the first fragment read in the `matchFrgList[k]`. The offset of this fragment read in the `CtgFrgRec` is 0. The `ctgListIdx` is recorded as the `ctgListIdx` field in the fragment's `frgRec`. Subsequently, all other fragments in the list will be added to the new contig.

6.1.2.2.3 For `m = 1` to `listLen(matchFrgList[k])`

Iterate through the fragment reads in the `matchFrgList[k]`.

6.1.2.2.3.1 `merFrgRec = matchFrgList[k][m]`

Get the next `merFrgRec` in the list.

6.1.2.2.3.2 If `frgList[merFrgRec.frgIdx].ctgListIdx = 0`

If the `ctgListIdx` for the fragment is 0, the fragment is not in a contig.

a) **ctgList**

ctgListIdx	ctgFrgList				
1	**frgIdx** 16 **offset** 0	**frgIdx** 6 **offset** 8	**frgIdx** 2 **offset** 14	**frgIdx** 14 **offset** 25	**frgIdx** 10 **offset** 34
2	**frgIdx** 1 **offset** 0	**frgIdx** 7 **offset** 9	**frgIdx** 15 **offset** 19		
3	**frgIdx** 12 **offset** 0	**frgIdx** 3 **offset** 6	**frgIdx** 13 **offset** 17	**frgIdx** 5 **offset** 33	

b) **Layout**

Contig 1

```
frgIdx 16: TAGGCTAGGCCCCATATGC
frgIdx  6:         GCCCCATATGCTGACGGCGCA
frgIdx  2:               TATGCTGACGGCGCATTTGAC
frgIdx 14:                          CGCATTTGACCCCAAAGTC
frgIdx 10:                                   CCCCAAAGTCCCCG
```

Contig 2

```
frgIdx  1: GATTGGGGACCAGCACCACCTTAGC
frgIdx  7:          CCAGCACCACCTTAGCAGGA
frgIdx 15:                    CTTAGCAGGATTGACACGGGTA
```

Contig 3

```
frgIdx 12: TTAGGATCGCGAGCTTA
frgIdx  3:       TCGCGAGCTTATCCAGAGTCGACCGG
frgIdx 13:                  TCCAGAGTCGACCGGTAGGGCTACACAAG
frgIdx  5:                                  AGGGCTACACAAGCCT
```

Figure D-3. A Contig List and Layout. In this example, 16 fragment reads are in the dataset. a) Twelve reads are aligned into three contigs. Each `ctgFrgList` contains `ctgFrgRec` that include the `frgIdx` (an index into the `frgList`) for each read in the contig. The records also include the `offset` of the read in the contig. Four reads (`frgIdx` = 4, 8, 9, and 11) did not overlap any other read and are not in any contig. b) The fragment reads for each contig are aligned according to the offsets listed in the `ctgFrgRec`.

6.1.2.2.3.2.1 `newCtgFrg = newCtgFrgRec(ctgList[ctgListIdx], merFrgRec.frgIdx)`

Make a new `CtgFrgRec` using the index of the fragment read. The offset of the read in the contig is calculated in the function. If the mer for the current fragment read is not in the same direction as in the contig, then I must set the offset to reflect the reverse complement of the read. An example is given in Figure D-4.

6.1.2.2.3.2.2 `ctgList[ctgListIdx] = addToList(ctgList[ctgListIdx], newCtgFrg)`

Add the new `CtgFrgRec` to the contig. The list of `CtgFrgRec` is ordered by offset.

6.1.2.2.3.2.3 `frgList[merFrgRec.frgIdx].ctgListIdx = ctgListIdx`

Record the `ctgListIdx` in the `frgRec` for the current fragment read.

6.1.2.2.3.3 Else If `frgList[merFrgRec.frgIdx].ctgListIdx <> ctgListIdx`

If the `ctgListIdx` for the current fragment does not match the `ctgListIdx` for the contig I am building, then I have two separate contigs that I merge.

Fragment reads in `MerfrgList` for mer ACCACACC:

```
1.  GACCACACCGTAGTG
2.  AGGATAGACCACACCGTAG
3f. GGGGTGGGTCACTACGGTGTGGTCT   (Forward)
3r. AGACCACACCGTAGTGACCCACCCC   (Reverse-complemented)
```

Make a new contig with fragment read 1:

```
1.  GACCACACCGTAGTG    offset = 0
```

Add fragment read 2:

```
1.        GACCACACCGTAGTG    offset = 6
2.  AGGATAGACCACACCGTAG      offset = 0
```

Add fragment read 3:

```
1.        GACCACACCGTAGTG              offset = 6
2.  AGGATAGACCACACCGTAG                offset = 0
3r.      AGACCACACCGTAGTGACCCACCCC     offset = 5
```

Figure D-4. Reversing Reads when Forming Contigs. The example in shows three fragment reads added one at a time to a growing contig. The mer occurs in the reverse direction in the third fragment, so the offset is from the original 3' end of the read.

6.1.2.2.3.3.1 `ctgList = mergeCtgs(ctgList, frgList[merFrgRec.frgIdx].ctgListIdx, ctgListIdx)`

Merging contigs requires the adjustment of all the offsets of the fragment reads in one of the contigs to reflect their new positions relative to fragment reads in the other contig. In the case that the mer is in the forward direction in one contig and reversed in the other, the offset adjustment must also reflect the reverse complementation of one contig. During merging, the `ctgListIdx` is updated for all the affected fragments' `frgRec`. After moving all fragment reads from one `ctgFrgList` to another, the empty list is deleted.

References

Allex, C.F., Baldwin, S.F., Shavlik, J.W., and Blattner, F.R. (1996). Improving the quality of automatic DNA sequence assembly using fluorescent trace-data classifications. *Proceedings, Fourth International Conference on Intelligent Systems for Molecular Biology*, 3-14. St. Louis, MO. Menlo Park, CA: AAAI Press.

Allex, C.F., Baldwin, S.F., Shavlik, J.W., and Blattner, F.R. (1997). Increasing consens accuracy in DNA fragment assemblies by incorporating fluorescent trace representatio *Proceedings, Fifth International Conference on Intelligent Systems for Molecular Biology*, 3-14. Halkidiki, Greece. Menlo Park, CA: AAAI Press.

Allex, C.F., Shavlik, J.W., and Blattner, F.R. (1999). Neural network input representations that produce accurate consensus sequences from DNA fragment assemblies. *BioInformatics*, to appear.

Anson, E. and Myers, E. (1997). ReAligner: A program for refining DNA sequence multi-alignments, *Journal of Computational Biology*, 4(3):369-383.

Ansorge, W., Sproat, B.S., Stegemann, J. and Schwager, C. (1986). A non-radioactive automated method for DNA sequence determination. *Journal of Biochemical and Biophysical Methods*, 13:315-323.

Baldi, P., and Brunak, S. (1998). *Bioinformatics: The machine learning approach.* Cambridge, MA: MIT Press.

Berger, M.P. and Munson, P.J. (1991). A novel randomized iterative strategy for aligning multiple protein sequences. *Computer Applications in the Biosciences*, 7(4):479-484.

Benson, D.A., Boguski, M.S., Lipman, D.J., Ostell, J., and Ouellette, B.F. (1998). GenBank. *Nucleic Acids Research*, 26(1):1-7.

Blattner, F.R., Plunkett, G. 3rd, Bloch, C.A., Perna, N.T., Burland, V., Riley, M., Collado-Vides, J., Glasner, J.D., Rode, C.K., Mayhew, G.F., Gregor, J., Davis, N.W., Kirkpatrick, H.A., Goeden, M.A., Rose, D.J., Mau, B., Shao, Y. (1997). The complete genome sequence of *Escherichia coli* K-12. *Science*, 277(5331):1453-74.

Bonfield, J.K., Smith, K.F. and Staden, R. (1995). A new DNA sequence assembly program. *Nucleic Acids Research*, 24:4992-4999.

Boswell, D.R. and McLachlan, A.D. (1984). Sequence comparison by exponentially-damped alignment. *Nucleic Acids Research*, 12:457-464.

Bouriakov, G.V. and Mayhew, G.F. (1995). *E. coli Genome Project*, University of Wisconsin – Madison. Unpublished.

Boysen, C., Simon, M.I., and Hood, L. (1997). Analysis of the 1.1-Mb Human alpha/delta T-cell receptor locus with bacterial artificial chromosome clones. *Genome Research.* 7:330-338.

Brutlag, D.L., Dautricourt, J.P., Diaz, R., Fier, J., Moxon, B., and Stamm, R. (1993). BLAZE: An implementation of the Smith-Waterman sequence comparison algorithm on a massively parallel computer. *Computers* and *Chemistry*, 17(2):203-207.

Chen, E.Y. (1994). The efficiency of automated DNA sequencing. Adams, M.D., Fields, C. and Venter, J.C., eds., *Automated DNA Sequencing and Analysis*, 3-10. San Diego, CA: Academic Press.

Chen, T. and Zhang, M.Q. (1998). Pombe: A gene-finding and exon-intron structure prediction system for fission yeast. *Yeast*; 14(8):701-710.

Chen, W. Q. and Hunkapiller, T. (1992). Sequence accuracy of large DNA sequencing projects. *DNA Sequence – Journal of DNA Sequencing and Mapping*, 2:335-342.

Craven, M.W. and Shavlik, J. W. (1993). Learning to predict reading frames in *E. coli* DNA sequences. *Proceedings of the 26th Hawaii International Conference on System Sciences*, 773-782. Wailea, HI.

Ewing, B. and Green, P. (1998). Base-calling of automated sequencer traces using Phred. II. Error probabilities. *Genome Research*, 8:186-194.

Ewing, B., Hillier, L., Wendl, M.C., and Green, P. (1998). Base-calling of automated sequencer traces using Phred. I. Accuracy assessment. *Genome Research*, 8:175-185.

Fickett, J.W. (1984). Fast optimal alignment. *Nucleic Acids Research*, 12(1):175-179.

Frenkel, K.A. (1991). The Human Genome Project and informatics. *Communications of the ACM*, 34(11):41-51.

Giddings, M.C., Brumley, R.L. Jr., Haker, M. and Smith, L.M. (1993). An adaptive, object oriented strategy for base calling in DNA sequence analysis. *Nucleic Acids Research*, 21:4530-4540.

Golden, J.B. III, Torgersen, D. and Tibbetts, C. (1993). Pattern recognition for automated DNA sequencing: I. On-line signal conditioning and feature extraction for basecalling. *Proceedings, First International Conference on Intelligent Systems for Molecular Biology*, 136-134. Bethesda, MD. Menlo Park, CA: AAAI Press.

Goodman, B. (1990). The genetic anatomy of us (and a few of our friends). *BioS*
40(7):484-489.

Green, E.D. and Waterston, R.H. (1991). The Human Genome Project: Prospects and implications for clinical medicine. *Journal of the American Medical Association*, 266(14):1966-1975.

Green, P. (1997a). Against a whole genome shotgun. *Genome Research*, 7:410-417.

Green, P. (1997b). Genome sequence assembly. *Program and Abstract Book, First Annual Conference on Computational Genomics*, 15. Herndon, VA: TIGR Science Education Foundation, Inc., Genomic Science Series.

Griffiths, J.F., Gelbart, W.M., Miller, J.H., and Lewontin, R.C. (1998). *Modern Genetic Analysis: First Edition*. New York, NY: W.H. Freeman and Company.

Heumann, J.M., Lapedes, A.S. and Stormo, G.D. (1994). Neural networks for determining protein specificity and multiple alignment of binding sites. *Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology*, 188-194. Bethesda, MD. Menlo Park, CA: AAAI Press.

Huang, X. (1994). On global sequence alignment. *Computer Applications in the Biosciences*, 10(3):227-235.

Huang, X. (1996). An improved sequence assembly program. *Genomics*, 33:21-31.

Jain, M. and Myers, E.W. (1997). Algorithms for computing and integrating physical maps using unique probes. *Journal of Computational Biology*, 4(4):449-466.

Johnson, M.S. and Doolittle, R.F. (1986). A method for the simultaneous alignment of three or more amino acid sequences. *Journal of Molecular Evolution*, 23(3):267-278.

Karplus, K., Barrett, C., and Hughey, R. (1998). Hidden Markov models for detecting remote protein homologies. *Bioinformatics*; 14(10):846-856.

Kelley, J.M. (1994). Automated dye-terminator DNA sequencing. Adams, M.D., Fields, C. and Venter, J.C., eds., *Automated DNA Sequencing and Analysis*, 175-181. San Diego, CA: Academic Press.

Khurshid, F. and Beck, S. (1993). Error analysis in manual and automated DNA sequencing. *Analytical Biochemistry*, 208: 138-143.

Lavorgna, G., Guffanti, A., Borsani, G., Ballabio, A., and Boncinelli, E. (1999). TargetFinder: Searching annotated sequence databases for target genes of transcription factors. *Bioinformatics*; 15(2):172-173.

Lawrence, C.B. and Solovyev, V.V. (1994). Assignment of position-specific error probability to primary DNA sequence data. *Nucleic Acids Research*, 22:7.

Li, P., Kupfer, K.C., Davies, C.J., Burbee, D., Evans, G.A., Garner, H.R. (1997). PRIMO: A primer design program that applies base quality statistics for automated large-scale DNA sequencing. *Genomics*, 40:476-485.

Lipshutz, R. J., Taverner, F., Hennessy, K., Gartzell, G., and Davis, R. (1994). DNA sequence confidence estimation. *Genomics*, 19:417-424.

Marshall, E. (1999). A high stakes gamble on genome sequencing. *Science*, 284(5422):1906-1909.

Martinez, H.M. (1983). An efficient method for finding repeats in molecular sequences. *Nucleic Acids Research*, 11:4629-4634.

Maxam, A.M. and Gilbert, W. (1977). A new method for sequencing DNA. *Proceedings of the National Academy of Science, USA* 74:560-564.

McClelland, J.L. and Rumelhart, D.E. (1986). *Parallel Distributed Processing, Vol I.* Cambridge, MA: MIT Press.

McCombie, W.R. and Martin-Gallardo, A. (1994). Large-scale, automated sequencing of human chromosomal regions. Adams, M.D., Fields, C. and Venter, J.C., eds., *Automated DNA Sequencing and Analysis*, 159-166. San Diego, CA: Academic Press.

McCulloch, W.S. and Pitts, W. (1943). A logical calculus of the ideas immanent in neural nets. *Bulletin of Mathematical Biophysics*; 5:115-137.

MIT. (1998). http://www-seq.wi.mit.edu/informatics/alewife.html.

Myers, E.W. (1994). Advances in sequence assembly. Adams, M.D., Fields, C. and Venter, J.C., eds., *Automated DNA Sequencing and Analysis*, 231-238. San Diego, CA: Academic Press.

Naeve, C.W., Buck, G.A., Niece, R.L., Pon, R.T., Roberson, M., and Smith, A.J. (1995). Accuracy of automated DNA sequencing: A multi-laboratory comparison of sequencing results. *BioTechniques*, 19(3):448-453.

Needleman, S.B. and Wunsch, C.D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443-453.

NHGRI. (1998).
http://www.nhgri.nih.gov/Grant_info/Funding/Statements/RFA/quality_standard.html

NCBI. (1999). http://www.ncbi.nlm.nih.gov/Web/Genbank/.

Noordewier, M.O., Towell, G. and Shavlik, J.W. (1991). Training knowledge-base neural networks to recognize genes in DNA sequences. Lippmann, R.P., Moody, J.E. and Touretzky, D.S., eds., *Advances in Neural Information Processing Systems*, 3:530-536. Denver, CO: Morgan Kaufmann.

Parker, L.T., Deng, Q., Zakeri, H., Carlson, C., Nickerson, D.A. and Kwok, P.Y. (1995). Peak height variations in automated sequencing of PCR products using taq dye-terminator chemistry. *BioTechniques*, 19:116-121.

Parsons, R., and Johnson, M.E. (1995). DNA sequence assembly and genetic algorithms: new results and puzzling insights. *Proceedings, Third International Conference on Intelligent Systems for Molecular Biology*, Cambridge, England. Menlo Park, CA: AAAI Press.

Parsons, R. (1993). Genetic algorithms for DNA sequence assembly. *Proceedings, First International Conference on Intelligent Systems for Molecular Biology*, 310-318. Bethesda, MD. Menlo Park, CA: AAAI Press.

Pedersen, A.G. and Engelbrecht, J. (1995). Investigations of *Escherichia coli* promoter sequences with artificial neural networks: New signals discovered upstream of the transcriptional start point. *Proceedings of the Third International Conference on Intelligent Systems for Molecular Biology*, 292-299. Cambridge, England. Menlo Park, CA: AAAI Press.

Perkin-Elmer. (1995). *DNA Sequencing: Chemistry Guide*. Foster City, CA.

Prober, J.M., Trainor, G.L., Dam, R.J., Hobbs, F.W., Robertson, C.W., Zagursky, R.J., Cocuzza, A.J., Jensen, M.A. and Baumeister, K. (1987). A system for rapid DNA sequencing with fluorescent chain-terminating dideoxynucleotides. *Science*, 238:336-341.

Qian, N. and Sejnowski, T. (1988). Predicting the secondary structure of globular proteins using neural network models. *Journal of Molecular Biology*, 202:865-884.

Richterich, P. (1998). Estimation of errors in "raw" DNA sequences: A validation study. *Genome Research*, 8:251-259.

Rost, B. and Sander, C. (1993). Prediction of protein secondary structure at better than 70% accuracy. *Journal of Molecular Biology*, 232:584-599.

Rowen, L. and Koop, B.F. (1994). Zen and the art of large-scale genomic sequencing. Adams, M.D., Fields, C. and Venter, J.C., eds., *Automated DNA Sequencing and Analysis*, 167-174. San Diego, CA: Academic Press.

Sanger, F., Nicklen, S. and Coulson, A.R. (1977). DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Science USA,* 74:5463-5467.

Seto, D., Koop, B.F. and Hood, L. (1993). An experimentally derived data set constructed for testing large-scale DNA sequence assembly algorithms. *Genomics*, 15:673-676.

Smith, L.M., Sanders, J.Z., Kaiser, R.J., Hughes, P., Dodd, C., Connell, C.R., Heiner, C., Kent, S.B.H. and Hood, L.E. (1986). Fluorescence detection in automated DNA sequence analysis. *Nature*, 321:674-679.

Smith, T.F. and Waterman, M.S. (1980). Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195-197.

Snyder E.E., and Stormo, G.D. (1993). Identification of coding regions in genomic DNA sequences: An application of dynamic programming and neural networks. *Nucleic Acids Research*, 21:607-613.

Staden, R. (1986). The current status and portability of our sequence handling software. *Nucleic Acids Research*, 14(1):215-231.

Staden, R. (1984a). Computer methods to aid the determination and analysis of DNA sequences. *Biochemical Society Transactions*, 12:1005-1008.

Staden, R. (1984b). A computer program to enter DNA gel reading data into a computer. *Nucleic Acids Research*, 12(1):499-503.

Staden, R. (1982a). Automation for the computer handling of gel reading data produced by the shotgun method of DNA sequencing. *Nucleic Acids Research*, 10(15):4731-4751.

Staden, R. (1982b). An interactive graphics program for comparing and aligning nucleic acid and amino acid sequences. *Nucleic Acids Research*, 10(9):2951-2961.

Staden, R. (1980). A new computer method for the storage and manipulation of DNA gel reading data. *Nucleic Acids Research*, 8(16):3673-3694.

Staden, R. (1979). A strategy of DNA sequencing employing computer programs. *Nucleic Acids Research*, 6(7):2601-2610.

Staden, R. (1978). Further procedures for sequence analysis by computer. *Nucleic Acids Research*, 5(3):1013-1015.

Staden, R. (1977). Sequence data handling by computer. *Nucleic Acids Research*, 4(11):4037-4051.

Streletc, V.B., Shindyalov, I.N., Kolchanov, N.A., and Milanesi, L. (1992). Fast, statistically based alignment of amino acid sequences on the base of diagonal fragments of DOT-matrices. *Computer Applications in the Biosciences*, 8(6):529-534.

Stolorz, P., Lapedes, A. and Xia, Y. (1992). Predicting protein secondary structure using neural net and statistical methods. *Journal of Molecular Biology*, 225:363-377.

Studier, F.W. (1989). A strategy for high-volume sequencing of cosmid DNAs: Random and directed priming with a library of oligonucleotides. *Proceedings of the National Academy of Science* USA: 86:6917-6921.

Subbiah, S. and Harrison, S.C. (1989). A method for multiple sequence alignment with gaps. *Journal of Molecular Biology*, 209(4):539-548.

Sun, Z.R., Zhang, C.T., Wu, F.H., and Peng, L.W. (1996). A vector projection method for predicting supersecondary motifs. *Journal of Protein Chemistry*; 15(8):721-729.

Sutton, G., White, O., Adams, M. and Kerlavage, A. (1995). TIGR Assembler: A new tool for assembling large shotgun sequencing projects. *Genome Science & Technology*, 1: 9-19.

Tibbetts, C., Bowling, J.M. and Golden, J.B. III. (1994). Neural networks for automated basecalling of gel-based DNA sequencing ladders. Adams, M.D., Fields, C. and Venter, J.C., eds., *Automated DNA Sequencing and Analysis*, 219-230. San Diego, CA: Academic Press.

Towell, G., Shavlik, J.W. and Noordewier, M.O. (1990). Refinement of approximate domain theories by knowledge-based neural networks. *Proceedings of the Eighth National Conference on Artificial Intelligence*, 861-866. Boston, MA. Menlo Park, CA: AAAI Press.

Uberbacher, E.C. and Mural, R.J. (1991). Locating protein-coding regions in human DNA sequences by a multiple sensor-neural network approach. *Proceedings of the National Academy of Science USA*, 88:11261-11265.

Vogt, G., Etzold, T., and Argos, P. (1995). An assessment of amino acid exchange matrices in aligning protein sequences: The twilight zone revisited. *Journal of Molecular Biology*, 249(4):816-831.

Weber, J.L. and Myers, E.W. (1997). Human whole-genome shotgun sequencing. *Genome Research*, 7:401-409.

Wade, N. (1999). Who'll sequence human genome first? It's up to *Phred*. *The New York Times*, March 23.

Wade, N. (1998). In genome race, government vows to move up finish. *The New York Times*, September 14.

Widrow, B., Rumelhart, D.E. and Lehr, M.A. (1994). Neural networks: Applications in industry, business, and science. *Communications of the ACM*, 37:93-105.

Wilson, R.K. (1999). How the worm was won: The *C. elegans* genome sequencing project. *Trends in Genetics*, 15(2):51-58.