JAMALAH: A System for the Detection of Single Nucleotide Polymorphisms

by

Charles Chen-Cheng

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 28, 1996

Author _____
Department of Electrical Engineering and Computer Science
May 26, 1996

Certified by _____
Eric S. Lander
Thesis Supervisor

Accepted by _____
R. Morgenthaler
Chairman, Department Committee on Graduate Theses

JAMALAH: A System for the Detection of

Single Nucleotide Polymorphisms

by

Charles Chen-Cheng

Submitted to the

Department of Electrical Engineering and Computer Science

May 28, 1996

In Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Dense linkage maps facilitate high resolution mapping, permit epidemiological studies, allow isolation of genes by positional cloning methods, and serve as important frameworks for complete genomic sequencing. As the Human Genome Project moves towards its final stages, scientists have begun to construct a third generation of genetic map, which will eventually consist of thousands of single nucleotide polymorphic markers. The ability to detect these polymorphisms in large scale using new technologies such as the oligonucleotide probe arrays promises to make this new map more powerful than the maps of previous generations.

JAMALAH is a prototype software system for rapid detection of potential single nucleotide polymorphisms. The system allows the identification of potential polymorphisms by sequence comparison analysis alone. Its operation requires minimal human intervention, making the system ideal for repeated use. Experimental studies have validated the system's potential and provided incentive for future expansion.

Thesis Supervisor: Eric S. Lander
Title: Director, Whitehead/MIT Center for Genome Research
Professor, MIT Department of Biology

Contents

Chapter 1
Introduction

The Human Genome Project (HGP) is an ongoing international research effort to construct detailed genetic and physical maps of the human genome and the genomes of several model organisms. The project's ultimate goal is to determine the genomes' complete nucleotide sequences, and to localize the many thousands of genes in each. The biological information generated by the Human Genome Project has already proven to be invaluable towards our understanding of some human genetic diseases, and is expected to have a profound impact on how biomedical research is done in the twenty-first century.

The initial objective of the genome mapping project was to construct STS-based genetic and physical maps of the human and mouse genomes. Aided by the introduction of computers and automated robots to the genome mapping efforts and the development of a variety of new laboratory technologies for genomic and genetic research, this objective was achieved by the end of 1995 [Collins 1995].

The first generation of these genetic maps consists of hundreds of restriction fragment length polymorphism (RFLP) markers, while the second-generation maps have thousands of simple sequence length polymorphism (SSLP) markers. So far, they have proven to be sufficiently dense to provide a solid foundation for the construction of high-resolution sequence maps of the genomes. However, scientists have begun to construct a third generation of genetic map based on single nucleotide polymorphism (SNP). This map promises to be more powerful than the maps of previous generations.

JAMALAH is a modular software system made up of a collection of small programs. The system is designed to identify potential bi-allelic nucleotide polymorphisms by analyzing a large collection of DNA sequences, most of which have been made available by the expressed sequence tag (EST) programs. The system receives as its input sets of sequences, and outputs a list of polymerase chain reaction (PCR) primers which can be used for the ultimate confirmation of the identified polymorphisms by comparative DNA sequencing. The system allows the user to specify a set of parameters by which potential polymorphic nucleotides can be discriminated from probable non-polymorphic ones. It operates automatically with little direct human supervision, and is ideal for repeated use.

1.1 Overview

Figure 1 gives an overview of the JAMALAH system. The sequence assembler accepts as its input a set of sequences which have been segregated into clusters according to their resemblance to each other. The sequence assembler performs assembly on the sequences in each of the clusters. If a

single contig can be successfully assembled, the sequences, along with alignment information, are passed on to the Single Nucleotide Polymorphism Detector (SNPD). SNPD then aligns the sequences to each other, according to the alignment instruction generated by the sequence assembler.

Using a set of user-specified criteria, SNPD examines the aligned sequences nucleotide by nucleotide as it searches for potential single nucleotide polymorphisms. If a potential polymorphic nucleotide is identified, SNPD generates a consensus sequence containing the polymorphic nucleotide, which it passes on to a primer-picking program. The primer-picking program analyzes the consensus sequence and attempts to choose a pair of forward and reverse PCR primers that flank the potential polymorphic locus. If the primers are picked successfully, they are added to a primer purchase order, along with other data such as the consensus sequence and the locations polymorphic nucleotides the primers flank.

The Primer Order and SNP Data Processor translates the output from the primer picking program to its "human-readable" form, and sends the translated information to a file format converter. This file format converter converts the information into Macintosh BinHex-encoded Microsoft Excel files. The encoded files are then sent by electronic mail to the user, and are eventually converted by the user's Macintosh electronic mail client into Microsoft Excel readable files. The user checks the primers in the files, before the primer purchase order is sent to an outside company. Ultimately, sequencing is performed to confirm the existence of the identified polymorphisms, before they are mapped onto the genetic map.

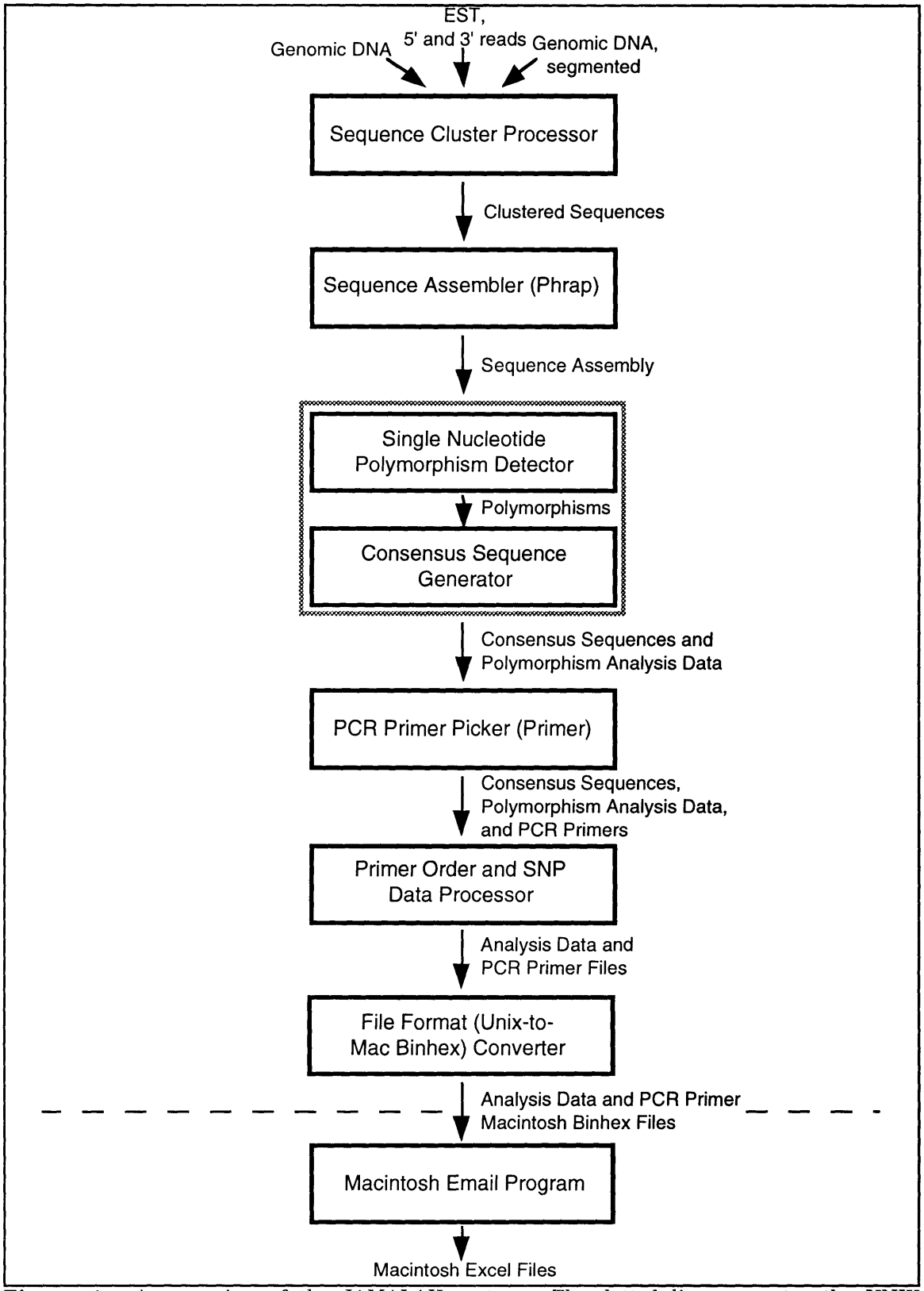

Figure 1: An overview of the JAMALAH system. The dotted line separates the UNIX components from the end-user's desktop Macintosh. Given as its input a dataset of clustered sequences, the system outputs a list of PCR primers which can be used for sequencing to confirm the existence of the single nucleotide polymorphisms identified by the system.

1.2 Background and Motivations

1.2.1 Why Genetic Maps?

Genetic maps have many uses. Sufficiently dense genetic linkage maps facilitate high resolution mapping, permit epidemiological studies, allow isolation of genes by positional cloning methods, and serve as important templates for complete genomic sequencing. They are constructed by determining how frequently two markers (i.e., a physical trait or a detectable DNA sequence) are inherited together. Genes that are close to each other on a chromosome have a much higher chance of being inherited together than genes that lie much farther apart. Genetic studies of families, by determining how frequently two traits are inherited together, can be used to construct genetic maps in which the distances between genes are measured in centimorgans. Two markers one centimorgan apart on the genetic map correlates with a recombinant frequency of one percent.

The first generation of genetic maps were constructed using restriction fragment length polymorphism (RFLP) markers [Botstein 1980]. RFLPs are fragments that are produced when DNA is cut with a restriction enzyme. Due to the fact that nucleotide substitutions occur in the DNA, different individuals occasionally vary in the size of the cleaved DNA fragments. These differences can be assayed by hybridization with radioactively labeled probes and autoradiography.

Although early genetic maps used RFLPs as markers, the use of simple sequence length polymorphisms (SSLPs), or microsatellite markers, has

become the gold standard. SSLPs are polymerase chain reaction assays flanking the short tandem repeats that are scattered throughout the mammalian genomes, including those of humans and mice [Weber 1989; Love 1990]. These repeat DNA markers--the most common repeat type of which is $(CA)_n$--are interspersed at a relatively high frequency, and are useful because their length, or number of repeats, varies among human individuals and between two or more strains of mice[Weissenbach 1992; Dietrich 1992]. Moreover, almost completely automated systems to identify the SSLP markers have been developed. One such system is the SSLP Pipeline developed at the Whitehead/MIT Center for Genome Research [Stein 1994].

As of March 1996, the most detailed and informative genetic map of the mouse, compiled by Dr. Eric Lander and colleagues at the Whitehead Institute/MIT Center for Genome Research, contains 6,580 such SSLPs and almost eight hundred RFLPs [Dietrich 1996]. The average spacing between markers on this map is about 0.2 centimorgans, or approximately 400 kilobases. At about the same time, Dr. Jean Weissenbach and his team at Généthon in Paris have also incorporated 5,264 such microsatellites into the human linkage map, which has an average spacing of 1.6 centimorgan [Dib 1996].

1.2.2 A Current Challenge and One Solution

Although SSLP-based genetic maps are an improvement over RFLP-based maps, they still suffer from some shortcomings, the most significant of which is the lack of means to detect SSLP markers rapidly and in large scale. As is the case for RFLP markers, current methods to detect SSLP markers still rely

heavily on gel-based electrophoresis sequencing techniques, the throughput of which remains to be limited by their sequential nature.

One new class of polymorphic markers, however, holds the potential of solving the bottleneck problem traditionally associated with genetic marker detection. This new class is bi-allelic single nucleotide polymorphisms. Bi-allelic nucleotide markers have several advantages over SSLP and RFLP markers. Two such advantages are their abundance (estimated 1.4×10^7 polymorphic nucleotide sites in the human genome due to neutral mutations [Kimura 1983]) and their suitability for high-throughput parallel detection methods such as allele-specific oligonucleotide hybridization on DNA arrays [Pease 1994]. Another potential advantage is the possibility that these markers can one day be automatically scored. Unlike microsatellites whose detection requires length measurements, single nucleotide polymorphic markers can be scored using high-throughput plus-minus detection systems (like oligonucleotide probe arrays) [Wang 1996a].

The Whitehead Institute/MIT Center for Genome Research has already begun to construct a third generation map of the human genome consisting of bi-allelic single nucleotide polymorphic markers [Wang 199a; Wang 1996b]. Although individual bi-allelic polymorphisms are usually not as informative as microsatellites, multiple closely linked markers can be combined into haplotypes that can be as informative as a repeat polymorphism [Nickerson 1992]. A study has already suggested that a map consisting of these markers can be as informative as a map composed of microsatellites, if its density is higher by a factor of approximately two [Kruglyak 1995].

1.2.3 Finding Single Nucleotide Polymorphic Markers

The National Center for Biotechnology Information (NCBI) has been receiving more than six hundred human complementary DNA (cDNA) sequence fragments daily from the Washington University - Merck Human EST Sequencing Project and other EST sequencing projects, since the EST sequencing program went full-scale in January of 1995 [Marra 1996]. Since then, more than 258,000 single-pass sequence reads have been deposited into the NCBI EST collection, increasing the total number of publicly available ESTs to more than 334,000, as of February 1996.

ESTs, or expressed sequence tags, are DNA sequences generated from single-pass, partial sequencing of oligo(dT)-primed cDNA clones from either the 3' or the 5' end, or both [Adams 1991]. They are sequenced from many different human subjects and cover a wide variety of genes from a wide range of organs and tissues. In many cases, the "same gene" is sequenced from the cDNAs of many different individuals [Adams 1995]. As of August 1995, more than half of all human genes have been sequenced [Boguski 1995].

The large number of publicly available sequences and the inherent high level of redundancy among the large number of EST sequences suggest that it might be possible to detect single nucleotide-based polymorphisms by analysis of the sequences alone. To begin to identify these bi-allelic, nucleotide-based markers hidden within this large, growing collection of sequences, a prototype system called JAMALAH has been developed. The system is discussed in detail in this thesis.

1.3 Overview of the Document

Chapter 2, *Related Work and Background*, reviews various work and research systems related to single nucleotide polymorphism detection technologies. Historic sequence comparison algorithms are also discussed briefly, as some understanding of them will be necessary for discussion in Chapter 5.

Chapter 3, *Single Nucleotide Polymorphism Screening Methodology*, describes in detail the approach used by the JAMALAH system to detect potential single nucleotide polymorphisms. The algorithms used by some of the components in the system are also described.

Chapter 4, *Design and Implementation of the JAMALAH System*, describes the design and implementation of the JAMALAH system. The interactions between various components of the system are explained here.

Chapter 5, *An Evaluation of the JAMALAH System*, presents the results of an actual session with the system.

Chapter 6, *Strengths and Limitations of the JAMALAH System*, discusses the limitations of the current version of the JAMALAH system, in its design and as revealed by actual experimental confirmation of its analysis. Comparisons to Demigloss, a similar system being develop by CHLC, are also made.

Chapter 7, *Conclusions and Future Directions*, makes some concluding remarks about the thesis, and outlines future directions for the JAMALAH system.

Chapter 2
Related Work and Background

This chapter reviews other researches which have addressed issues related to this thesis.

2.1 Oligonucleotide Probe Arrays

High-density oligonucleotide probe arrays, or DNA chips, represent the latest in DNA sequencing technologies. They have applications in the areas of pathogen detection, genetic testing, mRNA expression monitoring, and de novo sequencing, and offers the main advantage of speed. Unlike traditional electrophoresis-based sequencing technologies, nucleic acid sequence analysis using oligonucleotide probe arrays is not limited in its throughput by the sequential nature of the conventional electrophoresis techniques [Lipschutz 1995].

Oligonucleotide arrays are created using a combination of photolithography and oligonucleotide chemistry. Using a proper sequence of photolithographic masks and chemical coupling steps, specific oligonucleotide probes can be

densely displayed on the surfaces of the arrays in an information-rich format [Pease 1994]. Subsequent hybridization of a fluorescently labeled nucleic acid target to the oligonucleotide arrays directly yield sequence information. If the target has regions complementary to probes on the arrays, then the target will hybridize with those probes. In general, probes matching the target hybridize more strongly than probes with mismatches, insertions and deletions. Confocal fluorescence microscopy is used to detect the hybridization pattern of the fluorescently labeled target to the probe arrays. Fluorescence signals from complementary probe-target hybridization are 5 to 35 times stronger than those with single or double base-pair mismatches [Pease 1994].

The significant instability of internal probe-target mismatches, relative to perfect matches, can be exploited to design arrays of probes capable of rapidly discriminating differences between nucleic acid targets and detecting single nucleotide polymorphisms. For example, to determine the identity of unknown nucleotide in a target sequence, four different probes can be synthesized. Each of these probes would be designed to be perfectly complementary to the region containing the unknown nucleotide, except each of them would contain a different nucleotide at the position opposite the unknown nucleotide. When examined by fluorescence microscopy, the probe with the highest complementarity would exhibit the highest intensity. Since the identities and the locations of the probes are pre-defined, the identity of the unknown nucleotide can be determined.

Such an application of oligonucleotide probe arrays is, in fact, being explored by a team at the Whitehead Institute/MIT Center for Genome Research. Collaborating with the California-based company Affymetrix, the Whitehead

team is currently developing probe arrays specifically for assaying bi-allelic single nucleotide polymorphisms. Based on the preliminary experimental results, it might be possible to use high-density oligonucleotide probe arrays for parallel detection of several thousand single nucleotide polymorphisms [Wang 1996a; Wang 1996b].

2.2 Sequence Comparison Algorithms

As will become more clear in Chapter 3, a key procedure that is performed by JAMALAH is sequence assembly. Sequence assembly is the piecing together of several sequence fragments into a single, long contiguous sequence commonly referred to as a contig--a procedure that requires the sequences involved in the assembly to be compared against and aligned to each other. In many cases, the strengths and weaknesses of an assembly programs is directly linked to its choice of sequence comparison algorithm. In the rest of this chapter, a brief treatment of several standard sequence comparison algorithms is given. A basic comprehension of them is necessary to understand some behaviors of the JAMALAH system, which will be discussed in Chapter 5, *An Evaluation of the JAMALAH System*.

2.2.1 Dynamic Programming Algorithms

Various types of algorithms have been used for sequence alignment and comparison; however, dynamic programming algorithms have proved to be most potent. Dynamic programming is a class of algorithms typically applied to optimization problems in which a set of choices must be made in order to

arrive at a solution with the optimal value [Cormen 1994]. The solution is referred to as *an* optimal solution, as opposed to *the* optimal solution, because there may be several solutions that achieve the optimal value. A dynamic programming algorithm has the property that it solves every subproblem just once and remembers the answer to the subproblem. Thus, it avoids the work of recomputing the answers to all previously encountered subproblems.

Dynamic programming algorithms for DNA (or protein) sequence comparison calculate alignment scores that consider possible nucleotide substitutions, insertions, and deletions in the sequences. The alignment score can be calculated as a similarity score (more similar sequences produce a higher score) or a distance score (more similar sequences produce a lower score). In both cases, the algorithms guarantee the calculation of the optimal score for the set of match, mismatch, insertion, and deletion score parameters that are used [Pearson 1992].

To compare two sequences to each other (i.e., to find their optimal alignment to each other), a matrix comparison is done. The matrix comparison provides all possible alignments between the sequences, and can be examined to reveal an optimal alignment of the two sequences. Figure 2 illustrate such a comparison matrix for the alignment of two sequences of characters. The illustration shows that the process of finding an optimal alignment involves computing the score for each cell moving forward and through the matrix to find a path that maximizes or minimizes the final score. As it shows, in many instances, there are more than one way to align two sequences and still obtain the optimal score. Assume for the example shown in Figure 2 that a match has a bonus/penalty weight of +2; a mismatch, deletion or insertion has a weight of

-1; and the goal of the search is to attain the highest score possible. It is then clearly better to align the subsequence "L" in "MATCHLESS" and the subsequence "IN" using a mismatch and an insert/deletion rather than three insertions/deletions. However, if a mismatch has a score of -2 and a deletion or insertion has a score of -1, then the choices become equivalent.

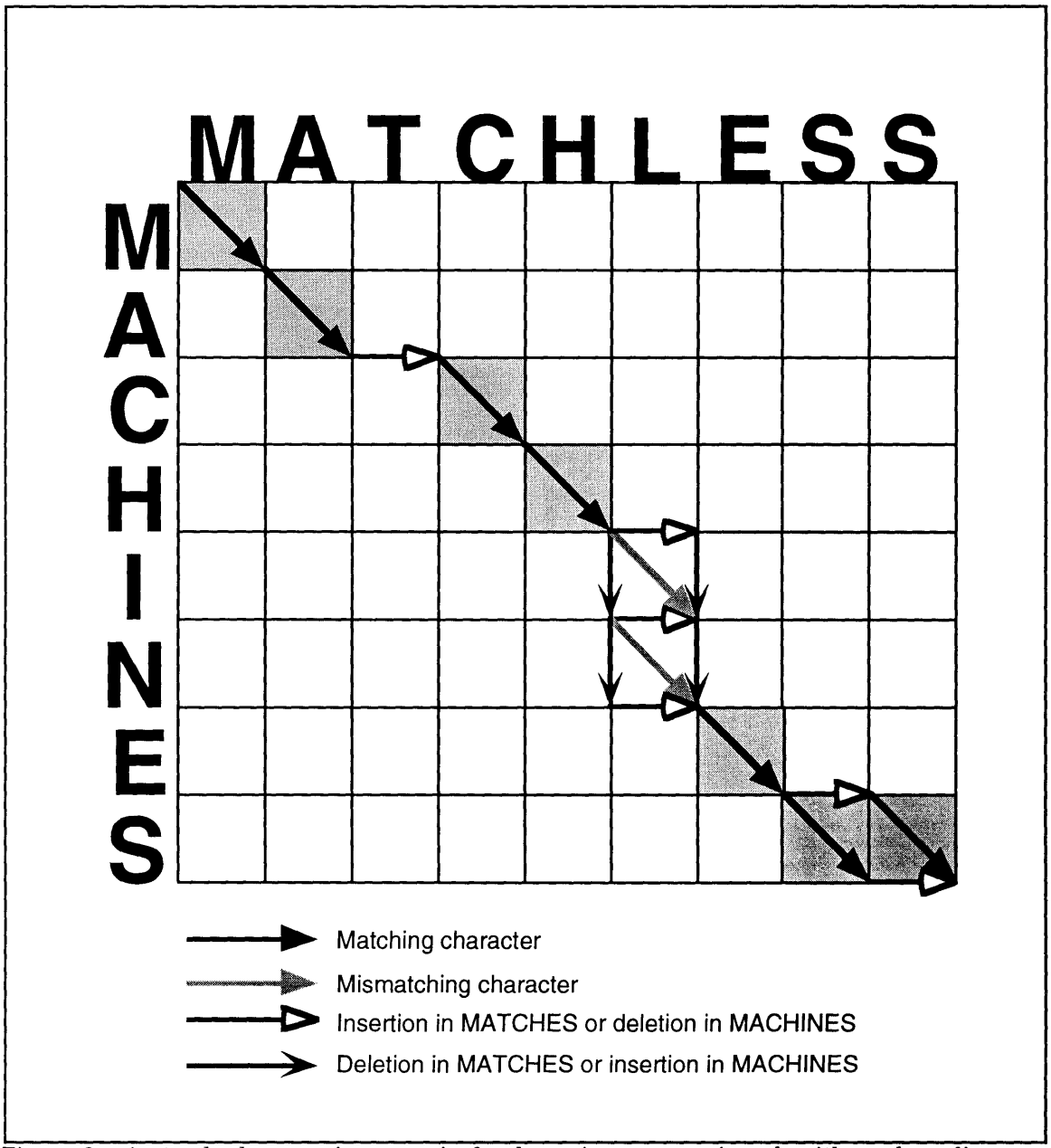

Figure 2: A standard comparison matrix for dynamic programming algorithms that align two sequences. The shaded cells indicate a matching character. Different types of arrows indicate either a mismatch or a need to shift one of the sequences to attain alignment of subsequent characters.

Variations of dynamic programming algorithm for biological problems have been developed and will be discussed next in the chapter. Most efficient implementations of these algorithms for comparing two sequences of length N have a running time proportional to the product of the lengths of the sequences, or $O(N^2)$, and require memory space proportional to the sum of the two lengths, O(N) [Pearson 1992].

2.2.2 Needleman-Wunsch Algorithm for Global Similarity

Needleman and Wunsch have been the first to apply dynamic programming to a biological problem [Ginsburg 1994]. The Needleman-Wunsch algorithm is characterized as one that achieves the best total alignment. It finds all possible matches and assigns weights to every pair according to each pair's similarity, and assigns penalties to insertions and deletions using a gap penalty that is independent of the length of the gap.

The algorithm utilizes a two dimensional array as its comparison matrix. All alignments between the two sequences are represented by paths through the matrix. There are a number of ways to move through the matrix, and the sum of weights, or score, is generated by adjusting the score according to the direction of the path the search takes, as illustrated in Figure 2. An optimal alignment is eventually found by tracing back through the matrix from the cell with the highest score [Needleman 1970].

2.2.3 Smith-Waterman Algorithm for Local Similarity

Unlike the Needleman-Wunsch algorithm, the Smith-Waterman algorithm achieves the optimal local sequence similarity, and is more suitable for functional or structural analogies. However, when the sequences are highly similar over a long stretch, the Smith-Waterman algorithm will also achieve global alignment [Smith 1981].

For the Smith-Waterman algorithm, every cell in the matrix is considered a starting point and no penalties are attached to overhangs of either sequence; weights and penalties are assigned as usual to matches, mismatches, insertions, and deletions. During a search through the comparison matrix, if the total score falls below a certain threshold at a cell, the current path of the search is stopped. The score is reset, and the cell is then used as a new starting point for a search. Like the Needleman-Wunsch algorithm, the Smith-Waterman algorithm identifies regions of greatest similarity by backtracking.

In comparison to the Needleman-Wunsch algorithm, the Smith-Waterman algorithm is more ideal for use in an assembly program. Assembly often involves piecing together several pieces of sequences that may not exhibit great global similarity but have regions that are highly similar, and overlap localized regions on other sequences. Therefore, the use of an algorithm that is sensitive towards finding these highly similar localized regions is more appropriate.

2.3 Sequence Assembly Software

One of the most important but labor intensive procedures in biology is sequence assembly. Sequence assembly is the piecing together of related sequence fragments into one contiguous sequence called a contig. The procedure involves several stages. First, the sequence of each fragment must be accurately entered into the computer. Next, the vector regions must be recognized, and either deleted or marked so they are not included in the assembly. Then the fragments must be accurately aligned and formed into contigs. Finally, the overlapping fragments in the contig must be examined and the conflicts and ambiguities in the sequences must be resolved to produce a consensus sequence.

Most programs available today are capable of handling potential deletions, insertions, and substitutions in the sequences, with a few programs such as XBAP and PC/Gene, which expect carefully edited sequences as input, being the more notable exceptions [Miller 1994]. In general, more capable algorithms search exhaustively for all the best possible initial alignments, even after some alignments have been accepted at some stringency level. Failure to do so usually results in erroneous alignments. For many algorithms, when the consensus sequence is ambiguous, many of them resort to the "majority wins" rule to handle ambiguity that might have arisen due to gaps and nucleotide substitutions [Miller 1994]. In many of the programs, Insertion/deletion misalignment of the sequences is a significant source of consensus error or ambiguity.

One of the more powerful assembly programs available today is the phragment assembly program, or Phrap, by Phil Green. Phrap is a program for assembling shotgun DNA sequence data [Green 1995]. One of its key features is its ability to use a combination of user-supplied and internally computed data quality information to improve accuracy of assembly in the presence of repeats--a major source of difficulty in sequence assembly [Green 1995]. It utilizes a very efficient implementation of the Smith-Waterman algorithm to perform pair-wise comparisons of the sequences, to generate internal quality scores for all nucleotide positions on the sequences, and to delineate the high quality regions of the sequences that it can use to construct the contig.

Phrap is capable of performing large-scale assembly of sequences. Apart from available memory of the computer running the program, there are no intrinsic limits to the number or the length of sequences that Phrap can assemble in one session. Moreover, the speed of assembly by Phrap is limited mainly by the number of repeats in the target sequences, rather than the total number of sequences, since repeats disproportionately increase the number of Smith-Waterman comparisons that need to be performed [Green 1995].

Chapter 3
Single Nucleotide Polymorphism Screening Methodology

This chapter describes the current protocol used for screening sequence data for bi-allelic single nucleotide polymorphisms. The procedures performed by the JAMALAH system and the algorithms used by the system are described. Exact values for the parameters used by the algorithms are not offered in this chapter, but are withheld for later discussion in Chapter 5, *An Evaluation of the JAMALAH System*. The relevant design issues pertaining to the implementation of the system are described later in Chapter 4. A flowchart of the protocol is shown in Figure 3.

3.1 Sequence Clustering

The first major step in the protocol (step 1 in Figure 3) involves collapsing all the available sequences in NCBI GenBank into clusters of sequences that are most likely to be derived from the same gene. This is necessary to facilitate the subsequent sequence assembly, as it is difficult to perform high quality assembly with a large number of sequences in a timely fashion.

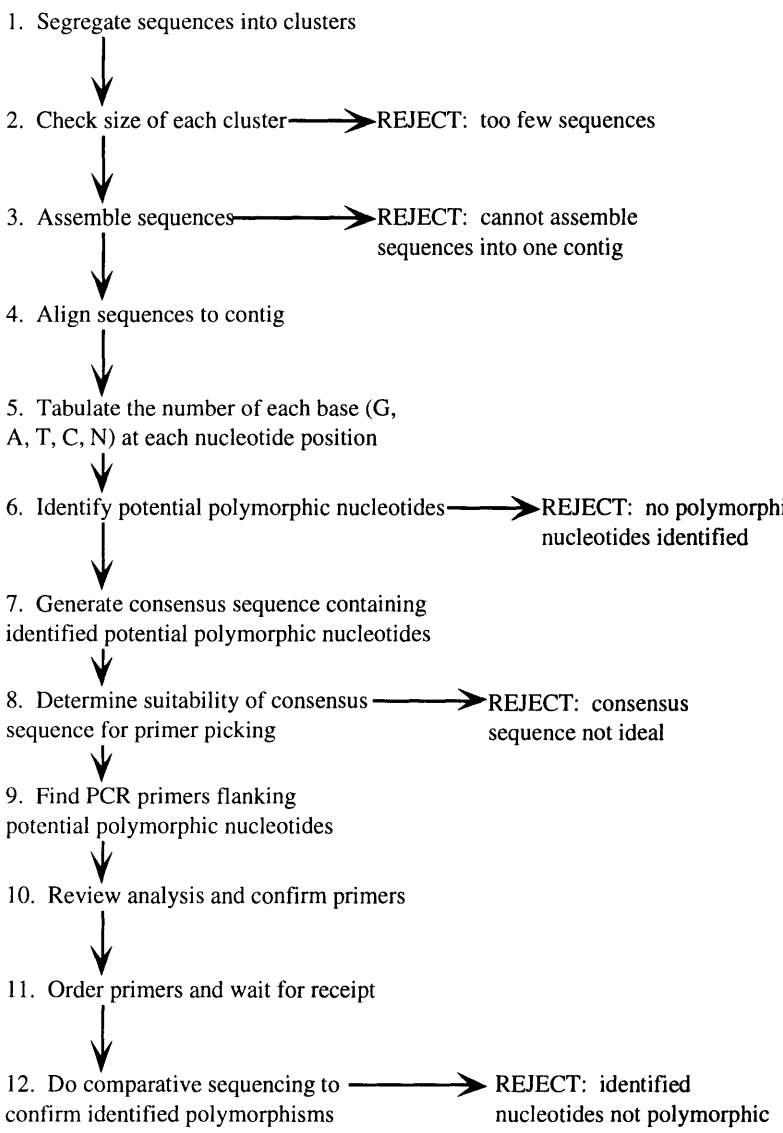

Figure 3: Single Nucleotide Polymorphism Screening Protocol. The first six steps of this protocol are the identification of potential single nucleotide polymorphisms from sequence data. The remaining steps are performed to confirm the existence of the identified polymorphisms.

Clustering is not a trivial task for the following reasons: GenBank, the official genetic sequence database of the National Institute of Health, is a comprehensive and historical collection of sequence data and thus contains a

large number of sequences, many of which are multiple representations of essentially the same data [Boguski 1995]. A gene can have both genomic clones and messenger RNAs in their partial or full-length form. Different gene sequence entries can possess differing amounts of flanking and intron sequences, and messenger RNA sequences can contain variation because of alternate splicing. In addition, ESTs are both fragmentary and have a higher error rate.

The JAMALAH system does not perform sequence clustering. Instead, it uses as its input a set of previously clustered sequences, called the UniGene set, obtained over the Internet from the National Center for Biotechnology Information. The clusters are made up of sequences that share statistically significant similarity in the 3' untranslated region [Boguski 1995; Schuler 1996].

3.2 Sequence Assembly and Alignment

Each sequence cluster is checked to see if it contains a sufficient number of sequences for potential single nucleotide polymorphisms to be identified (step 2 in Figure 3). If the cluster have enough sequences--suggesting that it might have a sufficiently high redundancy to allow any variation detected in the sequences to be statistically significant--assembly of all the sequences in the cluster is performed (step 3 in Figure 3). The sequences must be aligned to achieve maximum, if not complete, overlap, as illustrated in Figure 4, so that variations of a single nucleotide nature among the assembled sequences can be easily located.

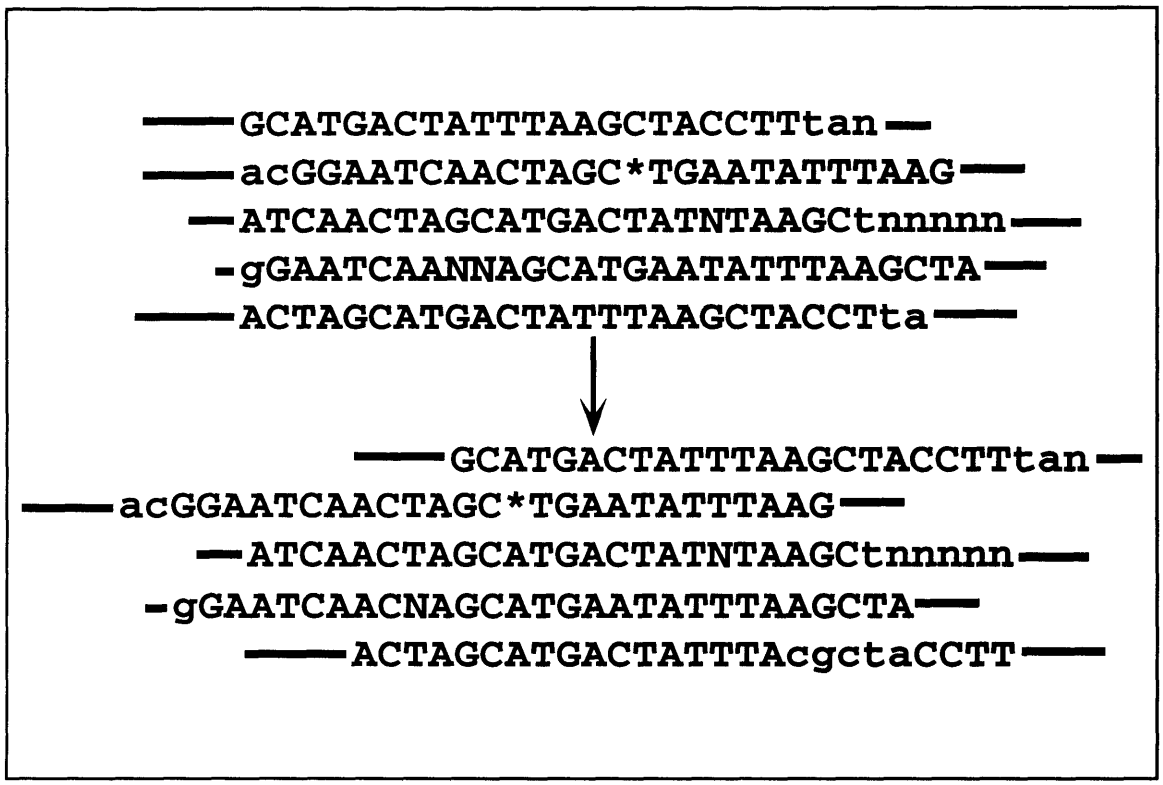

Figure 4: Sequence assembly. To assemble the sequences in a cluster, the sequences are compared against each other to maximize overlap.

To carry out sequence assembly, JAMALAH makes use of a readily available program called phragment assembly program, or Phrap, provided courtesy of Phil Green. Phrap is a program for assembling shotgun DNA sequence data [Green 1995]. One of its key features is its ability to use a combination of user-supplied and internally computed data quality information to improve accuracy of assembly in the presence of repeats--a major source of difficulty in sequence assembly [Green 1995].

Since the emphasis of this thesis is not on sequence assembly algorithms, only a brief summary of the main steps in Phrap's assembly procedure [Green 1995] is given here. Phrap reads in all the sequences that are going to be involved in the assembly, and trims any low quality sequences that consist

predominantly of a run of single bases (for example, poly A) from the ends of the sequences. This is necessary since such sequences can create spurious matches in the subsequent alignment. Phrap then constructs complementary sequences to all the input sequences, compares them and their complements, and rejects duplicate sequences. Phrap also checks for probable vector regions and eliminates them from further analysis.

Phrap then performs pairwise comparison of the sequences and their complements using an efficient implementation of the Smith-Waterman-Gotoh algorithm. This is done to determine the degrees of similarities among the sequences, and their best alignments to each other. The alignment information from the pairwise comparisons is used to revise sequence quality information in the following fashion. If part of a sequence is confirmed by another sequence to which it is complementary (either a user-supplied sequence or the complementary sequence of a user-supplied sequence), the quality values of the nucleotides within the confirmed region are augmented to reflect increased confidence in the likely accuracy of that region of the sequence. Similarly, if part of a sequence is confirmed by another similar sequence, its quality value is also augmented, though to a lesser degree. After this is done, Phrap attempts to identify probable chimeric and deletion reads, and excludes them from subsequent assembly analysis.

Based on qualities of matching and mismatching nucleotides, Phrap computes loglikelihood ratio (LLR) scores for various alignments of the sequences, and identifies the highest quality sequences among them (ones with highest LLR scores among several overlapping sequences). It then constructs contig layout for the sequences using a greedy algorithm, and generates a contig sequence

using the highest quality parts of the layout sequences. Finally, the individual sequences are then aligned to the contig sequence.

After Phrap has completed sequence assembly, a separate program in the JAMALAH checks the result to make sure that only a single contig sequence was generated. If multiple contig sequences are assembled from the sequences in a cluster, then the cluster is rejected from further analysis.

3.3 Identification of Potential Polymorphic Nucleotides

If the sequences in a cluster can be successfully assembled into one contig sequence by the sequence assembler, this suggest that the sequences are probably highly similar. Using the alignment information generated by the sequence assembler Phrap, the Single Nucleotide Polymorphism Detector (SNPD) reconstructs the overlapping of the sequences by aligning the original cluster sequences or their complements to the contig (step 4 in Figure 3). SNPD then performs polymorphism identification analysis on the aligned sequences using a set of user-specified parameters to discriminate nucleotide positions likely to contain polymorphisms from those where differences are likely due to sequencing or alignment error.

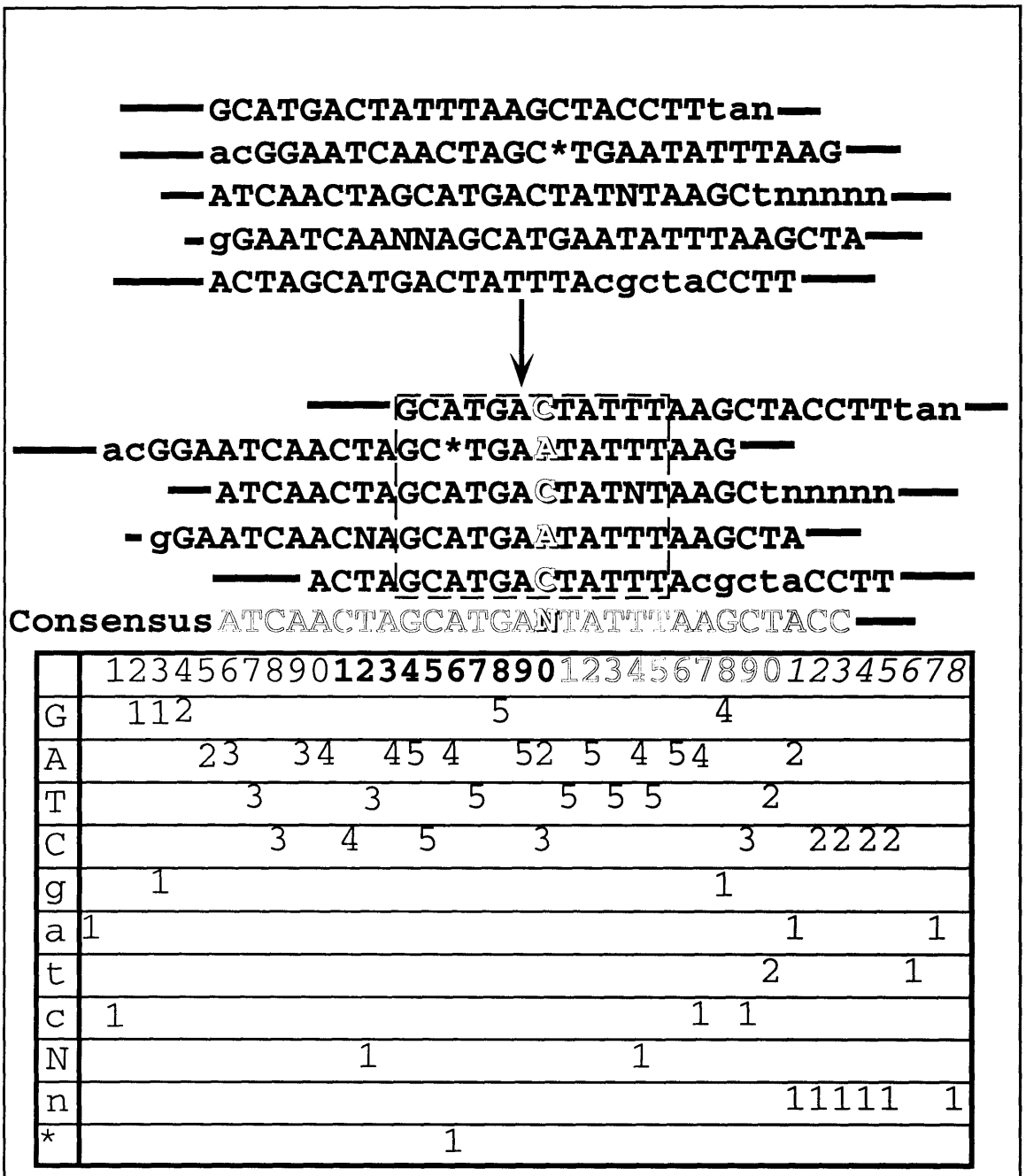

Figure 5: The number of each nucleotide (G/A/T/C/g/a/t/c) at every contig position is tabulated. Ambiguous nucleotides (N/n) and padding characters (*) are also counted. Padding characters are inserted into the gaps of certain sequences to allow their alignment to the contig (or consensus) sequence. High-quality nucleotides, as determined by Phrap, are displayed in uppercase. Low-quality nucleotides are shown in lowercase. The nucleotides displayed in outlined font (in the center of the picture) constitute an identified polymorphic nucleotide, with a minor allele frequency of 0.4. The quality check window is enclosed in dashes.

To identify potential nucleotide-based polymorphisms from the sequence assembly, SNPD tabulates the numbers of guanine (G/g), adenine (A/a), thymine (T/t), cytosine (C/c), no-call (N/n), and padding character (*) at each nucleotide position in the sequence assembly, as demonstrated by the example in Figure 5 above. No-call "nucleotides" are present in sequences, where the exact identify of the nucleotide is not known, i.e., the sequencing data is noisy at that particular nucleotide. Padding characters are characters inserted by Phrap (and other popular sequence assembly programs) into the gaps of certain sequences to allow the sequences to be aligned to the contig and other sequences.

For each nucleotide position, the most frequently-occurring nucleotide, the so-called major allele, is identified, and the allelic frequency for each nucleotide is calculated using the formula below:

$$\text{Allele Frequency (X)} = \frac{\text{Occurrence of X}}{\text{Occurrence of X} + \text{Occurrence of Major Allele}}$$

If the allele frequency for nucleotide X is at least `min_minor_allele_freq`, and nucleotide X is confirmed by at least `min_reads_per_allele` reads, i.e., there are at least two sequences that show nucleotide X at that particular position in the sequence assembly, then nucleotide X is marked as a potentially valid base for that particular position. If a nucleotide position has multiple potentially valid bases, then that nucleotide is temporarily marked as being potentially polymorphic. After this, a window that extends `qual_window_size` bases to the left and to the right of the potential polymorphic nucleotide is examined (see Figure 5). The total number of no-

calls and padding characters is determined. If this number exceeds qual_window_threshold percent of the total number of bases within the window, then the potentially polymorphic nucleotide in question is rejected from consideration as a potential polymorphic site. This last check ensures that the identified potential polymorphisms are not due to poor sequence assembly or poor sequence reads.

Many sequence analysis programs provide information about the quality of a sequence by showing the high quality regions of the sequence in uppercase and the low quality regions in lowercase. SNPD has been designed to take advantage of this common feature. The user is allowed to specify whether or not the SNPD algorithms should only consider high quality parts of the aligned sequences. This is done by specifying the parameter use_high_qual_only as being TRUE or FALSE. (Please refer to Appendix A for a listing of the default values for the parameters that are used by the SNPD algorithm. These parameters include min_minor_allele_freq, min_minor_allele_freq, min_reads_per_allele, qual_window_size, qual_window_threshold, among others.)

3.4 Generation of Consensus Sequence

If a potential nucleotide-based polymorphism is identified, SNPD will attempt to generate a consensus sequence, so that the polymorphism can be ultimately confirmed by comparative DNA sequencing (step 7 in Figure 3). The identity of each nucleotide position in the consensus sequence is determined by considering the identities of all the nucleotides at the corresponding position in the aligned sequences. Starting from the identified potential polymorphic site,

SNPD sequentially scans the sequence assembly, nucleotide by nucleotide, in the upstream direction. At each nucleotide position, it makes the following determination. If all the overlapping sequences exhibit the same nucleotide X at that position, it generates nucleotide X at the corresponding position in the consensus sequence (plain face nucleotides in Figure 6). If the number of occurrences of the most frequently occurring nucleotide X is at least 3 times greater than that of the second most frequently occurring nucleotide Y (no-call and padding character are considered), nucleotide X is generated in the consensus sequence (underlined nucleotides in Figure 6); otherwise, a no-call (N) is used instead (italicized nucleotides in Figure 6). A no-call is always generated for nucleotides identified to be potentially polymorphic (shadowed nucleotide in Figure 6). As shown in Figure 6, this determination is performed at every nucleotide position until the number of aligned sequences that overlap drops below `min_consensus_overlap`. The same procedure is then performed in the downstream direction.

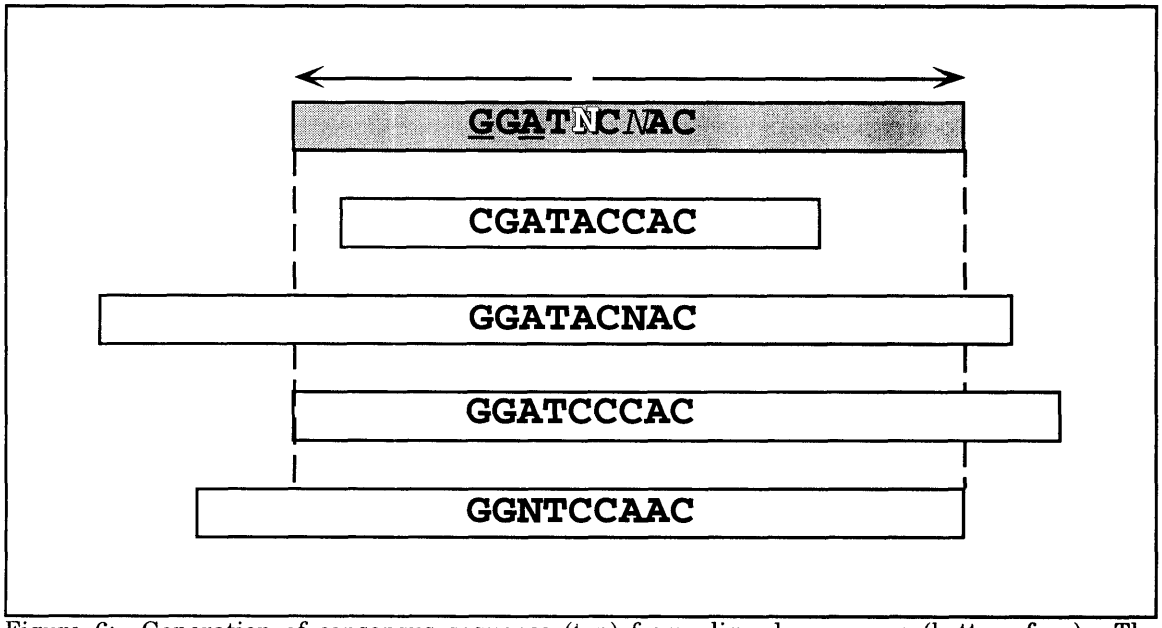

Figure 6: Generation of consensus sequence (top) from aligned sequences (bottom four). The sequence is generated as described above with the default setting. The value of `min_consensus_overlap` is 3 for this example.

After a consensus sequence is generated, the sequence is checked to ensure that it contains at least one unambiguous region of at least `min_primer_length` bases both upstream and downstream of the identified potential polymorphism (regions to which the forward and reverse polymerase chain reaction (PCR) primers can potentially anneal; step 8 in Figure 3). The existence of these regions would suggest that a forward primer and a reverse primer could potentially be picked upstream and downstream of the identified potential polymorphism, respectively, such that the resulting PCR product would span the identified potential polymorphism and allow its identity to be confirmed. In addition, the region for the forward primer must be at least `min_margin_upstream` bases upstream of the identified polymorphism; similarly, the region for the reverse primers must be at least `min_margin_downstream` bases downstream of the polymorphism. These margins are necessary, as nucleotides too close to the PCR primers cannot be read accurately from the sequencing data [David Wang 1996b]. Lastly, the sequence is checked to ensure that it can potentially have a PCR product at least `min_consensus_length` bases long, as a PCR product that is too short is inherently noisy and difficult to read after sequencing [David Wang 1996b]. If the consensus sequence survives all of the tests, it is considered a usable consensus sequence. (See Figure 7 for an illustration of these various regions.)

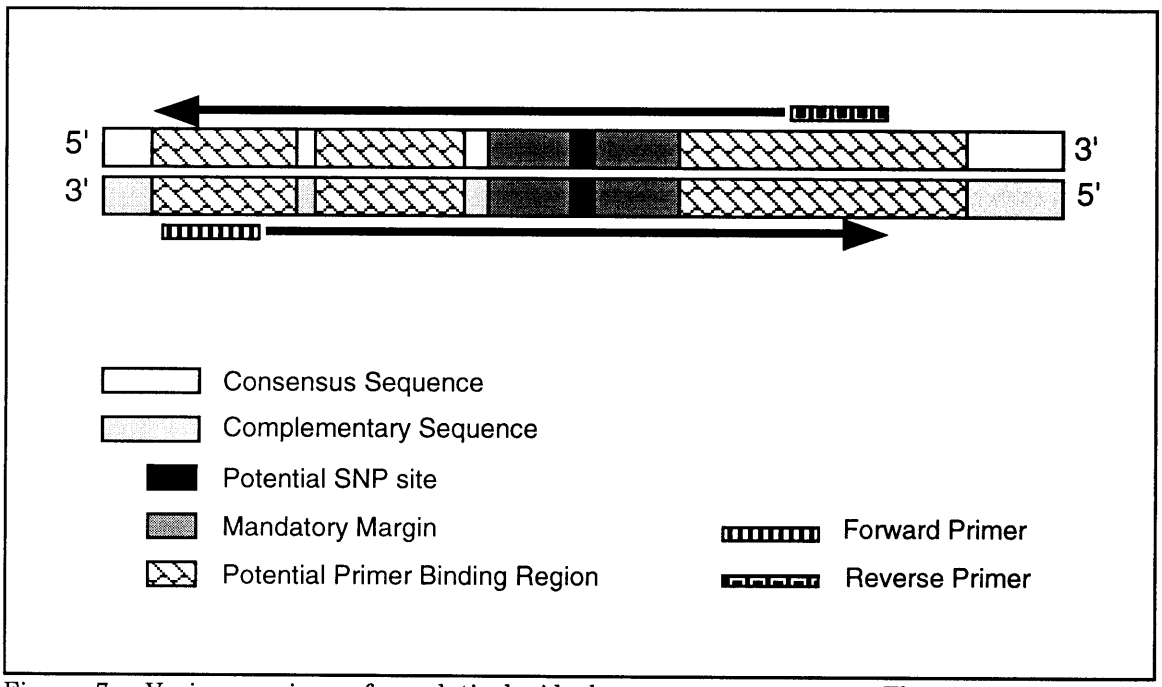

Figure 7: Various regions of a relatively ideal consensus sequence. The top strand is the consensus sequence generated by SNPD Detector.

In some instances, multiple potential polymorphic nucleotides can reside on the same consensus sequence. When this is the case, SNPD will attempt to generate one consensus sequence containing all the potential polymorphisms, if possible. Whether this is possible or not depends on how far apart the polymorphisms are, since an optimal length for PCR product for the dye primer sequencing technology used for comparative DNA sequencing is between 200 and 400 bases [Wang 1996b].

The parameters described can be arbitrarily chosen to generate a reliable consensus sequence. So far, using the default values for the parameters (see Appendix A for a listing of the values), this algorithm has proven to be quite reliable. Actual experimental results which support this claim are discussed later in Chapter 5.

3.5 Primer Picking

Once the consensus sequence is generated, primer picking is performed (step 9 in Figure 3). The selection of the forward and reverse primers is carried out using a piece of software called PRIMER. The original version of PRIMER was developed at the Whitehead Institute/MIT Center for Genome Research by Steve Lincoln, Mark Daly, and Eric Lander [Primer 1996]. The program picks primers for PCR reactions, according to a set of conditions specified by the user. Primer considers factors like melting temperature, concentrations of various solutions in PCR reactions, primer bending and folding, and other conditions when attempting to pick the optimal primer pair for a reaction.

After selecting all the primer pairs for all the consensus sequences, JAMALAH automatically generates a primer purchase form, which is sent by electronic mail to a human authority. The primers in the purchase form are verified by the authority (step 10 in Figure 3), before an order is placed with a biotechnology company (step 11). Sequencing is performed when the synthesized primers arrive from the company (step 12). The result of the sequencing is used to confirm the existence of the identified potential polymorphisms.

Chapter 4
Design and Implementation of the JAMALAH System

This chapter describes the principles that guide the design of the JAMALAH system. It also describes the software implementation details of the system. A complete listing of the programming code is provided in Appendix F.

4.1 Design Objectives

The following objectives were set for the design of the system:

- Must be functional and robust
- Require little or no human supervision to operate
- Modular design
- Easy for user to manipulate and analyze output
- Portable on UNIX workstations
- Use existing software where possible
- Easy to test, maintain and upgrade

Some of design principles and goals stated above were motivated by a previous paper describing the informatics group at the Whitehead Institute/MIT Center for Genome Research [Stein 1994]. The paper delineates the following important points about the dynamics of a typical genome mapping laboratory:

- Laboratory protocols are always changing and being improved.

- Laboratory scientists prefer the user-friendliness of desktop personal computers, particularly the Apple Macintosh computers. They are familiar and satisfied with the "off-the-shelf" software that runs on their Macintosh computers, and prefer not to have to master new applications

- Electronic mail is a good way to inform the user upon the completion of the tasks that require more than a few minutes, or even a few seconds, to complete.

Most of the objectives listed at the beginning of this chapter are achieved by the current prototype of the JAMALAH system. To accomplish the objectives, the followings decisions were made at a very early stage of developing JAMALAH. The UNIX operating system was chosen as the operating system on which the system would be built. The choice of UNIX as the operating system is advantageous for several reasons: First, in general, UNIX computer workstations are computationally more powerful than most desktop computers that are available today, and the proven reliability of their UNIX operating system is highly desirable. Second, UNIX is an open system that runs on many platforms; many programs written for UNIX are, therefore,

portable and can run on many different types of machines. Third, the UNIX operating system has been used by the scientific research world for a long time, many UNIX applications for analyzing biological data (including sequence assembly and primer-picking programs) are available in the public and commercial domains [Stein 1994]. This would accelerate the development of JAMALAH, as appropriate existing software could be adapted to perform certain tasks involved in detecting single nucleotide polymorphisms; this avoids the writing of thousands of lines of new code. Lastly, the superior inter-process communication architecture of the UNIX operating system promotes a tool-based approach to designs, and facilitates the design of modular systems. Modular, as opposed to monolithic, systems offer the advantages of ease of maintenance, ease of upgrading, and simplicity of testing. A modular design is also important for easy integration of existing software into the JAMALAH system. However, the UNIX operating system is not the preference of most laboratory scientists, as previously discussed [Stein 1994]. Attempts to remedy this problem have been made and will be described.

4.2 Implementation of the System

Figure 8 shows the components of the JAMALAH system. The components consist of a collection of UNIX programs connected serially through UNIX pipes. Simply, a pipe is a way to connect the output of one program to the input of another program directly, without generating any temporary intermediate file. Most of the programs within this so-called pipeline have been implemented as UNIX input-output filters (programs that read in some input, perform a transformation on it, and write some output). More specifically,

they receive their input from some UNIX standard input and generate output to some UNIX standard output.

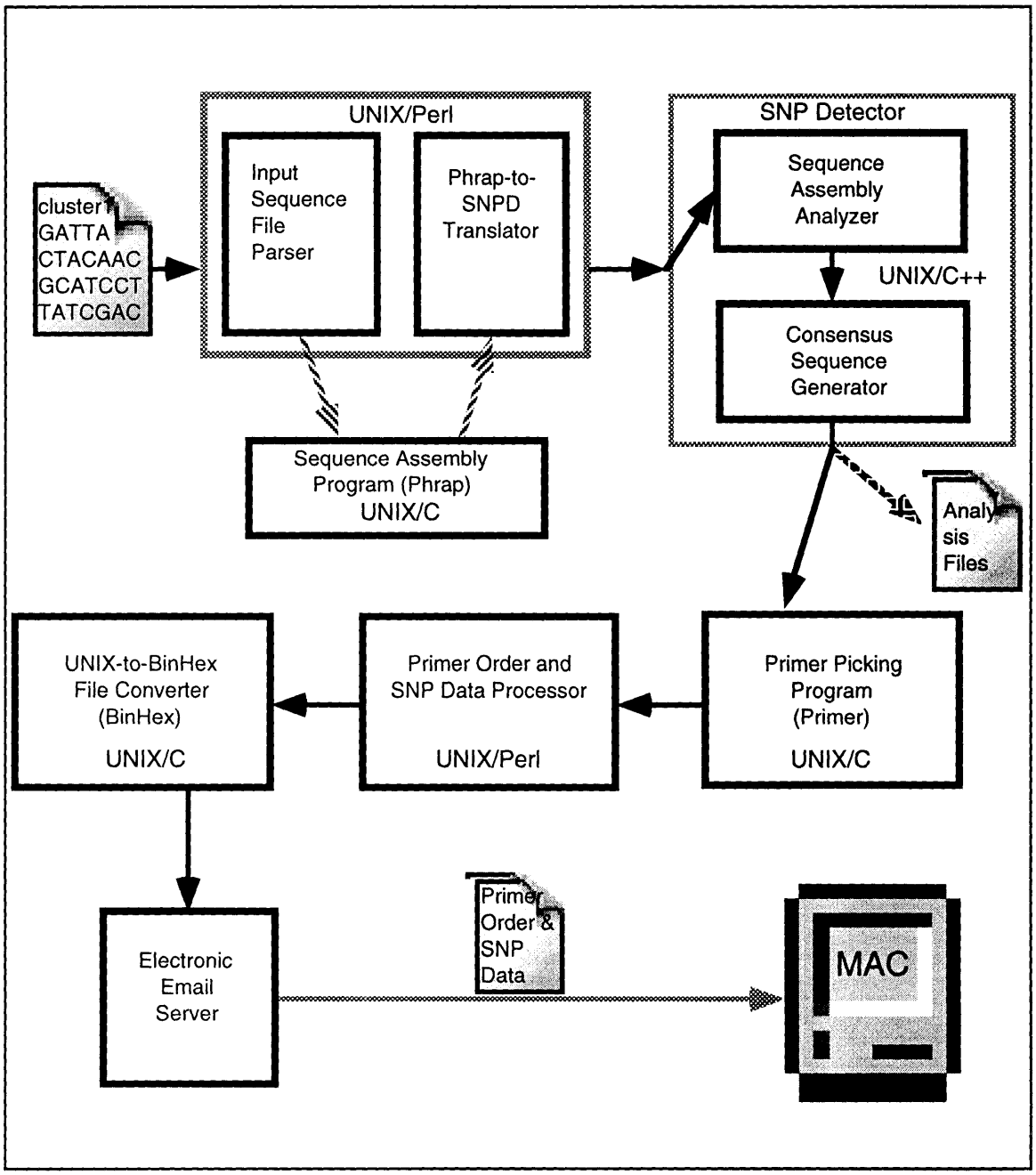

Figure 8: Components of the JAMALAH Pipeline. Except for the user's electronic mail client which runs on an Apple Macintosh PC, all components shown run on the UNIX operating system. They include analytical programs such as Phrap, SNP Detector and PRIMER; and a number of auxiliary programs whose jobs are to act as translators between the other components. Each component in the "pipeline" is implemented as an UNIX input-output filter, and the components are connected together through UNIX pipes. The system reads in a large sequence file through the input end of the pipeline. The user eventually receives the results of the SNP detection analysis in two Microsoft Excel files on a Macintosh Computer, if he or she so desires.

In a typical session for JAMALAH, the program at the beginning of the pipeline reads in the sequence data from a file residing on an UNIX workstation. Some time later, the program at the very end of the pipeline gathers up the final results of the analysis, formats them, and sends them to the user via electronic mail. Each component in the JAMALAH system will be described in more detail below.

4.2.1 Sequence File Parser and Phrap-to-SNPD Translator

The first component in the JAMALAH pipeline is a Perl program that parses the UniGene sequence data (UniGene cluster format) into a format that the sequence assembler Phrap can accept (FASTA format), and converts the result of the sequence assembly into a format that SNP Detector, the next component in the pipeline, can understand. This first program is made up of two sub-components--Input Sequence File Parser and Phrap-to-SNPD Translator. (See Figures A-B in Appendix B for samples of the UniGene format and the FASTA format.)

The Input Sequence File Parser sub-component reads the input file containing all the clustered sequences, extracts all the sequences in a cluster, writes the sequence to a temporary file, and executes the sequence assembler (Phrap) to perform assembly on the sequences in the temporary file. When the sequence assembler is done assembling the sequences, the Phrap-to-SNPD Translator sub-component reads the analysis files generated by the sequence assembler, reformats the assembly and alignment data in one of the files, and sends the reformatted data to the next component, the SNP Detector. For each cluster it looks at, this program also checks to make sure that the cluster contains a

sufficient number of sequences, and that the sequences are successfully assembled into one contig by the sequence assembler. The steps described are repeated, until all the clusters have been read and subjected to sequence assembly.

Perl (Practical Extraction and Report Language) was chosen as the programming language for this component, because Perl offers elegant and powerful means to manipulate text-based data [Wall 1991]. The language has been optimized for scanning and extracting information from large amounts of arbitrary text, and offers sophisticated pattern matching techniques. In addition, Perl programs are extremely portable.

4.2.2 The Sequence Assembler

To carry out sequence assembly, JAMALAH makes use of a readily available program called phragment assembly program, or Phrap, as previously described in Chapter 3, *Single Nucleotide Polymorphism Screening Methodology*. Phrap is a relatively large C program for assembly of DNA sequences. It is widely used and has been proven to be quite robust. It is important that the sequence assembly component is robust and that the result of its assembly is as accurate as possible, since a low quality assembly will significantly affect subsequent analysis by other components in the pipeline.

When Phrap is invoked by the Input Sequence File Parser, it reads in all the sequences from the file generated by the Sequence File Parser It then performs assembly analysis on the sequences, and outputs the result in

several files. As described in the previous section, the analysis data in one of the files, specifically the so-called "ace" file, is read by the Phrap-to-SNPD Translator and eventually used for analysis by SNPD. The "ace" file contains the following information: the sequences of the padded contig and sequence reads, the locations of the sequence reads with respect to the contig sequence, and information about how the contig sequence is pieced together as a mosaic of the high quality parts of the reads. (See Figure C in Appendix B for additional information on the "ace" format.)

By adapting Phrap for the JAMALAH system, the development time for the system has been dramatically reduced. The modular design of the JAMALAH system has made the integration process relatively painless, and the use of Perl makes the task of extracting information from the "ace" file generated by Phrap straightforward.

4.2.3 The Single Nucleotide Polymorphism Detector (SNPD)

The next major component in the pipeline is the Single Nucleotide Polymorphism Detector (SNPD), sometimes also referred to as the SNP Detector. The SNPD program is written in C++. This particular programming language's strong support for code abstraction has been most helpful in the development of the program. Its excellent performance in terms of speed is also desirable.

Through its UNIX standard input, the SNPD program receives the contig sequence, sequence reads, and the assembly alignment information generated by the sequence assembler. The Phrap-generated data have been translated

by the Phrap-to-SNPD Translator into a format SNPD expects (see Figure D in Appendix B for a sample of the format). SNPD aligns each sequence to the contig according to the alignment instructions provided by Phrap, and performs SNP identification analysis on the aligned sequence using the algorithm described in Chapter 3, *Single Nucleotide Polymorphism Screening Methodology*. If it identifies a potential single nucleotide polymorphism, it would attempt to generate a consensus sequence using an algorithm also described in Chapter 3.

Through its UNIX standard output, it writes out the consensus sequence and other information the primer-picking program--the next component in the pipeline--needs to select a pair of primers that flank the identified potential SNP. The format it uses is called Boulder IO, a text-based, input/output format developed at the Whitehead Institute/MIT Center for Genome Research [Boulder IO 1996]. Basically, the format is a series of tag-value attributes in the form `TAG=VALUE`. For example, the tag for an attribute pair could be "`SEQUENCE`" and its corresponding value could be "`GNAATTCTATCAT`". (A sample of the output SNPD generates is shown in Appendix C. Additional explanation is also provided in Appendix C.)

The SNP Detector can generate detailed analysis reports, including a complete compilation of the analysis for each nucleotide position. These additional reports are currently produced as plain UNIX text files.

4.2.4 The Primer Picker - PRIMER

The primer picking component comes immediately after SNPD. The specific program JAMALAH employs for this component is PRIMER, a C program developed by the Whitehead Institute/MIT Center for Genome Research [PRIMER 1996]. PRIMER picks primers for PCR reactions, according to a set of conditions and parameters specified by the user. These conditions and parameters include melting temperature, optimal length of primer, concentrations of various solutions in PCR reactions, location of the target region. If the value for a condition or parameter is not specified provided, PRIMER resorts to using the default value [PRIMER 1996].

The PRIMER parameter `TARGET` deserves special discussion here. It allows JAMALAH to specify a region or a list of regions in the consensus sequence that the chosen primers should flank but not touch. Specifically, SNPD provides PRIMER with the instruction to pick the primers around the identified potential SNP site and to pick them far enough upstream and downstream of the site so that the primers do not intrude the mandatory margins discussed in Section 3.4, *Generation of Consensus Sequence*. Like all other parameters, this parameter is specified using the Boulder IO format, as shown in Appendix C.

PRIMER communicates solely through UNIX standard input and output using the Boulder IO format described in the previous section and illustrated in Appendix C. A sample of the output that PRIMER has generated from data provided by SNPD is also offered in Appendix C.

4.2.5 The Primer Order/SNP Data Processor and BinHex

The output from the primer picker is directed to the Primer Order and SNP Data Processor. The Processor is a simple Perl program that parses the Boulder IO data generated by PRIMER to produce human-readable primer purchase orders and SNP data-related files. After the "files" have been generated, they are passed via UNIX pipe to a program called BinHex. BinHex, by Lincoln Stein, is an utility program that encodes text or binary data using the Macintosh BinHex format. It allows one to convert text files in such a way that when the files are decoded on a Macintosh, they appears as normal Macintosh file. In their encoded state, these encoded files are in ASCII and can be sent easily as electronic mail messages.

The JAMALAH system uses BinHex to encode the "files" generated by the Primer Order and SNP Data Processor as files which would eventually be recognized by the Macintosh operating system as belonging to the program Microsoft Excel, a popular spreadsheet application for the Apple Macintosh. The encoded files are then sent to the user via electronic mail. Since BinHex is a format supported by many electronic mail clients for the Macintosh computers which are preferred by laboratory scientists, files encoded in BinHex are usually recognized and automatically decoded when the user reads his or her mail. Moreover, many electronic mail clients provide automatic notification of new mail arrival. Thus, sending the primer purchase order and SNP data to the user upon the completion of the time-consuming SNP analysis is an elegant way to inform the user that the analysis has been completed. This approach frees the user from constantly checking up on the progress of the analysis. An additional advantage is obtained by providing the

results of the analysis to the user as Microsoft Excel spreadsheets. He or she is able to manipulate the data using Microsoft Excel, which many of the laboratory scientists have become accustomed to.

A snapshot of the primer purchase order and SNP analysis data as Microsoft Excel spreadsheets is provided in Appendix D.

Chapter 5
An Evaluation of the JAMALAH System

The robustness and functionality of the JAMALAH system have been evaluated on one set of sequence data. The experimental results from the evaluation are described in this chapter. Some relevant background information not given in previous chapters will also be provided.

5.1 The Sequence Data

The set of sequence data used in the evaluation is derived from the January 1996 NCBI release of the UniGene set. It differs from the official NCBI release in that the SINEs and the LINEs have been masked out, courtesy of Greg Schuler of NCBI [Hudson 1996]†. This particular data set contains 45,252 clusters, 120,774 sequences, and 38,548,154 nucleotides. The clusters average 2.7 sequences per cluster; the largest cluster contains 1,127 sequences. Among them, 7,661 clusters contain at least 4 sequences, and only 4 clusters have more than 200 sequences. The average length of the sequences is 319 bases. (See Table B in Appendix E for a table summary.)

†SINEs and LINEs stand for short interspersed elements and long interspersed elements, respectively; in brief, they are repetitive elements that are found throughout the human genome [Watson 1992].

5.2 The Analysis

The following sections summarize the various analyses performed by the JAMALAH system for the evaluation. The rationale behind the choice of the default value for some important parameters is also offered.

5.2.1 Assembly of Sequences

Default Phrap parameters were used for sequence assembly. (See [Green 1995] for additional information.)

The 7,657 clusters that have between 4 and 200 sequences were subjected to sequence assembly analysis. Of the 7,657 clusters on which assembly was performed, only 77 clusters (approximately 1%) could not be assembled to form one single contig; 7,580 clusters were successfully assembled into single contigs. The 7,580 contigs have a total of 3,304,222 nucleotides, giving each of them an average length of 436 bases.

5.2.2 Identification of Potential Polymorphisms

Default SNP Detector parameters listed in Appendix A were used for the identification of potential single nucleotide polymorphisms in the assembled sequences. The reasoning behind the choices of some parameter values is now given.

Minimum Minor Allele Frequency

The default value for minimum minor allele frequency of is currently 0.2, or 20%; this is a loose but reasonable bound for detecting single nucleotide polymorphisms, for several reasons. For one individual who is polymorphic for a particular nucleotide, he or she always exhibits the two allele equally, since an individual always inherits one allele from the father and the other

from the mother. But this not the case when a group of random individuals are screened. The reason is simply that some individuals in the pool are not polymorphic for the particular locus, even though a significant population of people exhibit polymorphism for that locus.

The attempt to identify potential nucleotide polymorphisms by analyzing the large collection of DNA sequences from NCBI is almost analogous to comparative DNA sequencing of random DNAs from multiple unrelated individuals. Simply, the NCBI UniGene collection is dominated by a large number of EST sequences. In Release 94 of the UniGene collection, the most recent release, 237,476 of the 245,320 sequences subjected to clustering analysis are ESTs; in other words, an overwhelming 96.8% of the sequences in the UniGene collection are ESTs. As was mentioned, most ESTs are sequenced from multiple individuals [Adam 1995]. In addition, there is little to suggest that other genes in the NCBI collection are from related individuals.

Based on the latest result from comparative DNA sequencing of 10 unrelated individuals performed by David Wang and colleagues under the direction of Eric Lander at the Whitehead Institute/MIT Center for Genome Research, single nucleotide polymorphisms occurs at a rate of approximately 1 per 900 bases, when the minor allele frequency cutoff is 0.3 [Wang 1996a; Hudson 1996]. The choice of 0.2 as the cutoff for the JAMALAH system is, therefore, a reasonable choice--albeit a safe one.

Minimum Number of Reads Per Allele

By default, a polymorphic nucleotide that satisfies the requirement for the minor allele frequency must also have 2 high-quality reads per allele. In other words, a locus tentatively identified to be polymorphic for G (guanine) and (A) must be confirmed by two sequence that exhibit a "G" and two sequences that exhibit an "A" for that locus. A choice of at least 2 for this parameter is probably necessary, if one considers the high error of the EST sequences. In fact, the choice of 2 is extremely safe. The probability that the same nucleotide can exhibit the same allele due to sequencing error, which occurs at a rate of 1 per 100 bases, is 0.0002, or 1 per 5000 bases.

Minimum Consensus Overlap

To generate a consensus sequence from the sequence assembly, multiple sequence reads, currently 2 reads, are required to confirm each nucleotide in the consensus sequence. The parameter for the number of reads required is called minimum consensus overlap. As described in Chapter 3, the nucleotide sequence of the consensus sequence is generated starting from the site of the identified potential SNP. The identity of each nucleotide in the consensus sequence is determined by considering the identities of the nucleotides in the aligned sequences at the corresponding nucleotide position. Whenever the overlapping level of the aligned sequenced drops below minimum consensus overlap, the generation of consensus sequence is terminated.

As previously mentioned, the default value for minimum consensus overlap is 2. Considering that the chance of two high quality reads showing the same nucleotide when they should be different in actuality is quite low, less than 0.001, the choice is extremely safe, especially if one consider the fact that the algorithm currently generates an 'N' for nucleotide positions that exhibit

ambiguity and only uses the "majority wins" rule when the occurrence of the major allele is at least 3 times greater than the occurrence of the second most frequently occurring allele. This conservative approach should almost completely avoid any chance that a consensus sequence nucleotide is called incorrectly, resulting in subsequent incorrect selection of PCR primers.

Minimum Primer Length

The default value for this parameter is currently 20 bases, the typical length for a PCR primer.

Minimum Margin Downstream and Minimum Margin Upstream

The default values are 10. These 10-base windows are necessary to produce readable sequencing gels.

Minimum Consensus Sequence Length

The default value for this parameter is currently 80. Considering the fact that the default size of a primer is 20, and that the minimum margin from the target region to the end of the primer is 10 by default, a choice of at least 70 would be necessary for this parameter.

A total number of 7,580 clusters were subjected to SNP identification analysis. 5,046 potentially polymorphic loci were identified to be present among 786 clusters. Assuming the contig sequences for these 786 clusters also have an average length of 436 bases, then a potential SNP is found at the rate of approximately 1 SNP per 69 base among the 786 contig sequences. Since 7,580 clusters were subjected to SNP detection analysis, the rate of screening for SNPs is 1 per 655 bases. (See Table C in Appendix E for a summary of the numbers reported here.) The frequency of SNPs reported here is within the range of 1 SNP per approximately 200 bases to 1 SNP per 1000 bases previously reported by others [Kimura 1983; Wang 1996a]. However, clustering of identified SNPs is observed in many of the consensus sequences, suggesting that many of the identified polymorphisms might be false.

5.2.3 Generation of Consensus Sequence and Primer Picking

Consensus sequences usable for primer picking could only be generated for 338 clusters using the default setting. Primers were successfully selected for 272 consensus sequences, using Primer's default setting .

5.3 The Confirmation Process and the Result

To evaluate the accuracy of the system and to confirm identified single nucleotide markers, comparative DNA sequencing was performed. Sixteen consensus sequences generated by JAMALAH were randomly chosen from the "not-terrible" ones and uniformly chosen from various parts of the output generated by JAMALAH; this was done to avoid any potential bias in the evaluation process. These "not-terrible" consensus sequences consist of the ones that do not exhibit clustering of potential SNPs. The sixteen consensus sequences chosen range in size from 89 bases to 582 bases, and average 187 bases in length.

All sixteen sequences were subjected to primer picking. Primer pairs were only picked for 9 out of the 16 sequences. In general, the sequences that failed primer picking are shorter than the other nine sequences, which, except for two, are all longer than 187 bases. Among the seven that failed, none is longer than 160 bases.

The selected primers were used to amplify the genomic DNA of three unrelated human individuals (an Amish, a Venezuelan, and an American from Utah), and

a pool of genomic DNA from ten unrelated individuals; four PCR reactions were carried out for each primer pair selected. Seven out of the 9 primer pairs (78%) were successful in amplification. This success rate is comparable to the success rate one gets if the primers are picked from previously sequenced DNA sequences (approximately 80%) [Wang 1996b], and strongly suggests that the algorithm for generating consensus sequence is functional.

Of the seven sequences that were successfully amplified, only 6 had readable sequences. Among these six that had readable sequences, 3 sequences confirmed the single nucleotide polymorphisms identified by JAMALAH. One of the three sequences showed one SNP not identified by the JAMALAH system. (Figure 9 on next page shows actual sequence traces for one of the confirmed single nucleotide polymorphisms originally identified by JAMALAH.) In summary, the rate of SNP confirmation for the JAMALAH system is 3 out of 6, or 50%. (See Table D in Appendix E for a summary of these numbers.)

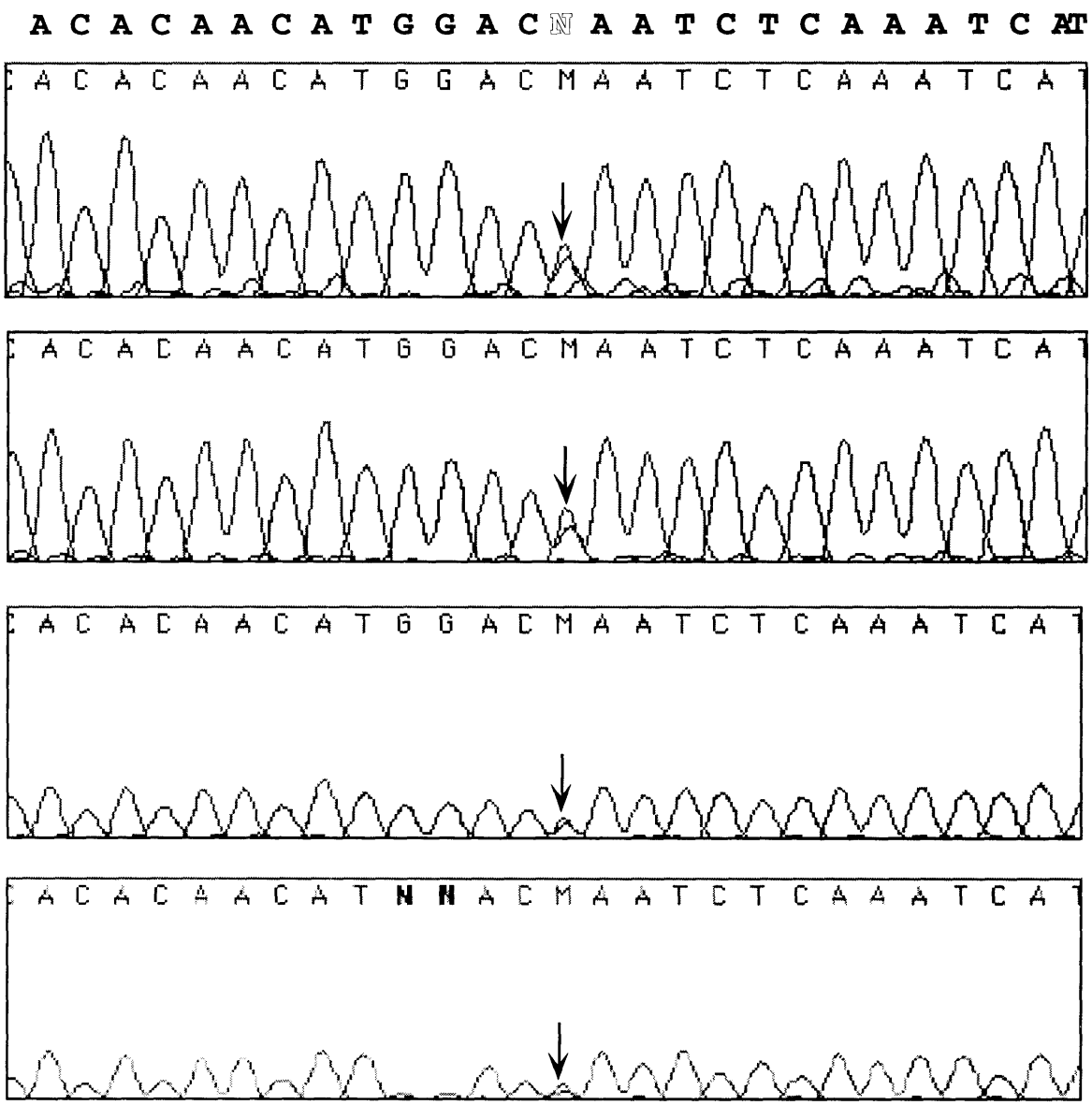

Figure 9: Actual sequence traces from comparative DNA sequencing confirmation of one of the single nucleotide polymorphisms identified by the JAMALAH system. Genomic DNA from unrelated individuals were amplified by polymerase chain reaction and sequenced using dye primer cycle sequencing technique. The top three traces belong to the three unrelated individuals--an Amish, an American from Utah, and a Venezuelan. The bottom one belongs to a pool of ten unrelated people. The arrows point to the polymorphic nucleotide. This particular nucleotide is polymorphic for A (adenine) and C (cytosine).

5.3 Interpretation of the Result

Based on the limited preliminary data, the rate of isolating single nucleotide polymorphisms using the JAMALAH system represents a two-fold improvement over isolation using STS (sequenced tag set) map [Wang, 199b]. However, this number is misleading, because only a small subset of the potential SNPs identified by JAMALAH were chosen to be sequenced. In addition, the way the confirmation experiment was performed could also have affected the rate of SNP confirmation. Perhaps, the individuals sequenced represent a portion of the population that does not exhibit polymorphism for the three loci that were not confirmed. Perhaps, there is a statistically significant number of people who do exhibit polymorphism for these loci. The inclusion of the genomic DNA pool, however, should have reduced the chance of such a scenario. More testing needs to be performed to determine the system's true performance.

5.3 Major Problem Identified

From careful analysis of the sequence assemblies generated by the JAMALAH system, one major problem with the current prototype has been identified. Specifically, the problem is the JAMALAH system's inability to recognize variations in nucleotides that arise due to alternative splicing of certain regions of certain sequences (see Figure 11 in Chapter 6, *Strengths and Limitations of the JAMALAH System*, for an illustration of alternatively spliced regions). In not being able to recognize these regions, JAMALAH falsely identifies a great number of nucleotides in these regions as being potentially polymorphic. One

would suspect that if these highly variable regions exist, then the sequences should never have been clustered together in the first place or would be able to be assembled into one single contig by the sequence assembler. However, this is not true. If two sequences are highly similar over a long stretch of nucleotides, great differences in the nucleotide sequence of a much shorter region would have no effect on the alignment. This is the natural behavior of dynamic programming algorithms for aligning sequences. However, the false identification of polymorphisms due to alternative splicing is one problem that can be remedied. Potential solutions are offered in the next chapter.

Chapter 6
Strengths and Limitations of the JAMALAH System

This chapter describes the strengths and limitations of the current JAMALAH system, and possible ways to remedy some of the problems discussed. Comparison of the JAMALAH system to a similar system being developed by a team at the Cooperative Human Linkage Center (CHLC) will also be given.

6.1 Strengths of the System

The development process and the evaluation of the system described in Chapter 5 have revealed the following strengths of the JAMALAH system:

- System is relatively easy to use.
- Easy to test and debug.
- When a high quality consensus sequence is generated by the SNP Detector, confirmation rate of actual polymorphism is relatively high.

6.1.1 System Easy to Use

Once the system is started on one of the UNIX workstations, no human intervention is necessary until the completion of the analysis. A typical session with the JAMALAH system usually involves typing a simple line of command to set up the various components in the pipeline. The analysis of over 45,000 UniGene sequence clusters is usually completed with 14-16 hours on a Sun SPARCstation 20 with 80 megabytes of RAM running SunOS version 4.1.3.

6.1.2 System Easy to Test and Debug

The modular structure of the system and its use of text-based format for data communication among the components in the pipeline makes the system easy to test. Since each component in the JAMALAH system is essentially an input-output filter, it is easy to test each individual component or a group of components by first detaching them from the rest of the pipeline. The well-defined format accepted and generated by each filter makes it extremely easy to write short simple Perl programs that parse the data generated by the filters to their human-readable forms. The use of the Boulder IO data format is particularly helpful, because a complete library of functions for manipulating the format is available. In addition, because both the Phrap-to-SNPD Translator and the SNP Detector are capable of generating detailed analysis and log files, it is easy to analyze the sequence assemblies for debugging purpose.

6.1.3 High Confirmation Rate for High Quality Consensus Sequences

When the sequences can be perfectly aligned and their alignment results in nearly perfect matches in the nucleotides of the sequences (as demonstrated in Figure 10 by a real example from the SNPD analysis file), the confirmation rate of identified single nucleotide polymorphisms is relatively high. By preliminary results presented in Chapter 5, *An Evaluation of the System*, the rate of confirmation is 50%. From careful studies of the analysis files generated by SNPD, such high quality alignments result in the generation of high quality consensus sequences that contain usually one or two ambiguous bases per approximately 170 bases. This suggest that one might be able to identify many real single nucleotide polymorphisms efficiently by performing confirmation experiments on these high quality consensus sequences.

```
GAAGGAACAAACCACTGAATCACACAACATGGAC[C]AatctcaAATCATTATGCTGATGGAAAGAA
GAAGGAACAAACcactgantcacacAACATGGAC[A]AATCTCAAATCATTATGCTGATGGAAAGAA
GAAGGAACAAACCACTGAATCACACAACATGGAC[C]AatctcaAATCATTATGCTGATGGAAAGAA
GAAGGAACAAACCACTGAATCACACAACATGGAC[A]AATCTCAAATCATTATGCTGATGGAAAGAA
GAAGGAACAAACCACTGAATCACACAACATGGAC[A]AATCTCAAATCATTATGCTGATGGAAAGAA
GAAGGAACAAACCACTGAATCACACAACATGGAC[C]AatctcaaagcattatGCTGATGGAAAGAA
GAAGGAACAAACCACTgaatcacacaa-------[-]-----------------------------
GAAGGAACAAACCACTGAATCACACAACATGGAC[A]AATCTCAAATCATTATGCTGATGGAAAGAA
GAAGGAACAAACCACTGAATCACACAACATGGAC[A]AATCTCAAATCATTATGCTGATGGAAAGAA
GAAGGAACAAACCACTGAATCACACAACATGGAC[A]AATCTCAAATCATTATgctgat--------
```

Figure 10: A high quality sequence alignment

6.2 Limitations of the System

The following weaknesses of the JAMALAH systems were revealed during the evaluation.

- Many probable non-polymorphisms are falsely identified.
- Inconsistent inter-component data format

6.2.1 False Identification of Non-Polymorphisms

A major cause of false identifications of potential single nucleotide polymorphisms by JAMALAH is its inability to accurately detect probable alternatively spliced regions or vector arms on the sequences. The presence of alternatively spliced regions in the sequences causes a series of adjacent nucleotide mismatches, which manifest themselves in the consensus sequences as a series of no-calls (see Figure 11). In many cases, these mismatches are identified as potential single nucleotide polymorphisms by the system.

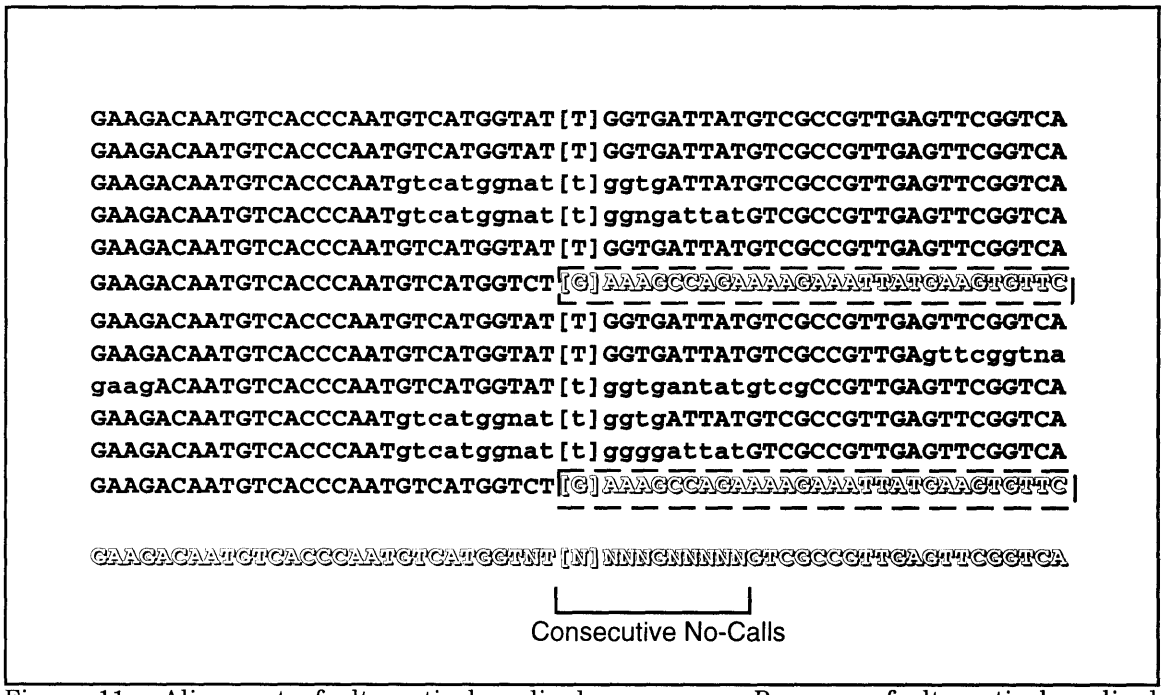

Figure 11: Alignment of alternatively spliced sequences. Presence of alternatively spliced regions results in many cases of false identification of potential SNPs. These false polymorphisms can be easily recognized by examining the consensus sequence, as they often appear as clusters of no-calls in the consensus sequence.

A preliminary analysis of the data from the evaluation described in Chapter 5 reveals that 3,701 of the 5,046 potential polymorphic loci identified by JAMALAH are identified from the sequences of only 86 clusters, suggesting

that an overwhelming majority of the identified potential polymorphisms are probably not real. Careful examination of the sequence alignment for these 86 clusters strongly suggests that many of the false polymorphisms are detected due to alternative splicing. There is clearly a need to account for mismatches in the nucleotides due to alternative splicing, if one wishes to reduce the amount of human intervention in the SNP screening process.

Several approaches can be taken to handle alternatively spliced subsequences. The first approach involves inserting an additional filter before the SNP Detector. The job of this component is to perform a preliminary alignment of the sequences, and to recognize probable variations due to alternative splicing, and remove the region from all sequences that overlap it. This filter would, however, have to duplicate many of the tasks the SNP Detector already performs, including the aforementioned alignment of sequences using Phrap-generated information and the analysis of nucleotide composition at each base position of the aligned sequences.

Then, the second approach is clearly to implement the recognition of the alternatively spliced subsequences within the SNP Detector itself. This would avoid the duplication of functionality in the SNP detection pipeline. However, this approach goes against one of the original design objectives for the system, specifically, to make the system as modular as possible. This approach makes an already relatively large piece of code even more difficult to modify and maintain.

The final approach is to attempt to recognize these alternatively spliced subregions by analysis of the consensus sequence alone. As has been observed

from the analysis generated by the SNP Detector, false nucleotide polymorphisms that arise due to alternative splicing manifest themselves in high numbers in very localized subregions of the sequences. These subregions appear in the consensus sequences as subsequences that exhibit a very high number of no-calls (N's), since a 'N' is generated for every identified, potentially polymorphic nucleotide. Based on the fact that single nucleotide polymorphisms appear approximately 1 every 1000 bases and the assumption that their occurrence is relatively uniform, it is possible to write a dynamic programming function to determine where these alternatively spliced regions begin and end, and thereby, remove them from the consensus sequence. Similarly, the multiple sequence alignment can be examined to identify these regions, as the sequence of all consensus sequences is determined directly from those of the aligned sequences. Steps to handle alternatively spliced subregions using this approach have been taken, but much work remains to be done.

6.2.2 Inconsistent Inter-Component Data Format

In the future, the Boulder IO format should also be adapted for the communication between the Phrap-to-SNPD Translator and the SNP Detector. The Boulder IO format was not used for communication of data between these two components, because when the SNP Detector was being implemented, the Boulder IO format was unknown to this author.

6.3 Demigloss / The CHLC System

Demigloss is an application currently being developed by Kenneth Buetow and colleagues at the Cooperative Human Linkage Center (CHLC) [Buetow 1996].

It is part of a larger system designed to identify potential single nucleotide polymorphisms from sequence assemblies. The system takes sets of clustered sequences from the National Center for Biotechnology Information (NCBI) UniGene collection, base-calls them if ABI sequencer trace data are available, determines if primers exist for the set, determines if the set has sufficient redundancy, assembles them, identifies potential polymorphisms, and determines if the candidate is flanked by existing primers.

The analytical core of the system is currently composed of three components: Phred for base-calling of sequence traces, Phrap for assembly of sequences, and Demigloss for identification of potential polymorphisms from sequence assemblies. Demigloss uses user-specified and derived nucleotide quality measures to discriminate regions likely to contain variants from those where differences are likely due to low sequence quality.

This CHLC system is of particular interest, because it shares several similarities with the JAMALAH system. They are similar in that both systems utilize publicly available, proven software (specifically Phrap) to perform the assembly of sequences, use a dedicated program for identifying polymorphisms from sequence assemblies, and allow the user to specify the parameters that are used to discern potential polymorphic nucleotides from false candidates. They, however, differ in the following ways. Unlike the CHLC system, JAMALAH currently does not generate quality data from ABI traces for any of the sequences, nor does JAMALAH identify only polymorphic sites flanked by published STS (sequenced tag set) primers. In addition, by default, JAMALAH currently analyzes all NCBI UniGene clusters having four or more sequences, while the CHLC system only checks ones containing ten or more

individual sequences. Therefore, JAMALAH's approach should allow it to identify more potential single nucleotide polymorphisms, since JAMALAH analyzes more clusters. Once JAMALAH's ability to detect alternatively spliced regions is improved, it will also be able to do so with high accuracy.

Information on the Demigloss system has not been published; therefore, very little is known about the other aspects of the Demigloss system, particularly with respect to the algorithm it uses to identify potential nucleotide polymorphisms from sequence assemblies. But what is known is that the Demigloss system use sequencing trace data to generate sequence quality scores, which are then used by Phrap to achieve high quality assembly of the sequences. While this may improve the quality of the alignment, it has the penalty of hindering throughput.

Chapter 7
Conclusions and Future Directions

7.1 Future Directions

This section outlines several directions in which the JAMALAH system can be expanded.

7.1.1 Evaluate Potential Benefits of Base-Calling

As discussed in Chapter 6, *Strengths and Limitations of the JAMALAH System*, performing base-calling on available sequencing trace data will definitely add dramatically to the time required for JAMALAH to completely analyze a large dataset such as the NCBI UniGene set. This bottleneck exists due to the current need to retrieve trace data from remote sites. If, however, the sequence assembler can generate many more high quality sequence assemblies, which might result in more accurate identification of potential SNPs, generation of longer consensus sequences that can be used for primer picking, etc., then it might be worthwhile to implement automated base-calling capability for the JAMALAH system. To study whether or not the need for this capability is warranted, quality files can be generated for a small subset of

the UniGene set, either manually or automatically by writing simple programs. This small subset of the UniGene can then be analyzed to determine the feasibility of automated base-calling.

7.1.2 Improve Recognition of Alternatively Spliced Regions

Currently, JAMALAH's ability to screen out alternatively spliced sequence regions is not robust, as discussed in Chapter 6. It needs to be improved.

7.1.3 Graphic Sequence and Trace Viewing Capability

To facilitate the confirmation of identified, potentially polymorphic nucleotides, it is desirable to have a program that presents in graphics the aligned sequences and their corresponding traces (if the traces are available). The adaptation of existing, publicly available software to achieve this goal is a sensible approach. Some publicly available software includes Ace.mbly, a graphic interactive program which allows the assembly and editing of DNA sequences generated by fluorescence sequencing machines [Thierry-Mieg 1995], and TED (Trace EDitor), a program that allows user to view and edit traces [Gleeson 1991].

7.2 Conclusions

This thesis has presented the design, implementation, and evaluation of the JAMALAH system, a prototype software system for detecting single nucleotide polymorphisms by sequence analysis. The system has been used to analyze a set of sequence data from the NCBI UniGene collection.

Comparative DNA sequencing has been conducted to confirm some of the potential polymorphisms identified by the system, and the accuracy of the polymorphism detection by the system has been evaluated. The results have shown the JAMALAH system to be capable of accurate identification of potential single nucleotide polymorphisms by sequence analysis.

The JAMALAH system, of course, is not a perfect system yet. Much work remains to be done to make it as powerful and useful as possible. The system still needs to be able to recognize alternatively spliced regions on sequences with more precision. Ideally, it also needs to provide a method for the user to review the analysis in a graphical, interactive and information-rich way. Some of these needs will probably be addressed soon.

Acknowledgments

Many thanks to:

Eric Lander, my thesis sponsor, for giving me the opportunity to pursue the work I have presented in this thesis. Professor Lander was one of the lecturers for the introductory biology class I took three years ago. If it were not for the enthusiasm and expertise he brought with him to every lecture, I would probably not have pursued a study in biology.

David Wang and Tom Hudson, my project advisors, for contributing their considerable expertise to this thesis, and for their patience with me. Their perspective and critiques added to what this thesis project has become.

Lincoln Stein, my "long-time" advisor, for his invaluable comments during the initial period of designing the JAMALAH system, and for always being an indispensable source of ideas regarding just about everything. His help with the drafting and proofreading of this paper also cannot be measured.

Helen Skaletsky and Steve Rozen for providing me with Primer 3 and answering my numerous questions about it.

John Rioux for the critical, insightful comments he gave during the early stage of developing JAMALAH.

Lois Bennett for putting up with my endless requests for disk space. Her help in various little ways cannot go unrecognized.

My dear friends outsides the confines of the Genome Center, specifically **J**ohn Yu, **A**udrey Kuang, **M**ichelle Hsu, **A**nn Chen, **L**eigh Lien, **A**nd **H**ovig Chitilian---whose priceless friendship inspired the name JAMALAH, for being whom they are. Their encouragement, guidance, and help have truly made my MIT experience a wonderful and crazy one.

Jim Clemens, one of my fellow M.Eng. students, for his friendship and his constant support the past few years.

Michael Rappa, my freshman advisor and friend, for standing by me and believing in me during a very difficult time in my life.

My mother and father, and the rest of my family for their their love and confidence in me throughout the years.

The Almighty God for His Grace and Love.

References

[Adams 1991] Adams MD, Kelley JM, Gocayne JD, Dubnick M, Polymeropoulos MH, Xiao H, Merril CR, Wu A, Olde B, Moreno RF, and Others. (1991). Complementary DNA sequencing: expressed sequence tags and human genome project. Science 252: 1651-1656.

[Adams 1995] Adams MD, Kerlavage AR, Fleischmann RD, Fuldner RA, Bult CJ, Lee NH, Kirkness EF, Weinstock KG, Gocayne JD, White O, Sutton G, Blake JA, Brandon RC, Chiu M-W, Clayton RA, Cline RT, Cotton MD, Earle-Hughes J, Fine LD, FitzGerald LM, FitzHugh WM, Fritchman JL, Geoghagen NSM, Glodek A, Gnehm CL, Hanna MC, Hedblom E, Hinkle Jr PS, Kelley JM, Klimek KM, Kelley JC, Liu L-I, Marmaros SM, Merrick JM, Moreno-Palanques RF, McDonald LA, Nguyen DT, Pellegrino SM, Phillips CA, Ryder SE, Scott JL, Saudek DM, Shirley R, Small KV, Spriggs TA, Utterback TR, Weidman JF, Li Y, Barthlow R, Bednarik DP, Cao L, Cepeda MA, Coleman TA, Collins E-J, Dimke D, Feng P, Ferrie A, Fischer C, Hastings GA, He W-W, Hu J-S, Huddleston KA, Greene JM, Gruber J, Hudson P, Kim A, Kozak DL, Kunsch C, Ji H, Li H, Meissner PS, Olsen H, Raymond L, Wei Y-F, Wing J, Xu C, Yu G-L, Ruben SM, Dillon PJ, Fannon MR, Rosen CA, Haseltine WA, Fields C, Fraser CM, and Venter JC. Initial assessment of human gene diversity and expression patterns based upon 83 million nucleotides of cNDA sequence. Nature 377: 3-17.

[Boguski 1995] Boguski MS, and Schuler GD. (1995). ESTablishing a human transcript map. Nature Genetics 10: 369-371.

[Botstein 1980] Botstein D, White RL, Skolnick M, and Davis RW. (1980). Construction of a genetic linkage map in man using restriction fragment length polymorphisms. Am. J. Hum. Genet. 32: 314-331.

[Boulder IO 1996] Boulder IO World-Wide Web site. *URL: http://www-genome.wi.mit.edu/genome_software/other/boulder.html*

[Buetow 1996] Buetow KH, Edmonson M, Zhang J, Rosen S, Levav E, and Manion F. (1996). Genetic Annotation of EST Sequence Data. The 9th Annual Genome Mapping and Sequencing Meeting at Cold Spring Harbor Laboratory.

[Collins 1990] Collins JF, and Coulson AFW. (1990). Significance of protein sequence similarities. Methods in Enzymology 183: 474-486.

[Collins 1995] Guyer MS, and Collins FS. (1995). How is the Human Genome Project doing, and what have we learned so far? Proc. Natl. Acad. Sci. USA 92: 10841-10848.

[Cooper 1985] Cooper DN, Smith BA, Cooke HJ, Niemann S, and Schmidtke J. (1985). An estimate of unique DNA sequence heterozygosity in the human genome. Human Genetics 69: 201-205.

[Cormen 1994] Cormen TH, Leiserson CE, and Rivest RL. (1994). *Introduction to Algorithms.* The MIT Press, Cambridge, MA.

[Dib 1996] Dib C, Fauré S, Fizames C, Samson D, Drouot N, Vignal A, Millasseau P, Marc S, Hazan J, Seboun E, Lathrop M, Gyapay G, Morissette J, and Weissenbach J. (1996). A comprehensive genetic map of the human genome based on 5,264 microsatellites. Nature 380: 152-154.

[Dietrich 1992] Dietrich W, Katz H, Lincoln SE, Shin HS, Friedman J, Dracopoli NC, and Lander ES. (1992). A genetic map of the mouse suitable for typing intraspecific crosses. Genetics 131: 423-447.

[Dietrich 1996] Dietrich WF, Miller JC, Steen R, Merchant MA, Damron-Boles D, Husain Z, Dredge R, Daly MJ, Ingalls KA, O'Connor TJ, Evans CA, DeAngelis MM, Levinson DM, Kruglyak L, Goodman N, Copeland NG, Jenkins NA, Hawkins TL, Stein L, Page DC, and Lander ES. (1996). A comprehensive genetic map of the mouse genome. Nature 380: 149-152.

[Ginsburg 1994] Ginsburg M. (1994). Sequence Comparison. In *Guide to Human Genome Computing*, Martin J. Bishop, ed. Academic Press, London.

[Gleeson 1991] Gleeson T, and Hillier L. A trace display and editing program for data from fluorescence based sequencing machines. Nucleic Acid Research 19: 6481-6483.

[Green 1995] Green P. (1995). Document for Phrap. *Software and information available by sending email to phg@u.washington.edu.*

[Green 1996] Green P. (1996). Document for Phred. *Software and information available by sending email to phg@u.washington.edu.*

[Hudson 1995] Hudson TJ, Stein LD, Gerety SS, Ma J, Castle AB, Silva J, Slonim DK, Baptista R, Kruglyak L, Xu S-H, Hu X, Colbert AME, Rosenberg C, Reeve-Daly MP, Rozen S, Hui L, Wu X, Vestergaard C, Wilson KM, Bae JS, Maitra S, Ganiatsas S, Evans CA, DeAngelis MM, Ingalls KA, Nahf RW, Horton LT, Anderson MO, Collymore AJ, Ye W, Kouyoumjian V, Zemsteva IS, Tam J, Devine R, Courtney DF, Renaud MT, Nguyen H, O'Connor TJ, Fizames C, Fauré S, Gyapay G, Dib C, Morissette J, Orlin JB, Birren BW, Goodman N, Weissenbach J, Hawkins TL, Foote S, Page DC, and Lander ES. (1995). An STS-Based Map of the Human Genome. Science 270: 1945-1954.

[Hudson 1996] Hudson TJ. (1996). Personal communication.

[Kimura 1983] Kimura M. (1983). *The Neutral Theory of Molecular Evolution.* Cambridge University Press, Cambridge, page 238.

[Kruglyak 1995] Kruglyak L, and Lander, ES. (1995). Complete Multipoint Sib-Pair Analysis of Qualitative and Quantitative Traits. Am. J. Hum. Genet. 57: 439-454.

[Lipschutz 1995] Lipschutz RJ, Morris D, Chee M, Hubbell E, Kozal MJ, Shah N, Shen N, Yang R, and Fodor SPA. (1995). Using Oligonucleotide Probe Arrays To Access Genetic Diversity. BioTechniques 9: 442-447.

[Love 1990] Love JM, Knight AM, McAleer MA, and Todd JA. (1990). Towards construction of a high resolution map of the mouse genome using PCR-analyzed microsatellites. Nucleic Acids Research 18: 4123-4130.

[Marra 1996] Marra M, and Hillier L, the Washington University-Merck Human EST Sequencing Project, and the I.M.A.G.E. Consortium. (1996). 258,000 Human ESTs and Counting. The 9th Annual Genome Mapping and Sequencing Meeting at Cold Spring Harbor Laboratory.

[Miller 1994] Miller MJ, and Powell JI. (1994). A Quantitative Comparison of DNA Sequence Assembly Programs. Journal of Computational Biology 1: 257-269.

[Needleman 1970] Needleman S, and Wunsch C. (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.

[Nickerson 1992] Nickerson DA, Whitehurst C, Boysen C, Charmley P, Kaiser R, and Hood L. (1992). Identification of Clusters of Biallelic Polymorphic Sequence-Tagged Sites (pSTSs) That Generate Highly Informative and Automatable Markers for Genetic Linkage Mapping. Genomics 12: 377-387.

[Olson 1989] Olson ML, Hood L, Cantor C, and Botstein D. (1989). A common language for physical mapping of the human genome. Science 245: 1434-1435.

[Pearson 1992] Pearson WR, and Miller W. (1992). Dynamic Programming Algorithms for Biological Sequence Comparison. Methods in Enzymology 210: 575-601.

[Pease 1994] Pease AC, Solas D, Sullivan EJ, Cronin MT, Holmes CP, and Fodor SP. (1994). Light-generated oligonucleotide arrays for rapid DNA sequence analysis. Proc. Natl. Acad. Sci. USA 91: 5022-5026.

[PRIMER 1996] *Software and information available by sending email to primer2@genome.wi.mit.edu.*

[Schuler 1996] Schuler GD, Miller W, Myers EW, Lipman DJ, and Boguski MS. (1996). UniGene: A Unified Collection of Transcript Sequences for Placement on the Human Gene Map. The 9th Annual Genome Mapping and Sequencing Meeting at Cold Spring Harbor Laboratory.

[Smith 1981] Smith TF, and Waterman MS. (1981). Identification of Common Molecular Subsequences. Journal of Molecular Biology 147: 195-197.

[Stein 1994] Stein LD, Marquis A, Dredge RD, Reeve MP, Daly M, Rosen S, and Goodman N. (1994). Splicing UNIX into a Genome Mapping Laboratory. The Usenix Summer 1994 Conference.

[Stein 1995] Stein LD. Personal communication, 1995.

[Thierry-Mieg 1995] Thierry-Mieg D, Thierry-Mieg J, and Sauvage U. (1995). Ace.mbly: a graphic interactive program to support shotgun and directed sequencing projects. *Software and information available by sending anonymous FTP from the following World-Wide Web sites. URL1: ftp://ftp.crbm.cnrs-mop.fr/pub/acembly (France). URL2: ftp://ncbi.nlm.nih.gov/repository/acedb/acembly (USA).*

[UniGene 1996] UniGene World-Wide Web site. *URL: http://www.ncbi.nlm.nih.gov/Schuler/UniGene*

[Wall 1991] Wall L, and Schwartz RL. (1991). *Programming Perl.* O'Reilly & Associates, Sebastopol, CA.

[Wang 1995] Wang DG-W, Hawkins TL, Spencer SB, Siao C-J, Hudson TJ, and Lander, ES. Toward automated comparative DNA sequencing of PCR products for identification of sequence polymorphisms in the human genome. The 45th Annual Meeting of the American Society of Human Genetics, October 1995. Minneapolis, Minnesota.

[Wang 1996a] Wang D, Sapolsky R, Rioux J, Spencer J, Kruglyak L, Hubbell E, Ghandour G, Hawkins T, Hudson T, Lipschutz R, and Lander E. (1996). Toward a third generation genetic map of the human genome: Bi-allelic polymorphisms detected on high-density DNA probe arrays. The 9th Annual Genome Mapping and Sequencing Meeting at Cold Spring Harbor Laboratory.

[Wang 1996b] Wang DG-W. (1996). Personal communication.

[Weber 1989] Weber JL, and May PE. Abundant class of human DNA polymorphisms which can be typed using the polymerase chain reaction. Am. J. Hum. Genet. 57: 388-396.

[Weissenbach 1992] Weissenbach J, Gyapay G, Dib C, Vignal A, Morissette J, Millasseau P, Vaysseix G, and Lathrop M. (1992). A second-generation linkage map of the human genome. Nature 359: 794-801.

[Watson 1992] Watson JD, Gilman M, Witkowski J, and Zoller M. (1992). *Recombinant DNA*. Scientific American Books, New York.

Appendix A: Default SNP Detector Parameters

Table A: SNP Detector Parameters for Identification of SNP Polymorphisms

Parameter Name	Description	Default Value
use_high_qual_only	If TRUE, SNPD considers only high-quality nucleotides in analysis. Otherwise, all nucleotides are considered. If FALSE, all nucleotides are considered.	TRUE
min_minor_allele_freq	An allele whose allele frequency is at least this value (and satisfies the min_reads_per_allele requirement below) is considered a valid allele for that locus. Otherwise, it is not.	20 (%)
min_reads_per_allele	An allele that is confirmed by at least this many sequence reads (and satisfies the min_minor_allele freq requirement above) is considered a valid allele for that locus. Otherwise, it is not.	2
qual_window_size	Width of window to perform quality check (see qual_window_threshold). Actual width of the window is 2 * qual_window_size + 1 centered on nucleotide under consideration	5
qual_window_threshold	If the quality check window (see qual_window_size above and Figure 7) contains a greater percentage of "N" and "*" than this, do not consider nucleotide under consideration a potential SNP.	10
print_sequence_window	Print the aligned sequences surrounding the identified potential SNP. Window center on SNP.	TRUE
sequence_window_size	Size of sequence window to print. Actual width of window is 2 * qual_window_size + 1.	20
no_file_conflict	If FALSE, print info of all loci that show multiple alleles. If TRUE, do not print.	TRUE
no_file_polymorphism	If FALSE, print info of all loci that have identified to be potential polymorphic. If TRUE, do not print.	TRUE
no_file_analysis	If FALSE, print info of all loci. If TRUE, do not print.	TRUE
no_file_all	If TRUE, do not print the conflict file, the polymorphism file, or the analysis file.	TRUE

Table B: SNP Detector Parameters for Generation of Consensus Sequence

Parameter Name	Description	Default Value
min_consensus_overlap	Each nucleotide in the consensus sequence must be confirmed by this many reads.	2
min_consensus_length	An usable consensus sequence must be at least this long.	80
max_pcr_product_length	Maximum length of PCR product allowed for sequencing purposes.	400
min_primer_length	Minimum length of an ideal primer. An usable consensus sequence must have an unambiguous region of this length both upstream and downstream of the identified potential SNP.	20
min_margin_upstream	This many bases upstream of the identified potential SNP cannot be analyzed when checking min_primer_length. In other words, the forward primer cannot be picked within this region.	10
min_margin_downstream	his many bases downstream of the identified potential SNP cannot be analyzed when checking min_primer_length. In other words, the reverse primer cannot be picked within this region.	10

Appendix B: Data and File Formats

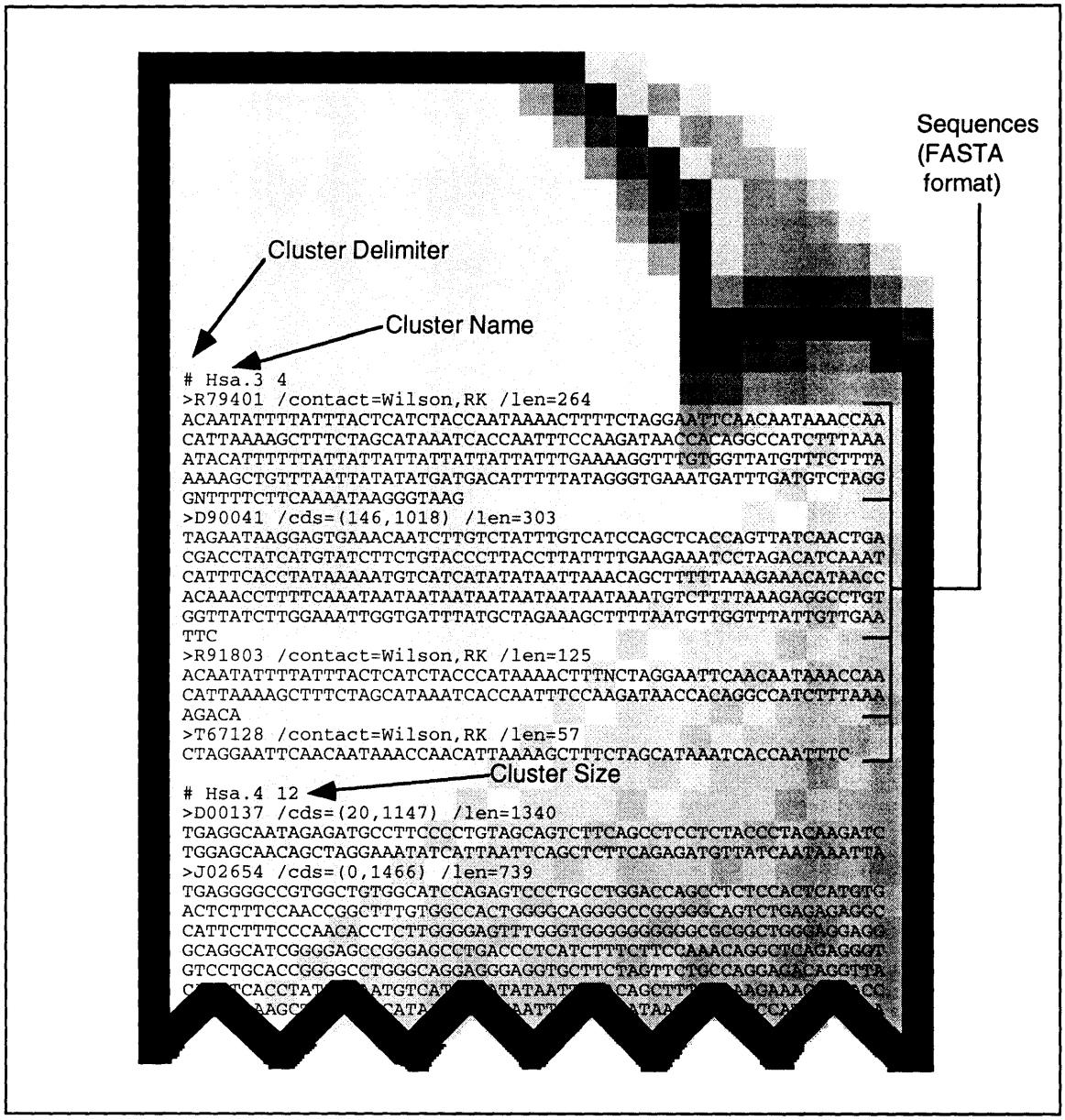

Figure A: Sample sequence data from the NCBI UniGene set. The sequences are segregated into individuals clusters. The end of a cluster and the beginning of the next cluster is indicated with a '#' character placed at the beginning of the line. Also on that line are the name of the next cluster and the number of sequences the cluster contains.

```
>R79401 /contact=Wilson,RK /len=264
ACAATATTTTATTTACTCATCTACCAATAAAACTTTTCTAGGAATTCAACAATAAACCAA
CATTAAAAGCTTTCTAGCATAAATCACCAATTTCCAAGATAACCACAGGCCATCTTTAAA
ATACATTTTTTATTATTATTATTATTATTTGAAAAGGTTTGTGGTTATGTTTCTTTA
AAAAGCTGTTTAATTATATATGATGACATTTTTATAGGGTGAAATGATTTGATGTCTAGG
GNTTTTCTTCAAAATAAGGGTAAG
>D90041 /cds=(146,1018) /len=303
TAGAATAAGGAGTGAAACAATCTTGTCTATTTGTCATCCAGCTCACCAGTTATCAACTGA
CGACCTATCATGTATCTTCTGTACCCTTACCTTATTTTGAAGAAATCCTAGACATCAAAT
CATTTCACCTATAAAAATGTCATCATATATAATTAAACAGCTTTTTAAAGAAACATAACC
ACAAACCTTTTCAAATAATAATAATAATAATAATAATAAATGTCTTTTAAAGAGGCCTGT
GGTTATCTTGGAAATTGGTGATTTATGCTAGAAAGCTTTTAATGTTGGTTTATTGTTGAA
TTC
>R91803 /contact=Wilson,RK /len=125
ACAATATTTTATTTACTCATCTACCCATAAAACTTTNCTAGGAATTCAACAATAAACCAA
CATTAAAAGCTTTCTAGCATAAATCACCAATTTCCAAGATAACCACAGGCCATCTTTAAA
AGACA
>T67128 /contact=Wilson,RK /len=57
CTAGGAATTCAACAATAAACCAACATTAAAAGCTTTCTAGCATAAATCACCAATTTC
```

Figure B: Phrap input file format. The UniGene cluster header is removed to obtain the FASTA format which is accepted by Phrap.

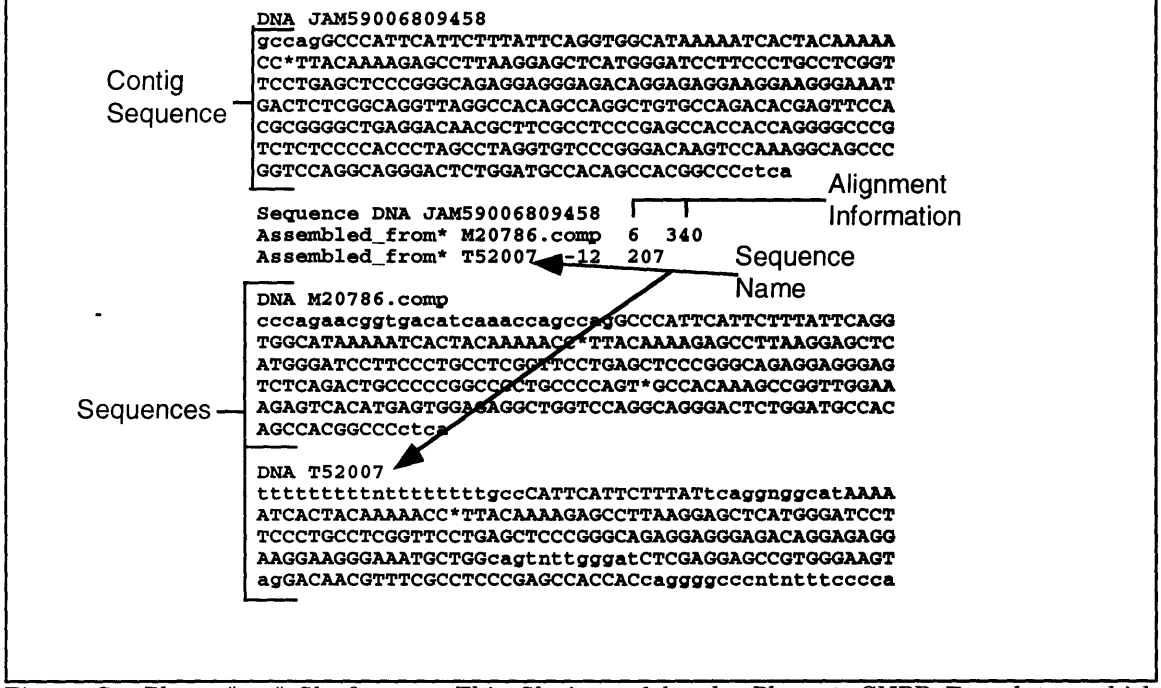

Figure C: Phrap "ace" file format. This file is read by the Phrap-to-SNPD Translator, which extracts the alignment information, the contig sequence, and the sequences to be aligned to the contig from the file. The left number in the alignment information indicates the contig position to which the left end of the specified sequence should be aligned. The number on the right indicates the contig position to which the right end should be aligned.

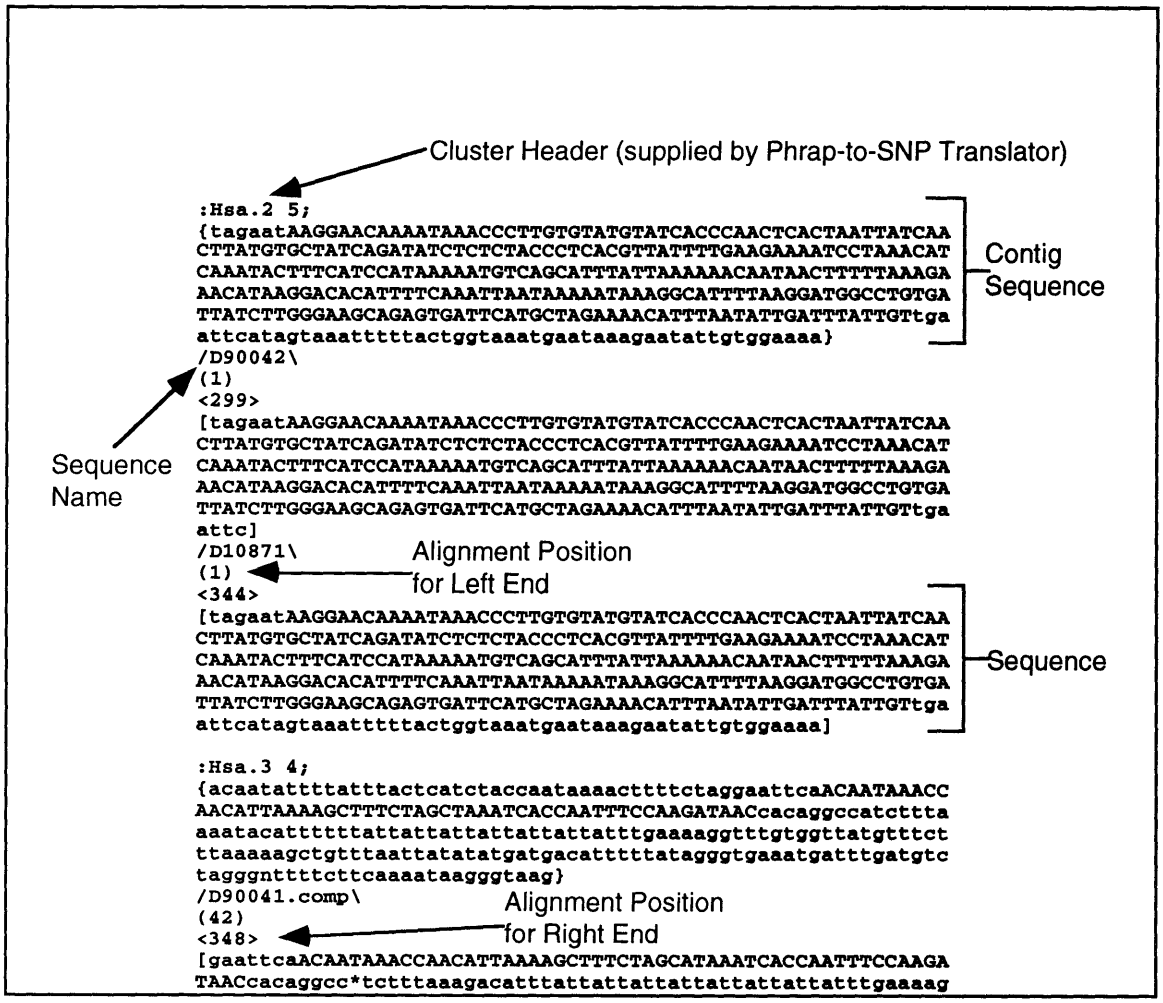

Figure D: The data format currently used by SNP Detector to receive alignment information and sequences from Phrap-to-SNPD Translator.

Appendix C: Sample SNPD and PRIMER Output, and Boulder IO

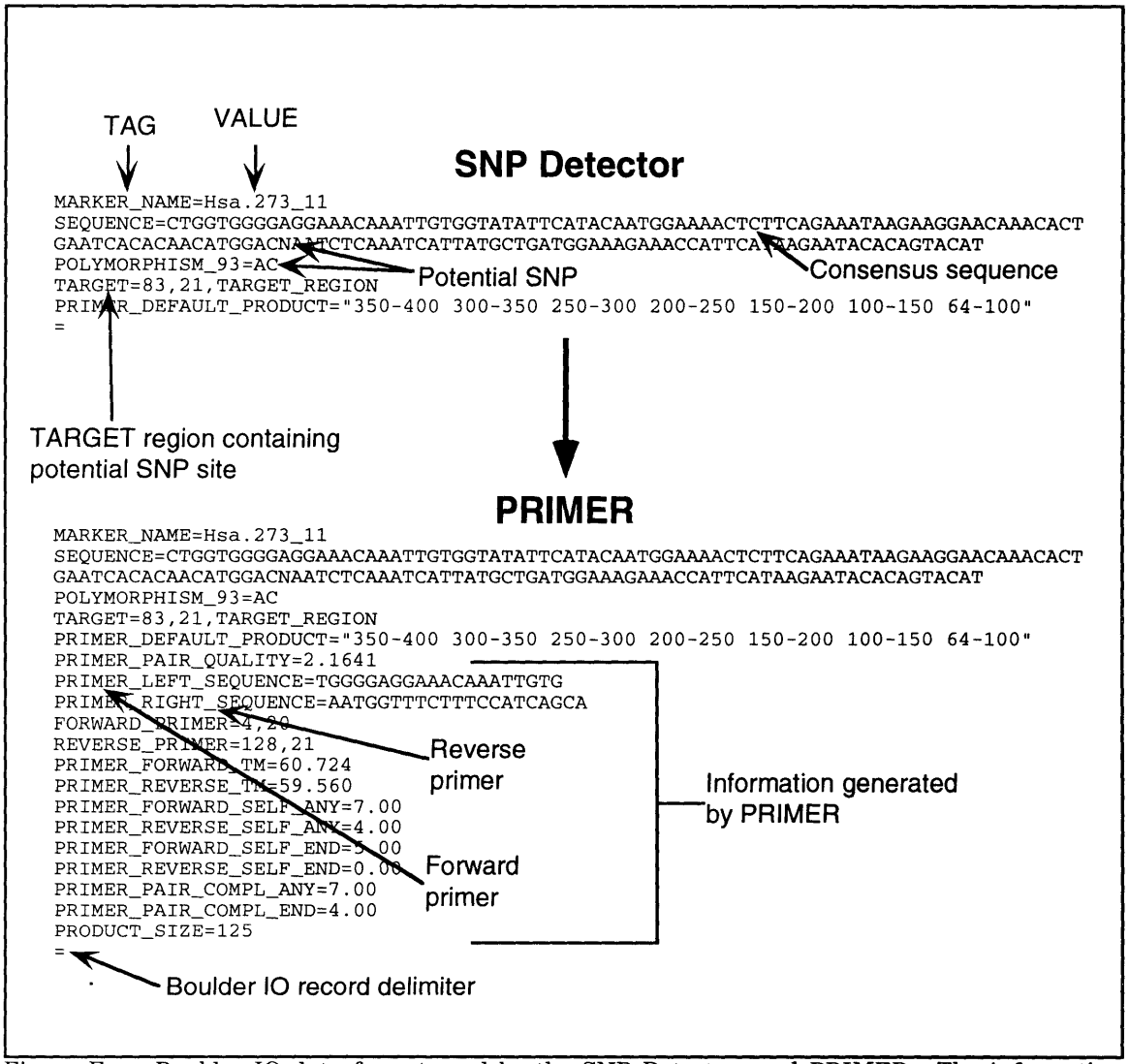

Figure E: Boulder IO data format used by the SNP Detector and PRIMER. The information generated by PRIMER is added to the information generated by the SNP Detector, before the collective information is passed onto the next component, the Primer Order and SNP Data Processor (not shown).

Appendix D: Snapshot of SNP Data Sheet and Primer Purchase Form

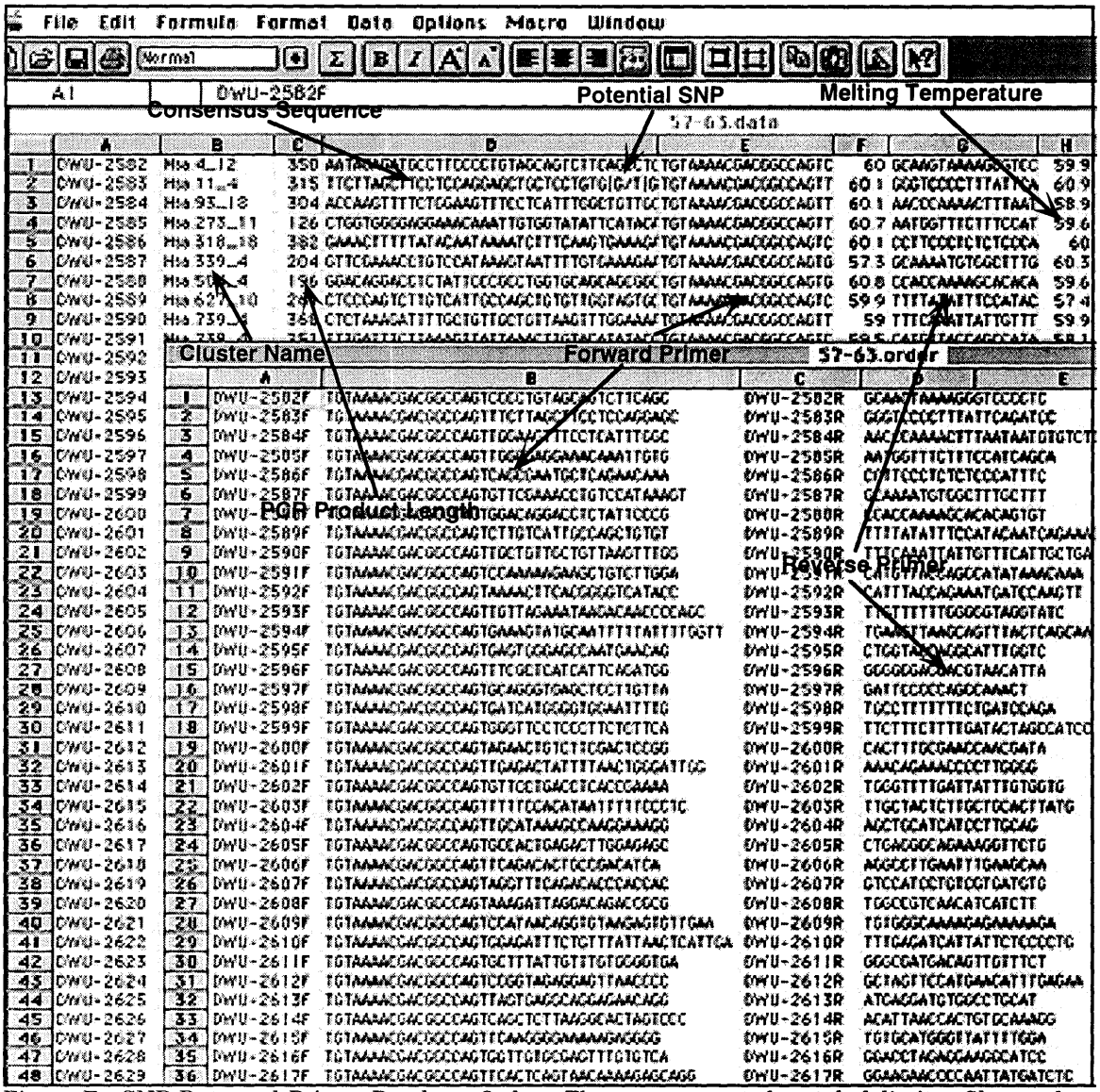

Figure F: SNP Data and Primer Purchase Order. They are generated as tab-delimiter files, and converted by BinHex and the user's Macintosh electronic mail client into MS Excel spreadsheets.

Appendix E: Summary of Data from Evaluation of the JAMALAH System

Table B: Summary of UniGene Sequence Data

Number of clusters	45,252
Number of sequences	120,774
Number of nucleotides	38,548,154 bases
Number of sequences in largest cluster	1,127
Number of clusters with 4 or more sequences	7,661
Number of clusters with 200 or more sequences	4
Average number of sequences per cluster	2.7
Average length of sequence	319 bases

Table C: Summary of Analysis by the JAMALAH System

Number of clusters analyzed	45,252
Number of clusters subjected to sequence assembly (clusters with 4-200 sequences)	7,657
Number of clusters that could be assembled into single contigs	7,580
Total number of nucleotides in contig sequences	3,304,222 bases
Average length of contig sequence	436 bases
Number of SNPs identified	5,046
Number of clusters with SNPs	786
Number of clusters with SNPs and usable consensus sequences	338
Number of clusters with SNPs, usable consensus sequences, and primer pairs picked using PRIMER default setting	272
Rate of SNP screening	1 / 655 bases

Table D: Summary of Results from Primer Picking, Sequencing, and Confirmation

Number of consensus sequences subjected to primer picking	16
Number of consensus sequences successful in primer picking	9
Rate of successful primer picking	56%
Number of successful amplifications	7
Rate of successful PCR amplification	78%
Number of readable sequences	6
Rate of readable sequences	86%
Number of identified potential SNPs confirmed	3
Number of SNPs confirmed	4
Rate of SNP confirmation	50%

Appendix F: Complete Program Listing

The Input Sequence File Parser and the Phrap-to-SNPD Translator / JamIt.pl

```
#!/usr/local/bin/perl
# -*-Mode: perl;-*-
push(@INC,'/usr/local/lib/perl');

# Print things out as they come.
$|=1;

# Some configuration variables.

# Smallest clusters to check.
$MINIMUM_SAMPLE_SIZE = 4;

# Largest clusters to check.
$MAXIMUM_NUMBER_INPUT_SEQ = 200;

# Command to invoke the sequence assembler to use.
$PHRAP_COMMAND_PREFIX = "/tmp_mnt/usr/local/users1/chchench/phrap";
$PHRAP_COMMAND_SUFFIX =  "-ace >/dev/null 2>&1";

# Name of log file to use for system "front end"
$LOG_FILENAME = "./JAMALAH.log";

# Prefix for the name of temporary files generated by sequence parser.
$FILENAME_PREFIX = "JAM";

# Initialize some variables.  If DEBUG_MODE is true, generated debugging
# information during execution.
$DEBUG_MODE = 0;
$DEBUG_OUT = STDOUT;

# If SHOW_PROGRESS is true, generate progress update to standard error.
$SHOW_PROGRESS = 1;

# Initialize some more variables.
$FILE_COUNTER = 0;
$NOT_USE_FILE = 1;
```

```
# Initialize some more variables.
$CLUSTER_SIZE = 0;
%CLUSTER = ();
%ASSEMBLY_INFO1 = ();
%ASSEMBLY_INFO2 = ();
$CONTIG_SEQ = "";
$CLUSTER_NAME = "";

# check commandline for options.  Currently supported features are:
# -debug:  if specified, print debugging info to DEBUG_OUT.
# -file:  if specified, directly use specified "ace" file.
# -log:  if specified, use specified log file.
for ($i=0; $i<@ARGV; $i++) {
    if ($ARGV[i] eq "-debug") {
       $DEBUG_MODE = true;
    } elsif ($ARGV[$i] eq "-file") {
       $NOT_USE_FILE = 0;
       $FILE_TO_USE = $ARGV[++$i];
       $FILE_TO_USE =~ s/.ace//;
       $CLUSTER_NAME = $FILE_TO_USE;
    } elsif ($ARGV[$i] eq "-log") {
       $LOG_FILENAME = $ARGV[++$i];
    }
}


# Main program; driver loop

# Open log file.  Write start time of analysis to log file.
open(LOG, ">>$LOG_FILENAME");
print LOG "\nTIME OF PROCESSING:  ";
$CURRENT_TIME = `date`;
print LOG $CURRENT_TIME;
if ($SHOW_PROGRESS) { print STDERR "\nTIME OF PROCESSING:  "; }
if ($SHOW_PROGRESS) { print STDERR $CURRENT_TIME; }

# Just use the specified "ace" file.  Parse and send alignment info to
# the sequence analyzer through standard output and quit.
if ($NOT_USE_FILE == 0) {
    open(LOG, ">>$LOG_FILENAME");
    &exec_alignment($FILE_TO_USE);
    print LOG "PROCESS COMPLETED\n\n";
    if ($SHOW_PROGRESS) { print STDERR "PROCESS COMPLETED\n\n"; }
    system("rm -f .*$FILENAME_PREFIX*");
    close(LOG);
    exit(0);
}

# Main loop for parsing cluster file.  Not very Perl-like code.
# Read UniGene cluster file through standard input.
while (<STDIN>) {
    # Look for lines with the cluster delimiter "#"
    if (/^#(.*)\s/) {
       # Extract rest of header from the line.
       $HEADER_INFO = substr($_, 2);
       chop($HEADER_INFO);

       if (close(FILEHANDLE)) {
           &print_cluster;

           if (!$DEBUG_MODE) {
              # Check to see if the cluster just read has two few or too
              # many s equences.  If not, call procedure to execute
```

```
            # sequence assembly.
            $NUMBER_OF_INPUT_SEQUENCES = &get_sequence_number($FILENAME);
            if ($NUMBER_OF_INPUT_SEQUENCES > $MAXIMUM_NUMBER_INPUT_SEQ) {
                print LOG "FILE <$FILENAME> contains too many sequences\n";
                if ($SHOW_PROGRESS) {
                   print STDERR "FILE <$FILENAME> contains too many sequences\n";
                }
            }
            if (($NUMBER_OF_INPUT_SEQUENCES >= $MINIMUM_SAMPLE_SIZE) &&
                ($NUMBER_OF_INPUT_SEQUENCES<= $MAXIMUM_NUMBER_INPUT_SEQ)) {
                &exec_alignment($FILENAME);
            }
            # Delete any temporary files not deleted elsewhere.
            system("rm -f .*$FILENAME_PREFIX*");
         }
         print LOG "PROCESS COMPLETED\n\n";
         if ($SHOW_PROGRESS) { print STDERR "PROCESS COMPLETED\n\n"; }
      }

      # Keep track of the number of clusters parsed.
      $FILE_COUNTER++;

      # Extract cluster name from cluster header.
      $CLUSTER_NAME = $HEADER_INFO;
      &extract_cluster_info($HEADER_INFO);

      # Generate some random number for naming the new temporary file.
      srand;
      # Attach the random number to the temporary file prefix.
      $FILENAME = ".$FILE_COUNTER" . $FILENAME_PREFIX . rand;
      # Open the temporary file for writing.
      open(FILEHANDLE,">$FILENAME");
   }
   else {
      # We don't find a cluster delimiter on this line, so we must be
      # looking at one of the lines for sequences.  Print the line to
      # the temporary file.
      print FILEHANDLE "$_";

      if (/^>(.*)\s/) {
         if ($DEBUG_MODE) {
            print $DEBUG_OUT "We found the first line of a sequence.\n";
         }
      }
   }
}

# Close the temporary file for the very last cluster and perform analysis
# on it.
close(FILEHANDLE);
&print_cluster;
if (!$DEBUG_MODE) {
   if (&get_sequence_number($FILENAME) > 1) {
      &exec_alignment($FILENAME);
   }
}

# Print completion information to the log file.  Finish things up.
print LOG "PROCESS COMPLETED\n\n";
if ($SHOW_PROGRESS) { print STDERR "PROCESS COMPLETED\n\n"; }
system("rm -f .*$FILENAME_PREFIX*");

print LOG "\n-------------------------------------------------\n";
```

```
if ($SHOW_PROGRESS) { print STDERR "\n-----------------------------------------------
--\n"; }
close(LOG);


# Subroutines start here.

# Subroutine for extracting cluster info.  Not really useful right now.
sub extract_cluster_info {
    print LOG "\nPROCESSING <$_[0]>...\n";
    if ($SHOW_PROGRESS) {
       print STDERR "\nPROCESSING <$_[0]>...\n";
    }
}

# Subroutine for debugging purposes.
sub print_cluster {
    if ($DEBUG_MODE) {
       print $DEBUG_OUT "========== $FILENAME ==========\n";
       foreach $key (keys %CLUSTER) {
           print $DEBUG_OUT "***** $key *****\n";
           print $DEBUG_OUT "$CLUSTER{$key}\n";
       }
    }
}

# Subroutine that invokes the sequence assembler and checks to ensure
# that the result of the assembly is one single contig.  It then sends
# the contig sequences, the cluster sequences, and their alignment info
# out through standard out.
sub exec_alignment {
    local($NUMBER_OF_CONTIGS);

    if ($NOT_USE_FILE == 1) {
       # We are use "ace" file so we have to call sequence assembler Phrap.
       &call_phrap($_[0]);

       # Determine number of contigs generated by assembly analysis.
       $NUMBER_OF_CONTIGS = &get_sequence_number("$_[0].contigs");

       # If no contig could be generated, print error message.
       if ($NUMBER_OF_CONTIGS <= 0) {
           print LOG "NO CONTIG of <$_[0]> could be assembled\n";
           if ($SHOW_PROGRESS) {
              print STDERR "NO CONTIG of <$_[0]> could be assembled\n";
           }
           return;
       }
       else {
           # If more than 1 contig could be generated, also print error msg.
           if ($NUMBER_OF_CONTIGS > 1) {
              print LOG "MORE THAN 1 CONTIG of <$_[0]> could be assembled\n";
              print LOG "\tIGNORE THIS CLUSTER\n";
              if ($SHOW_PROGRESS) {
                  print STDERR "MORE THAN 1 CONTIG of <$_[0]> could be assembled\n";
                  print STDERR "\tIGNORE THIS CLUSTER\n";
              }
              return;
           }
       }

       # Delete unnecessary Phrap output files now.
       system("rm -f $_[0].contigs");
    }
```

```
# Initialize variables before parsing Phrap "ace" file.
$CONTIG_SEQ = "";
$CLUSTER_SIZE = 0;
%CLUSTER = ();
%ASSEMBLY_INFO1 = ();
%ASSEMBLY_INFO2 = ();

# Open Phrap "ace" file.
open(ACE_FILE, "$_[0].ace") || die "cannot open file <$_[0].ace>\n";
local($READING_MODE);
local($FRAGMENT_NAME);

# Read contig sequence into CONTIG_SEQ.
# Read each sequence and add it to the associative array CLUSTER.  Use
#   GenBank accession name for the sequence as index to the array.
while (<ACE_FILE>) {
   if (/^DNA Contig(.*)\s/) {
       # We've found the header for the contig sequence.
       $READING_MODE = 1;
       print LOG "READING CONTIG SEQUENCE from file <$_[0].ace>\n";

   } elsif (($READING_MODE == 1) && (/^\s/)) {
       # We are done reading the contig sequence, so now it's turn
       # to read the cluster sequences or their complements used in
       # assembly.
       $READING_MODE = 0;
       print LOG "READING PADDED EST sequences from file <$_[0].ace>: ";
       if ($SHOW_PROGRESS) {
          print STDERR "READING PADDED EST sequences from file <$_[0].ace>: ";
       }

   } elsif (/^DNA (.*)\s/) {
       # We've found the header for a sequence.  Increment the counter.
       s/DNA //;
       chop;
       $FRAGMENT_NAME = $_;
       $READING_MODE = 2;
       $CLUSTER_SIZE++;
       print LOG " $CLUSTER_SIZE";
       if ($SHOW_PROGRESS) { print STDERR " $CLUSTER_SIZE"; }

   } elsif (($READING_MODE == 2) && (/^\s/)) {
       $READING_MODE = 0;
   } elsif (/^Sequence Contig(.*)\s/) {
       $READING_MODE = 3;
   } elsif (($READING_MODE == 3) && (/^\s/)) {
       $READING_MODE = 0;

   } elsif (/^Base_segment.*/) {
       break;

   } else {
       if ($READING_MODE == 1) {
          # Reading in contig sequence.
          chop;
          $CONTIG_SEQ .= $_;
       } elsif ($READING_MODE == 2) {
          # Reading in cluster sequence or its complement.
          chop;
          $CLUSTER{$FRAGMENT_NAME} .= $_;
       } elsif ($READING_MODE == 3) {
          # Reading in other information.
```

```
            if (/^Assembled_from\*(.*)\s/i) {
                # Read in alignment information.
                chop;
                local(@TEMP_INFO);
                @TEMP_INFO = split(/ /);

                # Reading in left and right contig position indices.
                $ASSEMBLY_INFO1{$TEMP_INFO[1]} = $TEMP_INFO[3];
                $ASSEMBLY_INFO2{$TEMP_INFO[1]} = $TEMP_INFO[5];
            }
        }
    }
}

    print LOG "\n";
    if ($SHOW_PROGRESS) { print STDERR "\n"; }
    # We're done reading the file.
    close(ACE_FILE);

    if ($NOT_USE_FILE) {
       print LOG "DELETING ACE FILE <$_[0].ace>\n";
       if ($SHOW_PROGRESS) {
           print STDERR "DELETING ACE FILE <$_[0].ace>\n";
       }
       system("rm -f .*$FILENAME_PREFIX*");
    }

    # Now call another subroutine to send the parsed info to the
    # next component in the pipeline in a specified format.
    &call_align_to_contig;
}


# Subroutine that actually invokes the Phrap.
sub call_phrap {
    print LOG "INVOKING $PHRAP_COMMAND_PREFIX to assemble contig(s) of file
<$_[0]>...\n";
    if ($SHOW_PROGRESS) {
       print STDERR "INVOKING $PHRAP_COMMAND_PREFIX to assemble contig(s) of file
<$_[0]>...\n";
    }

    # Execute Phrap through shell.
    unless (fork) {
       exec("$PHRAP_COMMAND_PREFIX $_[0] $PHRAP_COMMAND_SUFFIX");
    }
    wait;
    $CURRENT_TIME = `date`;
    print LOG "DONE ASSEMBLING <$_[0]> at $CURRENT_TIME";
    if ($SHOW_PROGRESS) {
       print STDERR "DONE ASSEMBLING <$_[0]> at $CURRENT_TIME";
    }

    # Delete unnecessary files, not including the "ace" file.
    system("rm -f $_[0].contigs.qual");
    system("rm -f $_[0].log");
    system("rm -f $_[0].singlets");
}


# Subroutine that takes the information parsed from the "ace" file and
# sends them out through the standard output.
sub call_align_to_contig {
    local($SEQ_LENGTH);
```

```
    local($CLUSTER_COUNTER);
    $CLUSTER_COUNTER = 0;

    # Determine length of the contig sequence.
    $SEQ_LENGTH = length($CONTIG_SEQ);
    print LOG "LENGTH OF CONTIG is $SEQ_LENGTH\n";
    if ($SHOW_PROGRESS) {
       print STDERR "LENGTH OF CONTIG is $SEQ_LENGTH\n";
    }

    # Send out cluster name.
    print STDOUT ":$CLUSTER_NAME;\n";
    # Send out contig sequence.
    print STDOUT "{$CONTIG_SEQ}\n";

    # Now send out each sequence and its alignment information.
    foreach $SEQ (keys %CLUSTER) {
       $CLUSTER_COUNTER++;

       # Determine length of each sequence.
       $SEQ_LENGTH = length($CLUSTER{$SEQ});
       print LOG "[$CLUSTER_COUNTER] ALIGNING <$SEQ> (LENGTH $SEQ_LENGTH) TO POSITIONS
$ASSEMBLY_INFO1{$SEQ} <=> $ASSEMBLY_INFO2{$SEQ}\n";
       if ($SHOW_PROGRESS) {
           print STDERR "[$CLUSTER_COUNTER] ALIGNING <$SEQ> (LENGTH $SEQ_LENGTH) TO
POSITIONS $ASSEMBLY_INFO1{$SEQ} <=> $ASSEMBLY_INFO2{$SEQ}\n";
       }

       # Send out sequence name.
       print STDOUT "/$SEQ\\\n";
       # Send out the sequence's alignment index for the left end.
       print STDOUT "($ASSEMBLY_INFO1{$SEQ})\n";
       # Send out the sequence's alignment index for the right end.
       print STDOUT "<$ASSEMBLY_INFO2{$SEQ}>\n";
       # Send out the sequence's sequence?
        print STDOUT "[$CLUSTER{$SEQ}]\n";
    }
    # Send out end-of-cluster character.
    print STDOUT "\0\n";

    $CURRENT_TIME = `date`;
    print LOG "DONE ALIGNING ESTs to contig at $CURRENT_TIME";
    if ($SHOW_PROGRESS) {
       print STDERR "DONE ALIGNING ESTs to contig at $CURRENT_TIME";
    }
}


# Subroutine for determining number of sequences in a file.
# It assumes the FASTA format is used.
sub get_sequence_number {
    local($COUNTER);
    $COUNTER = 0;
    open(SEQUENCE, $_[0]);
    while (<SEQUENCE>) {
       if (/^>(.*)\s/) {
           $COUNTER++;
       }
    }
    close(SEQUENCE);
    $COUNTER;
}
```

The Single Nucleotide Polymorphism Detector (SNPD) / Analysis.H

```
#ifndef ANALYSIS_H


#include "project.H"
#include "parameter.H"
#include "sequence.H"
#include "utility.H"
#include <iostream.h>
#define ANALYSIS_H


// Declaration of some constants
#define _G 0
#define _A 1
#define _T 2
#define _C 3
#define _N 4
#define _BOTTOM _G
#define _TOP _N


// Declaration of different nucleotide types
enum PositionType {
  UNDEFINED,
  SAME,
  CONFLICTING,
  POLYMORPHIC
  };


class PositionInfo {

// The PositionInfo class is closer to being a C-style struct than
// anything else.  It is essential several arrays for keeping track of
// a number of different attributes at each nucleotide position.  They
// are:
//   occurrence of G:  nucleotide guanine / high quality
//   occurrence of A:  nucleotide adenine / high quality
//   occurrence of T:  nucleotide thymine / high quality
//   occurrence of C:  nucleotide cytosine / high quality
//   occurrence of *:  nucleotide no-call / high quality
//   occurrence of g:  nucleotide guanine / low quality
//   occurrence of a:  nucleotide adenine / low quality
//   occurrence of t:  nucleotide thymine / low quality
//   occurrence of c:  nucleotide cytosine / low quality
//   occurrence of n:  nucleotide no-call / low quality
//   occurrence of *:  padding character
//   total of occurrences:  sum of all above
//   allele frequency G:  occurrence G/g / (occurrence G + highest occurrence)
//   allele frequency A:  occurrence A/a / (occurrence G + highest occurrence)
//   allele frequency T:  occurrence T/t / (occurrence G + highest occurrence)
//   allele frequency C:  occurrence C/c / (occurrence G + highest occurrence)
//   allele frequency N:  occurrence N/n / (occurrence G + highest occurrence)
```

```
//   nucleotide type:  POLYMORPHIC, CONFLICTING, SAME, or UNDEFINED


public:
  PositionInfo(void) {
    TOTAL = HQ_total = LQ_total = 0;

    HQ_bases[_G] = HQ_bases[_A] = HQ_bases[_T] = HQ_bases[_C] =
      HQ_bases[_N] = 0;
    LQ_bases[_G] = LQ_bases[_A] = LQ_bases[_T] = LQ_bases[_C] =
      LQ_bases[_N] = 0;

    // Allele frequencies
    RATIOS[_G] = RATIOS[_A] = RATIOS[_T] = RATIOS[_C] = RATIOS[_N] = 0.0;

    // Occurrence of padding character
    P_bases = 0;

    // Nucleotide type
    POSITION_type = UNDEFINED;

    // Pointer to char array for storing polymorphism string
    POLYMORPHISM = 0;
  };


  ~PositionInfo(void) { if (POLYMORPHISM) delete [] POLYMORPHISM; };
  // Effects:  Destructor for the class.

  // Variables in the class/struct
  int TOTAL;

  int HQ_total;

  int LQ_total;

  int P_bases;

  int HQ_bases[5];

  int LQ_bases[5];

  float RATIOS[5];

  PositionType POSITION_type;

  char* POLYMORPHISM;

protected:

private:

};


class FragmentInfo {

// The FragmentInfo class is a class/struct based on the Sequence class.
// Each object contains the following variables:
//   START_POS:  The nucleotide position on the contig sequence to which
//               the FIRST nucleotide on the fragment should be aligned.
//   END_POS:  The nucleotide position on the contig sequence to which
//             the LAST nucleotide on the fragment should be aligned.
//   FRAGMENT_SEQ:  The sequence of the fragment
```

```
//   FRAGMENT_NAME:  The name of the fragment, if available

public:

  FragmentInfo(void) {
    // Effects:  Constructor for the class.  Initializes the variables.
    START_POS = 0;
    END_POS = 0;
    FRAGMENT_SEQ = 0;
    FRAGMENT_NAME = 0;
  };


  ~FragmentInfo(void) {
    // Effects:  Destructor for the class.  Deletes the fragment name, if
    //           one exists.
    delete FRAGMENT_SEQ;
    if (FRAGMENT_NAME) delete [] FRAGMENT_NAME;
  };


  int START_POS;

  int END_POS;

  Sequence* FRAGMENT_SEQ;

  char* FRAGMENT_NAME;

protected:

private:

};


class ConsensusSequence {

// The ConsensusSequence class is another class based on the Sequence class.
// Each class object has the following attributes:
//   CONSENSUS_SEQ:  Holds the sequence for the consensus sequence
//   FRONT_INDEX_IN_CONTIG:  The nucleotide position in the contig sequence
//                           to which the FIRST nucleotide in the consensus
//                           sequence is aligned.
//   BACK_INDEX_IN_CONTIG:  The nucleotide position in the contig sequence
//                          to which the LAST nucleotide in the consensus
//                          sequence is aligned.
//   POLYMORPHIC_SITES:  An array of polymorphism sites (integers)

public:
  ConsensusSequence(Sequence* consensusSeq, int frontIndexInContig,
                    int backIndexInContig) {
    // Effects:  Constructor for the class.
    FRONT_INDEX_IN_CONTIG = frontIndexInContig;
    BACK_INDEX_IN_CONTIG = backIndexInContig;
    CONSENSUS_SEQ = consensusSeq;
    POLYMORPHIC_SITES = new Array;
  };


  ~ConsensusSequence(void) {
    // Effects:  Destructor for the class.
    delete CONSENSUS_SEQ;
    delete POLYMORPHIC_SITES;
```

```
  };


  int FRONT_INDEX_IN_CONTIG;
  int BACK_INDEX_IN_CONTIG;
  Sequence* CONSENSUS_SEQ;
  Array* POLYMORPHIC_SITES;

protected:

private:

};


class AlignedCluster {

// The AlignedCluster class allows one to specify a contig sequence, and
// other sequences that are aligned to the contig sequence.  It has a
// variety of functions specifically for analyzing the aligned sequences
// to identify potential single nucleotide polymorphisms in the sequence
// assembly.

public:
  AlignedCluster(Parameter* params, Sequence* contigSeq, char* clusterName);
  // Effects:  Constructor for the class.  The pointer for a Parameter object
  //           which has the values for the criteria/parameters that are
  //           used in identifying potential SNPs must be provided.  A
  //           contig sequence must also be specified.


  ~AlignedCluster(void);
  // Effects:  Destructor for the class.  All sequences stored in the
  //           AlignedCluster object are destroyed.


  void addFragment(Sequence* fragSeq, char* fragName,
                             int startPos, int endPos);
  // Effects:  Add the sequence fragment to the contig sequence, so that
  //           the first nucleotide of the fragment is lined up with the
  //           nucleotide position <startPos> in the contig sequence, and
  //           the last nucleotide is lined up with the nucleotide position
  //           <endPos> in the contig sequence.


  int clusterSize(void) { return pCLUSTER_SIZE; };
  // Effects:  Return the number of fragments aligned to the contig sequence.


  void analyzePosition(int i, bool useHighQualOnly = TRUE);
  // Effects:  Analyze a nucleotide position overlapped by the contig
  //           sequence.  After it is analyzed, it is classified as UNDEFINED,
  //           POLYMORPHIC, CONFLICTING, or SAME.


  void analyzeAllPositions(bool useHighQualOnly = TRUE);
  // Effects:  Analyze all the nucleotide positions which are overlapped by
  //           the contig sequence.  After a nucleotide position has been
  //           analyzed, it is classified as UNDEFINED, POLYMORPHIC,
  //           CONFLICTING, or SAME.


  void printPositionInfo(int i, ostream& output = cout);
  // Effects:  Prints information about the nucleotide at contig position
```

```
  //           <i> to <output>.  By default, <output> is standard output.
  //           Information such as the number of each type of nucleotide, etc.
  //           is printed.


  void printAllPositions(ostream& output = cout);
  // Effects:  Calls printPositionInfo to print info on all nucleotide
  //           positions to <output>.  By default, <output> is standard
  //           output.


  void printConflictPositions(bool useHighQualOnly = TRUE,
                              ostream& output = cout);
  // Effects:  Prints all nucleotides that show multiple alleles after
  //           the nucleotides have been analyzed.  If <useHighQualOnly>
  //           is TRUE, only high quality bases at each contig position
  //           are considered for analysis.  Print result to <output>
  //           which is standard output by default.


  void printPolymorphicPositions(bool useHighQualOnly = TRUE,
                                 ostream& output = cout);
  // Effects:  Prints each nucleotide that is potentially polymorphic after
  //           the nucleotide has been analyzed.  If <useHighQualOnly>
  //           is TRUE, only high quality bases at each contig position
  //           are considered for analysis.  Print result to <output>
  //           which is standard output by default.


  void printPolymorphismWindow(int contigPos, int windowSize = -1,
                               ostream& output = cout);
  // Effects:  Print nucleotide sequences of all the sequences that are
  //           within a distance of <windowSize> bases to the left or
  //           to the right of the nucleotides lined up at the contig
  //           position <contigPos>.  Print to <output>.  By default
  //           <output> is the standard output.


  void outputPrimerBoulderInput(ostream& output = cout);
  // Effects:  Print the data needed by PRIMER to pick PCR primers
  //           that flank the potentially polymorphic nucleotides
  //           identified.  The format used is Boulder IO, which is
  //           accepted by PRIMER.


protected:

private:

  // Pointer the contig sequence.  If NIL, the sequence is nonexistent.
  Sequence* pCONTIG_SEQ;

  // An array of pointers to objects which are for used for keeping track
  // of various attributes of a nucleotide position.
  PositionInfo* pCONTIG_INFO;

  // Length of the contig sequence
  int pCONTIG_LEN;

  // Number of sequences aligned to the contig sequence
  int pCLUSTER_SIZE;

  // An array of pointers to sequence fragments
  FragmentInfo** pFRAG_INFO;
```

```
// Variable to keep track of the size of the array for storing pointers
// to sequence fragments
int pFRAG_ALLOC_SIZE;

// Pointer to the Parameter object containing various parameter values
Parameter* pPARAMS;

// Pointer to name of the cluster/sequence assembly
char* pCLUSTER_NAME;

// Pointer to an array of ConsensusSequence objects
Array* pCONSENSUS_SEQ_ARRAY;


void analyzeFragment(FragmentInfo* fragInfo);
// Effects:  Analyzes a fragment nucleotide by nucleotide.  The
//           information for each contig position is updated accordingly.


void getOverlappingSeqIndexes(int includedSite, int overlapLevel,
                              int& frontEnd, int& backEnd,
                              bool useHighQualOnly = TRUE);
// Effects:  Sets <frontEnd> to the leftmost point in the aligned
//           sequence assembly that is overlapped by <overlapLevel>
//           sequences starting from <includedSite>.  Do the same for
//           <backEnd> in the opposite direction.  Consider only high
//           quality nucleotides if <useHighQualOnly> is TRUE.


bool isConflictingBase(int contigPos, bool useHighQualOnly = TRUE);
// Effects:  Return TRUE if the sequences exhibit more than one allele
//           at the contig position <contigPos>.  Return FALSE otherwise.
//           Consider only high quality nucleotides if <useHighQualOnly>
//           is TRUE.


bool isPolymorphicBase(int contigPos, bool useHighQualOnly = TRUE);
// Effects:  Return TRUE if the nucleotide at contig position <contigPos>
//           is potentially polymorphic.  Return FALSE otherwise.


bool isPolymorphicAndCanBeSequenced(int i, int consensusOverlapLevel,
                                    int& frontEnd, int& backEnd,
                                    bool useHighQualOnly = TRUE);
// Effects:  Return TRUE if the potentially polymorphic nucleotide at
//           contig position <i> has a related consensus sequence that
//           has an unambiguous region both upstream and downstream of
//           the nucleotide in question.  The consensus sequence must
//           be confirmed by <consensusOverlapLevel> sequences.


int longestUnambiguousSeqLength(int frontEnd, int backEnd,
                                bool useHighQualOnly = TRUE);
// Effects:  Return the length of the longest umambiguous region in
//           the aligned sequences between the contig positions <frontEnd>
//           and <backEnd>.


char determineConsensusBase(int contigPos, bool useHighQualOnly = TRUE);
// Effects:  Determine the identity of the nucleotide for the contig
//           position <contigPos>.  Consider only high quality nucleotides
//           if <useHighQualOnly> is TRUE.
```

```
  char resolveConflictBase(int contigPos, bool useHighQualOnly = TRUE);
  // Effects:  Determine the probable nucleotide at contig position
  //           <contigPos>.  Consider only high quality nucleotides if
  //           <useHighQualOnly> is TRUE.


  int getMostFrequentBase(int contigPos, bool useHighQualOnly = TRUE);
  // Effects:  Get the nucleotide (G/A/T/C) with the highest occurrence
  //           at the contig position <contigPos>.  Consider only high
  //           quality nucleotides if <useHighQualOnly> is TRUE.


  char getBaseCharacter(int i);
  // Effects:  Translate nucleotide array index <i> into its character
  //           equivalent.  The mapping is as follows:
  //           0 ==> G
  //           1 ==> A
  //           2 ==> T
  //           3 ==> C
  //           4 ==> N


  PositionType getPositionType(int contigPos, bool useHighQualOnly = TRUE);
  // Effects:  Return the position type at <contigPos>:  POLYMORPHIC,
  //           CONFLICTING, SAME, or UNDEFINED.  Consider only high quality
  //           nucleotides if <useHighQualOnly> is TRUE.


  bool addSequenceToConsensusArray(Sequence* newSequence, int contigPos,
                                   int frontIndexInContig,
                                   int backIndexInContig);
  // Effects:  Add sequence <newSequence> containing the polymorphic
  //           nucleotide at contig position <contigPos> to consensus
  //           sequence array.

};

#endif
```

The Single Nucleotide Polymorphism Detector (SNPD) / Analysis.C

```
#include "analysis.H"
#include <string.h>

#define ALLOC_BLOCK_SIZE 10

extern int gNumSNPs;
extern int gNumClusterWithSNP;
extern int gNumClusterWithSNPConsensus;


char AlignedCluster::getBaseCharacter(int i)
//
// Effects:  Translate nucleotide array index <i> into its character
//           equivalent.  The mapping is as follows:
//           0 ==> G
//           1 ==> A
//           2 ==> T
//           3 ==> C
//           4 ==> N
//
{
  switch (i) {
  case _G:
    return 'G';
  case _A:
    return 'A';
  case _T:
    return 'T';
  case _C:
    return 'C';
  case _N:
    return 'N';
  }
}


AlignedCluster::AlignedCluster(Parameter* params, Sequence* contigSeq,
                               char* clusterName)
//
// Effects:  Constructor for the class.  The pointer for a Parameter object
//           which has the values for the criteria/parameters that are
//           used in identifying potential SNPs must be provided.  A
//           contig sequence must also be specified.
//
{
  pCONTIG_SEQ = contigSeq;
  pCONTIG_LEN = contigSeq->length();
  pCONTIG_INFO = new PositionInfo[pCONTIG_LEN];
  pCLUSTER_SIZE = 0;
  pFRAG_INFO = new (FragmentInfo*)[(pFRAG_ALLOC_SIZE = ALLOC_BLOCK_SIZE)];
  pPARAMS = params;
  pCONSENSUS_SEQ_ARRAY = new Array;
  if (clusterName) {
```

```
    pCLUSTER_NAME = new char[strlen(clusterName) + 1];
    strcpy(pCLUSTER_NAME, clusterName);
    for (int c=0; pCLUSTER_NAME[c]; c++)
      if (pCLUSTER_NAME[c] == ' ')
       pCLUSTER_NAME[c] = '_';
  }
  else
    pCLUSTER_NAME = 0;
}


AlignedCluster::~AlignedCluster(void)
//
// Effects:  Destructor for the class.  All sequences stored in the
//           AlignedCluster object are destroyed.
//
{
  delete pCONTIG_SEQ;
  delete [] pCONTIG_INFO;
  for (int i=0; i<pCLUSTER_SIZE; i++)    delete pFRAG_INFO[i];
  delete [] pFRAG_INFO;

  for (i=0; i<pCONSENSUS_SEQ_ARRAY->size(); i++) {
    delete (ConsensusSequence*)pCONSENSUS_SEQ_ARRAY->element(i);
  }
  delete pCONSENSUS_SEQ_ARRAY;

  if (pCLUSTER_NAME)
    delete [] pCLUSTER_NAME;
}


void AlignedCluster::addFragment(Sequence* fragSeq, char* fragName,
                                 int startPos, int endPos)
//
// Effects:  Add the sequence fragment to the contig sequence, so that
//           the first nucleotide of the fragment is lined up with the
//           nucleotide position <startPos> in the contig sequence, and
//           the last nucleotide is lined up with the nucleotide position
//           <endPos> in the contig sequence.
//
{
  FragmentInfo* newFrag = new FragmentInfo;
  newFrag->START_POS = startPos;
  newFrag->END_POS = endPos;
  newFrag->FRAGMENT_SEQ = fragSeq;
  if (fragName) {
    newFrag->FRAGMENT_NAME = new char[strlen(fragName)+1];
    strcpy(newFrag->FRAGMENT_NAME, fragName);
  }
  pFRAG_INFO[pCLUSTER_SIZE++] = newFrag;
  if ((pCLUSTER_SIZE % ALLOC_BLOCK_SIZE) == 0) {
    FragmentInfo** tempArray = new (FragmentInfo*)
      [pFRAG_ALLOC_SIZE + ALLOC_BLOCK_SIZE];
    for (int i=0; i<pCLUSTER_SIZE; i++)
      tempArray[i] = pFRAG_INFO[i];
    delete [] pFRAG_INFO;
    pFRAG_INFO = tempArray;
    pFRAG_ALLOC_SIZE += ALLOC_BLOCK_SIZE;
  }
  analyzeFragment(newFrag);
}
```

```
void AlignedCluster::analyzeFragment(FragmentInfo* fragInfo)
//
// Effects:  Analyze all the nucleotide positions which are overlapped by
//           the contig sequence.  After a nucleotide position has been
//           analyzed, it is classified as UNDEFINED, POLYMORPHIC,
//           CONFLICTING, or SAME.
//
{
  int startIndex = fragInfo->START_POS;
  if (fragInfo->START_POS < 0)
    startIndex = 0;

  for (int i=startIndex; i<pCONTIG_LEN; i++) {
    if ((i >= fragInfo->START_POS) && (i <= fragInfo->END_POS)) {
      char c = fragInfo->FRAGMENT_SEQ->getChar(i - fragInfo->START_POS);
      switch (c) {
      case 'G':
       pCONTIG_INFO[i].HQ_bases[_G]++;
       break;
      case 'A':
       pCONTIG_INFO[i].HQ_bases[_A]++;
       break;
      case 'T':
       pCONTIG_INFO[i].HQ_bases[_T]++;
       break;
      case 'C':
       pCONTIG_INFO[i].HQ_bases[_C]++;
       break;
      case 'N':
       pCONTIG_INFO[i].HQ_bases[_N]++;
       break;
      case 'g':
       pCONTIG_INFO[i].LQ_bases[_G]++;
       break;
      case 'a':
       pCONTIG_INFO[i].LQ_bases[_A]++;
       break;
      case 't':
       pCONTIG_INFO[i].LQ_bases[_T]++;
       break;
      case 'c':
       pCONTIG_INFO[i].LQ_bases[_C]++;
       break;
      case 'n':
       pCONTIG_INFO[i].LQ_bases[_N]++;
       break;
      case '*':
       pCONTIG_INFO[i].P_bases++;
       break;
      }
    }
  }
}


void AlignedCluster::printConflictPositions(bool useHighQualOnly,
                                            ostream& output)
//
// Effects:  Prints all nucleotides that show multiple alleles after
//           the nucleotides have been analyzed.  If <useHighQualOnly>
//           is TRUE, only high quality bases at each contig position
//           are considered for analysis.  Print result to <output>
//           which is standard output by default.
//
```

```
{
  for (int i=0; i<pCONTIG_LEN; i++) {
    PositionType positionType = getPositionType(i, useHighQualOnly);
    if ((positionType== CONFLICTING) || (positionType == POLYMORPHIC))
      printPositionInfo(i, output);
  }
}


void AlignedCluster::printPolymorphismWindow(int contigPos, int windowSize,
                                             ostream& output)
//
// Effects:  Print nucleotide sequences of all the sequences that are
//           within a distance of <windowSize> bases to the left or
//           to the right of the nucleotides lined up at the contig
//           position <contigPos>.  Print to <output>.  By default
//           <output> is the standard output.
//
{
  if (windowSize < 0 )
    windowSize = pPARAMS->qual_window_size;
  int leftEdgePos = contigPos - windowSize;
  int rightEdgePos = contigPos + windowSize;

  for (short i=0; i<pCLUSTER_SIZE; i++) {
    FragmentInfo* currentFrag = pFRAG_INFO[i];
    for (short j=leftEdgePos; j<=rightEdgePos; j++) {
      if (j == contigPos) output << "[";
      if ((j >= currentFrag->START_POS) && (j <= currentFrag->END_POS)) {
       output << currentFrag->FRAGMENT_SEQ->getChar
         (j - currentFrag->START_POS);
      }
      else {
       output << "-";
      }
      if (j == contigPos) output << "]";
    }
    if (currentFrag->FRAGMENT_NAME)
      output << "    " << currentFrag->FRAGMENT_NAME << "\n";
  }
}


void AlignedCluster::printPolymorphicPositions(bool useHighQualOnly,
                                               ostream& output)
//
// Effects:  Prints each nucleotide that is potentially polymorphic after
//           the nucleotide has been analyzed.  If <useHighQualOnly>
//           is TRUE, only high quality bases at each contig position
//           are considered for analysis.  Print result to <output>
//           which is standard output by default.
//
{
  int numberOfPolymorphisms = 0;
  //////////////////////////
  int numberOfPolyTEMP = 0;
  //////////////////////////

  for (int i=0; i<pCONTIG_LEN; i++) {
    if (isPolymorphicBase(i, useHighQualOnly)) {

      //////////////////////////
      gNumSNPs++;
      numberOfPolyTEMP++;
```

```
if (numberOfPolyTEMP <= 1)
 gNumClusterWithSNP++;
//////////////////////////

int frontEnd;
int backEnd;

if (!isPolymorphicAndCanBeSequenced(i, pPARAMS->min_consensus_overlap,
                              frontEnd, backEnd, useHighQualOnly))
 continue;

int numberOfN = 0;
Sequence* newConsensusSequence = new Sequence;
for (int j=frontEnd; j<=backEnd; j++) {
 char c = determineConsensusBase(j, useHighQualOnly);
 newConsensusSequence->putChar(c);
 if (c == 'N') numberOfN++;
}
addSequenceToConsensusArray(newConsensusSequence, i, frontEnd, backEnd);

numberOfPolymorphisms++;

if (output) {
 if (numberOfPolymorphisms > 1)
   output << "................................................\n";
 else {
   output << "::::::::::::::\n";
   output << pCLUSTER_NAME << "\n";
   output << "::::::::::::::\n\n";

   output << "CONTIG_SEQUENCE=";
   pCONTIG_SEQ->printSequence(TRUE, output);
   output << "CONTIG_SEQUENCE_LENGTH=";
   output << pCONTIG_LEN << "\n";
   output << "\n";
 }

 printPositionInfo(i, output);

 if (pPARAMS->print_qual_window) {
   printPolymorphismWindow(i, pPARAMS->print_qual_window_size, output);
   output << "\n";
 }

 output << "POLYMORPHISM_NAME=";
 if (pCLUSTER_NAME)
   output << pCLUSTER_NAME;
 else
   output << "UNKNOWN";
 output << "-" << (i+1) << "\n";

 output << "POLYMORPHISM=" << pCONTIG_INFO[i].POLYMORPHISM << "\n";
 output << "POLYMORPHISM_POSITION_IN_CONTIG=" << (i+1) << "\n";
 output << "\n";

 output << "SEQUENCE=";
 newConsensusSequence->printSequence(FALSE, output);

 output << "\n";
 output << "SEQUENCE_N_CONTENT="
   << (100.0 * numberOfN / (backEnd - frontEnd + 1)) << "\n";

 output << "SEQUENCE_LENGTH=" << (backEnd - frontEnd + 1) << "\n";
 output << "POLYMORPHISM_POSITION_IN_CONSENSUS=" << (i - frontEnd + 1)
```

```
          << "\n";
        output << "\n";
      }
    }
    if (numberOfPolymorphisms > 0)
      output << "\n\n";
  }
}


bool AlignedCluster::isPolymorphicAndCanBeSequenced(int i,
                                                    int consensusOverlapLevel,
                                                    int& frontEnd,
                                                    int& backEnd,
                                                    bool useHighQualOnly)
//
// Effects:  Return TRUE if the potentially polymorphic nucleotide at
//           contig position <i> has a related consensus sequence that
//           has an unambiguous region both upstream and downstream of
//           the nucleotide in question.  The consensus sequence must
//           be confirmed by <consensusOverlapLevel> sequences.
//
{
  if (!isPolymorphicBase(i, useHighQualOnly))
    return FALSE;

  getOverlappingSeqIndexes(i, consensusOverlapLevel, frontEnd, backEnd,
                           useHighQualOnly);
  if ((backEnd - frontEnd + 1) < pPARAMS->min_consensus_length) {
    return FALSE;
  }

  if (((i - frontEnd) < (pPARAMS->min_primer_length
                         + pPARAMS->min_length_upstream_of_SNP)) ||
      ((backEnd - i) < (pPARAMS->min_primer_length
                        + pPARAMS->min_length_downstream_of_SNP))) {
    return FALSE;
  }

  if (longestUnambiguousSeqLength(frontEnd,
                                  i - pPARAMS->min_length_upstream_of_SNP,
                                  useHighQualOnly)
      < pPARAMS->min_primer_length) {
    return FALSE;
  }

  if (longestUnambiguousSeqLength(i + pPARAMS->min_length_downstream_of_SNP,
                                  backEnd, useHighQualOnly)
      < pPARAMS->min_primer_length) {
    return FALSE;
  }

  return TRUE;
}


int AlignedCluster::longestUnambiguousSeqLength(int frontEnd, int backEnd,
                                                bool useHighQualOnly)
//
// Effects:  Return the length of the longest umambiguous region in
//           the aligned sequences between the contig positions <frontEnd>
//           and <backEnd>.
//
{
```

```
  int longestUnambiguousSeqLen = 0;

  int tempLen = 0;
  for (int i=frontEnd; i<=backEnd; i++) {
    if (determineConsensusBase(i, useHighQualOnly) == 'N') {
      if (tempLen > longestUnambiguousSeqLen)
       longestUnambiguousSeqLen = tempLen;
      tempLen = 0;
    }
    else
      tempLen++;
  }
  if (tempLen > longestUnambiguousSeqLen)
    longestUnambiguousSeqLen = tempLen;

  return longestUnambiguousSeqLen;
}


bool AlignedCluster::isConflictingBase(int contigPos, bool useHighQualOnly)
//
// Effects:  Return TRUE if the sequences exhibit more than one allele
//           at the contig position <contigPos>.  Return FALSE otherwise.
//           Consider only high quality nucleotides if <useHighQualOnly>
//           is TRUE.
//
{
  int counter = 0;

  for (int i=_BOTTOM; i<_TOP; i++) {
    if (pCONTIG_INFO[contigPos].HQ_bases[i] > 0) {
      counter++;
      continue;
    }
    if (!useHighQualOnly) {
      if (pCONTIG_INFO[contigPos].LQ_bases[i] > 0)
       counter++;
    }
  }

  if (counter > 1) {
    return TRUE;
  }

  return FALSE;
}


void AlignedCluster::analyzeAllPositions(bool useHighQualOnly)
//
// Effects:  Analyze all the nucleotide positions which are overlapped by
//           the contig sequence.  After a nucleotide position has been
//           analyzed, it is classified as UNDEFINED, POLYMORPHIC,
//           CONFLICTING, or SAME.
//
{
  for (int i=0; i<pCONTIG_LEN; i++)
    analyzePosition(i, useHighQualOnly);
}


void AlignedCluster::analyzePosition(int i, bool useHighQualOnly)
//
// Effects:  Analyze a nucleotide position overlapped by the contig
```

```
//            sequence.  After it is analyzed, it is classified as UNDEFINED,
//            POLYMORPHIC, CONFLICTING, or SAME.
//
{
  PositionInfo* pos = &pCONTIG_INFO[i];

  float mostAbundant = -1;
  for (int j=_BOTTOM; j<=_TOP; j++) {
    pos->HQ_total += pos->HQ_bases[j];
    pos->LQ_total += pos->LQ_bases[j];

    int subtotal = pos->HQ_bases[j];
    if (!useHighQualOnly)
      subtotal += pos->LQ_bases[j];
    if (subtotal > mostAbundant)
      mostAbundant = subtotal;
  }

  pos->TOTAL = pos->HQ_total + pos->LQ_total + pos->P_bases;

  if (mostAbundant > 0) {
    for (j=_BOTTOM; j<=_TOP; j++) {
      int topValue = pos->HQ_bases[j];
      if (!useHighQualOnly)
       topValue += pos->LQ_bases[j];
      pos->RATIOS[j] = topValue * 100 /
       (topValue + mostAbundant);
    }
  }
}


void AlignedCluster::printAllPositions(ostream& output)
//
// Effects:  Calls printPositionInfo to print info on all nucleotide
//           positions to <output>.  By default, <output> is standard
//           output.
//
{
  for (int i=0; i<pCONTIG_LEN; i++)
    printPositionInfo(i, output);
}


void AlignedCluster::printPositionInfo(int i, ostream& output)
//
// Effects:  Prints information about the nucleotide at contig position
//           <i> to <output>.  By default, <output> is standard output.
//           Information such as the number of each type of nucleotide, etc.
//           is printed.
//
{
  output << "CONTIG POSITION (" << (i+1) << ") ["
    << pCONTIG_SEQ->getChar(i) << "]\n";
  output << "TOTAL OVERLAP=" << pCONTIG_INFO[i].TOTAL << "\n";
  output << "HIGH QUALITY OVERLAP=" << pCONTIG_INFO[i].HQ_total << "\n";
  output << "LOW QUALITY OVERLAP=" << pCONTIG_INFO[i].LQ_total << "\n";
  output << "\n";

  output.precision(3);

  output << "TOTAL G="
    << (pCONTIG_INFO[i].HQ_bases[_G] + pCONTIG_INFO[i].LQ_bases[_G])
      << "\t\tR=" << pCONTIG_INFO[i].RATIOS[_G]
```

```
        << "\t\tG=" << pCONTIG_INFO[i].HQ_bases[_G]
          << "\tg=" << pCONTIG_INFO[i].LQ_bases[_G]
            << "\n";
  output << "TOTAL A="
    << (pCONTIG_INFO[i].HQ_bases[_A] + pCONTIG_INFO[i].LQ_bases[_A])
      << "\t\tR=" << pCONTIG_INFO[i].RATIOS[_A]
        << "\t\tA=" << pCONTIG_INFO[i].HQ_bases[_A]
          << "\ta=" << pCONTIG_INFO[i].LQ_bases[_A]
            << "\n";
  output << "TOTAL T="
    << (pCONTIG_INFO[i].HQ_bases[_T] + pCONTIG_INFO[i].LQ_bases[_T])
      << "\t\tR=" << pCONTIG_INFO[i].RATIOS[_T]
        << "\t\tT=" << pCONTIG_INFO[i].HQ_bases[_T]
          << "\tt=" << pCONTIG_INFO[i].LQ_bases[_T]
            << "\n";
  output << "TOTAL C="
    << (pCONTIG_INFO[i].HQ_bases[_C] + pCONTIG_INFO[i].LQ_bases[_C])
      << "\t\tR=" << pCONTIG_INFO[i].RATIOS[_C]
        << "\t\tC=" << pCONTIG_INFO[i].HQ_bases[_C]
          << "\tc=" << pCONTIG_INFO[i].LQ_bases[_C]
            << "\n";
  output << "TOTAL N="
    << (pCONTIG_INFO[i].HQ_bases[_N] + pCONTIG_INFO[i].LQ_bases[_N])
       << "\t\t\t\tN=" << pCONTIG_INFO[i].HQ_bases[_N]
          << "\tn=" << pCONTIG_INFO[i].LQ_bases[_N]
            << "\n";

  output << "TOTAL *=" << pCONTIG_INFO[i].P_bases << "\n";
  output << "\n";
}


char AlignedCluster::determineConsensusBase(int contigPos,
                                            bool useHighQualOnly)
//
// Effects:  Determine the identity of the nucleotide for the contig
//           position <contigPos>.  Consider only high quality nucleotides
//           if <useHighQualOnly> is TRUE.
//
{
  PositionType positionType = getPositionType(contigPos, useHighQualOnly);

  switch (positionType) {
  case SAME:
    return getBaseCharacter(getMostFrequentBase(contigPos, useHighQualOnly));
    break;
  case POLYMORPHIC:
    return 'N';
    break;
  case CONFLICTING:
    return resolveConflictBase(contigPos, useHighQualOnly);
    break;
  default:
    return 'N';
    break;
  }
  return 'N';
}


char AlignedCluster::resolveConflictBase(int contigPos, bool useHighQualOnly)
//
// Effects:  Determine the probable nucleotide at contig position
//           <contigPos>.  Consider only high quality nucleotides if
```

```
//           <useHighQualOnly> is TRUE.
//
{
  int mostFreqBase = getMostFrequentBase(contigPos, useHighQualOnly);
  int mostFreqBaseNumber = pCONTIG_INFO[contigPos].HQ_bases[mostFreqBase];
  if (!useHighQualOnly) {
    mostFreqBaseNumber += pCONTIG_INFO[contigPos].LQ_bases[mostFreqBase];
  }

  if (mostFreqBaseNumber <= 2)
    return 'N';

  for (int i=_BOTTOM; i<_TOP; i++) {
    if (i == mostFreqBase)
      continue;
    int totalBases = pCONTIG_INFO[contigPos].HQ_bases[i];
    if (!useHighQualOnly)
      totalBases = pCONTIG_INFO[contigPos].LQ_bases[i];

    if (((mostFreqBaseNumber - totalBases) * 100.0
        / (mostFreqBaseNumber + totalBases)) < 50.0)
      return 'N';
  }

  return (getBaseCharacter(mostFreqBase));
}


PositionType AlignedCluster::getPositionType(int contigPos,
                                             bool useHighQualOnly)
//
// Effects:  Return the position type at <contigPos>:  POLYMORPHIC,
//           CONFLICTING, SAME, or UNDEFINED.  Consider only high quality
//           nucleotides if <useHighQualOnly> is TRUE.
//
{
  if (pCONTIG_INFO[contigPos].POSITION_type != UNDEFINED)
    return pCONTIG_INFO[contigPos].POSITION_type;

  if (isPolymorphicBase(contigPos, useHighQualOnly))
    return POLYMORPHIC;
  else
    return pCONTIG_INFO[contigPos].POSITION_type;
}


void AlignedCluster::getOverlappingSeqIndexes(int includedSite,
                                              int overlapLevel,
                                              int& frontEnd, int& backEnd,
                                              bool useHighQualOnly)
//
// Effects:  Sets <frontEnd> to the leftmost point in the aligned
//           sequence assembly that is overlapped by <overlapLevel>
//           sequences starting from <includedSite>.  Do the same for
//           <backEnd> in the opposite direction.  Consider only high
//           quality nucleotides if <useHighQualOnly> is TRUE.
//
{
  frontEnd = 1;
  backEnd = -1;

  for (int i=includedSite; i>=0; i--) {
    int level = pCONTIG_INFO[i].HQ_total;
    if (!useHighQualOnly)
```

```
      level += pCONTIG_INFO[i].LQ_total;
    if (level < overlapLevel)
      break;
    frontEnd = i;
  }

  for (i=includedSite; i<pCONTIG_LEN; i++) {
    int level = pCONTIG_INFO[i].HQ_total;
    if (!useHighQualOnly)
      level += pCONTIG_INFO[i].LQ_total;
    if (level < overlapLevel)
      return;
    backEnd = i;
  }
}


bool AlignedCluster::isPolymorphicBase(int contigPos, bool useHighQualOnly)
//
// Effects:  Return TRUE if the nucleotide at contig position <contigPos>
//           is potentially polymorphic.  Return FALSE otherwise.
//
{
  if (pCONTIG_INFO[contigPos].POSITION_type != UNDEFINED) {
    if (pCONTIG_INFO[contigPos].POSITION_type == POLYMORPHIC)
      return TRUE;
    else
      return FALSE;
  }

  char polymorphism[6];
  char* polymorphicBases = polymorphism;
  *polymorphicBases = 0;

  int overlapLevel = pCONTIG_INFO[contigPos].HQ_total;
  if (!useHighQualOnly)
    overlapLevel += pCONTIG_INFO[contigPos].LQ_total;

  if (overlapLevel < (pPARAMS->min_qual_reads*2)) {
    if (!isConflictingBase(contigPos, useHighQualOnly))
      pCONTIG_INFO[contigPos].POSITION_type = SAME;
    else
      pCONTIG_INFO[contigPos].POSITION_type = CONFLICTING;
    return FALSE;
  }
  else {
    if (!isConflictingBase(contigPos, useHighQualOnly)) {
      pCONTIG_INFO[contigPos].POSITION_type = SAME;
      return FALSE;
    }
    else {
      pCONTIG_INFO[contigPos].POSITION_type = CONFLICTING;
    }
  }

  int couldBePolymorphic = 0;
  for (int i=_BOTTOM; i<_TOP; i++) {
    if (pCONTIG_INFO[contigPos].RATIOS[i]>=pPARAMS->min_polymorphic_ratio) {
      int numberReads = pCONTIG_INFO[contigPos].HQ_bases[i];
      if (!useHighQualOnly)
       numberReads += pCONTIG_INFO[contigPos].LQ_bases[i];
      if (numberReads >= pPARAMS->min_qual_reads) {
       couldBePolymorphic++;
       *(polymorphicBases++) = getBaseCharacter(i);
```

```
      }
    }
  }
  *polymorphicBases = 0;
  if (couldBePolymorphic < 2) {
    return FALSE;
  }

  int windowStart = contigPos - pPARAMS->qual_window_size;
  if (windowStart < 0)
    windowStart = 0;
  int windowEnd = contigPos + pPARAMS->qual_window_size;
  if (windowEnd >= pCONTIG_LEN)
    windowEnd = pCONTIG_LEN - 1;

  float totalExamined = 0;
  float totalAmbiguous = 0;
  for (i=windowStart; i<=windowEnd; i++) {
    totalExamined += pCONTIG_INFO[i].TOTAL;
    totalAmbiguous += pCONTIG_INFO[i].HQ_bases[_N] +
      pCONTIG_INFO[i].LQ_bases[_N] + pCONTIG_INFO[i].P_bases;
  }

  if ((totalAmbiguous * 100 / totalExamined)
      > pPARAMS->qual_window_threshold) {
  }

  pCONTIG_INFO[contigPos].POSITION_type = POLYMORPHIC;
  pCONTIG_INFO[contigPos].POLYMORPHISM = new char[strlen(polymorphicBases)+1];
  strcpy(pCONTIG_INFO[contigPos].POLYMORPHISM, polymorphism);

  return TRUE;
}


int AlignedCluster::getMostFrequentBase(int contigPos, bool useHighQualOnly)
//
// Effects:  Get the nucleotide (G/A/T/C) with the highest occurrence
//           at the contig position <contigPos>.  Consider only high
//           quality nucleotides if <useHighQualOnly> is TRUE.
//
{
  int mostFrequentBase = 0;
  int mostFrequentBaseNum = 0;

  for (int i=_BOTTOM; i<_TOP; i++) {
    int totalBases = pCONTIG_INFO[contigPos].HQ_bases[i];
    if (!useHighQualOnly)
      totalBases += pCONTIG_INFO[contigPos].LQ_bases[i];
    if (totalBases > mostFrequentBaseNum) {
      mostFrequentBaseNum = totalBases;
      mostFrequentBase = i;
    }
  }

  return mostFrequentBase;
}


bool AlignedCluster::addSequenceToConsensusArray(Sequence* newSequence,
                                                 int contigPos,
                                                 int frontIndexInContig,
                                                 int backIndexInContig)
//
```

```
// Effects:  Add sequence <newSequence> containing the polymorphic
//           nucleotide at contig position <contigPos> to consensus
//           sequence array.
//
{
  for (int i=0; i<pCONSENSUS_SEQ_ARRAY->size(); i++) {
    ConsensusSequence* cs=(ConsensusSequence*)pCONSENSUS_SEQ_ARRAY->element(i);
    if ((*newSequence == *(cs->CONSENSUS_SEQ)) &&
        (frontIndexInContig == cs->FRONT_INDEX_IN_CONTIG) &&
        (backIndexInContig == cs->BACK_INDEX_IN_CONTIG)) {
      cs->POLYMORPHIC_SITES->addElementHigh((void*)contigPos);
      return FALSE;
    }
  }

  ConsensusSequence* newConSeq = new ConsensusSequence(newSequence,
                                                       frontIndexInContig,
                                                       backIndexInContig);
  newConSeq->POLYMORPHIC_SITES->addElementHigh((void*)contigPos);
  pCONSENSUS_SEQ_ARRAY->addElementHigh(newConSeq);
  return TRUE;
}


void AlignedCluster::outputPrimerBoulderInput(ostream& output)
//
// Effects:  Print the data needed by PRIMER to pick PCR primers
//           that flank the potentially polymorphic nucleotides
//           identified.  The format used is Boulder IO, which is
//           accepted by PRIMER.
//
{
  if (!pCONSENSUS_SEQ_ARRAY)
    return;

  int seqArraySize = pCONSENSUS_SEQ_ARRAY->size();
  if (!seqArraySize)
    return;

  ////////////////////////////
  gNumClusterWithSNPConsensus++;
  ////////////////////////////

  int minUpPrimerWinSize = pPARAMS->min_length_upstream_of_SNP
    + pPARAMS->min_primer_length;
  int minDownPrimerWinSize = pPARAMS->min_length_downstream_of_SNP
    + pPARAMS->min_primer_length;

  for (int i=0; i<seqArraySize; i++) {
    ConsensusSequence* conSeqInfo = (ConsensusSequence*)pCONSENSUS_SEQ_ARRAY-
>element(i);
    Sequence* conSeq = conSeqInfo->CONSENSUS_SEQ;
    Array* polySites = conSeqInfo->POLYMORPHIC_SITES;
    int conSeqLength = conSeq->length();

//    if (conSeqLength <= pPARAMS->max_pcr_product_length) {
    if (TRUE) {

      output << "MARKER_NAME=" << pCLUSTER_NAME;
      output << "\n";
      output << "SEQUENCE=";
      conSeq->printSequence(FALSE, output);
      output << "\n";
```

```
      for (int j=0; j<polySites->size(); j++)
       output << "POLYMORPHISM" << "_"
         << ((int)polySites->element(j) - conSeqInfo->FRONT_INDEX_IN_CONTIG)
           << "=" << pCONTIG_INFO[(int)polySites->element(j)].POLYMORPHISM
             << "\n";

      output << "TARGET=" // TEMPORARY
       << ((int)polySites->element(0) - conSeqInfo->FRONT_INDEX_IN_CONTIG-10)
         << "," << ((int)polySites->element(polySites->size()-1) -
                    (int)polySites->element(0) + 1 + 20) << ","
                      << "TARGET_REGION\n";

      output << "PRIMER_DEFAULT_PRODUCT=\"350-400 300-350 250-300 200-250 150-200 100-
150 64-100\"\n";

      output << "=\n";
    }
  }
}
```

The Single Nucleotide Polymorphism Detector (SNPD) / Parameter.H

```
#ifndef PARAMETER_H

#include "project.H"
#include <iostream.h>
#define PARAMETER_H


class Parameter {

// This is simply structure containing various parameter variables.  Putting
// all these variable in one structure allows the parameters to be passed
// around more easily.

public:

  Parameter(void) {
    // By default, do not print separate output files.
    no_file_conflict = FALSE;
    no_file_polymorphism = FALSE;
    no_file_analysis = FALSE;
    no_file_all = FALSE;

    // By default, print the window of bases surrounding the polymorphic
    // sites, if polymorphism file is printed.  The window is centered
    // on the polymorphic site, and the width of the window to print is
    // <print_qual_window_size> * 2 + 1.
    print_qual_window = TRUE;
    print_qual_window_size = 20;

    // By default, consider only high-quality bases in polymorphism
    // analysis.  If <use_high_qual_only> is set to FALSE, all bases
    // will be considered when scanning for potential polymorphic sites
    // and generating consensus sequences.
    use_high_qual_only = TRUE;

    // By default, must have 2 high-quality reads per polymorphic nucleotide
    // per polymorphic site.  Minimum allelic frequence of 20% for the minor
    // allele of polymorphic nucleotide.
    min_qual_reads = 2;
    min_minor_allele_freq = 20.0;

    // Check the bases surrounding the potential polymorphic site for
    // ambiguous bases.  The window to check is centered on the polymorphic
    // site and has a width of <qual_window_size> * 2 + 1.  If the percentage
    // of ambiguous bases exceeds <qual_window_threshold>, reject this site.
    qual_window_size = 5;
    qual_window_threshold = 10.0;

    // Each base in the consensus sequence which contains the polymorphic
    // site must be covered by <min_consensus_overlap> reads.  The length
    // of the consensus sequence must be <min_consensus_length> bases.  This
    // is necessary for primer picking.
```

```
    min_consensus_overlap = 3;
    min_consensus_length = 80;

    // These parameters are here for primer picking purposes.  The forward
    // primer must not be picked in the <min_length_upstream_of_SNP>-base
    // region upstream of a polymorphic site, and the reverse primer must
    // not be picked in the <min_length_downstream_of_SNP>-base region
    // downstream of a polymorphic site.  In addition, there must be an
    // unambiguous region of <min_primer_length> bases upstream and down-
    // stream of the polymorphic site where primers can potentially be
    // picked.
    min_primer_length = 20;
    min_margin_upstream = 10;
    min_margin_downstream = 10;

    // This parameter is currently not being used in Release 1.0.
    max_pcr_product_length = 400;
};


  ~Parameter(void) { };


  void parseParameters(int argc, char *argv[]);
  // Effects:  Parse the command line arguments that trail the executed
  //           command, and set various parameters to their user-specified
  //           values.

  void printParameters(ostream& output = cerr);
  // Effects:  Prints the values of all parameters to <output> in a
  //           human-readable fashion.

  // Declaration of all parameters
  bool no_file_conflict;
  bool no_file_polymorphism;
  bool no_file_analysis;
  bool no_file_all;

  bool print_qual_window;
  int print_qual_window_size;

  bool use_high_qual_only;

  int min_qual_reads;
  float min_minor_allele_freq;
  int qual_window_size;
  float qual_window_threshold;

  int min_consensus_overlap;
  int min_consensus_length;

  int max_pcr_product_length;
  int min_primer_length;
  int min_margin_upstream;
  int min_margin_downstream;
};

#endif
```

The Single Nucleotide Polymorphism Detector (SNPD) / Parameter.C

```
#include "parameter.H"
#include <iostream.h>
#include <stdio.h>


void Parameter::parseParameters(int argc, char *argv[])
//
// Effects:  Parse the command line arguments that trail the executed
//           command, and set various parameters to their user-specified
//           values.
//
{
  // Examine each argument one by one.  If an argument requires the user
  // to specify a value, assume the argument that follows immediately is
  // the value.
  for (int i=1; i < argc; i++) {
    if (!strcmp(argv[i], "-no_file_conflict"))
      no_file_conflict = TRUE;
    else if (!strcmp(argv[i], "-no_file_polymorphism"))
      no_file_polymorphism = TRUE;
    else if (!strcmp(argv[i], "-no_file_analysis"))
      no_file_analysis = TRUE;
    else if (!strcmp(argv[i], "-no_file_all"))
      no_file_all = TRUE;
    else if (!strcmp(argv[i], "-no_files")) {
      no_file_conflict = TRUE;
      no_file_polymorphism = TRUE;
      no_file_analysis = TRUE;
      no_file_all = TRUE;
    }

    else if (!strcmp(argv[i], "-not_print_qual_window"))
      print_qual_window = FALSE;
    else if (!strcmp(argv[i], "-print_qual_window_size"))
      sscanf(argv[++i], "%d", &print_qual_window_size);

    else if (!strcmp(argv[i], "-ignore_qual"))
      use_high_qual_only = FALSE;

    else if (!strcmp(argv[i], "-min_qual_reads"))
      sscanf(argv[++i], "%d", &min_qual_reads);
    else if (!strcmp(argv[i], "-min_minor_allele_freq"))
      sscanf(argv[++i], "%f", &min_minor_allele_freq);

    else if (!strcmp(argv[i], "-qual_window_size"))
      sscanf(argv[++i], "%d", &qual_window_size);
    else if (!strcmp(argv[i], "-qual_window_threshold"))
      sscanf(argv[++i], "%f", &qual_window_threshold);

    else if (!strcmp(argv[i], "-min_consensus_overlap"))
      sscanf(argv[++i], "%d", &min_consensus_overlap);
    else if (!strcmp(argv[i], "-min_consensus_length"))
      sscanf(argv[++i], "%d", &min_consensus_length);
```

```
    else if (!strcmp(argv[i], "-max_pcr_product_length"))
      sscanf(argv[++i], "%d", &max_pcr_product_length);
    else if (!strcmp(argv[i], "-min_primer_length"))
      sscanf(argv[++i], "%d", &min_primer_length);
    else if (!strcmp(argv[i], "-min_margin_upstream"))
      sscanf(argv[++i], "%d", &min_margin_upstream);
    else if (!strcmp(argv[i], "-min_margin_downstream"))
      sscanf(argv[++i], "%d", &min_margin_downstream);

    else if (argv[i][0] == '-') {
      cerr << "ERROR:  option " << argv[i] << " not recognized\n";
      exit(1);
    }
    else {
      // Cannot recognize the argument.  Print error message and
      // terminate program.
      cerr << "ERROR:  commandline argument " << argv[i]
       << " not recognized\n";
      exit(1);
    }
  }
}


void Parameter::printParameters(ostream& output)
//
// Effects:  Prints the values of all parameters to <output> in a
//           human-readable fashion.
//
{
  output.precision(4);

  output << "Print conflict file:                          ";
  if (no_file_conflict) output << "NO\n"; else output << "YES\n";

  output << "Print polymorphism file:                      ";
  if (no_file_polymorphism) output << "NO\n"; else output << "YES\n";

  output << "Print analysis file:                          ";
  if (no_file_analysis) output << "NO\n"; else output << "YES\n";

  output << "Print complete sequence file:                 ";
  if (no_file_all) output << "NO\n"; else output << "YES\n";

  output << "Consider high quality bases only:             ";
  if (!use_high_qual_only) output << "NO\n"; else output << "YES\n";

  output << "Minimum number of quality reads per allele:   "
    << min_qual_reads << "\n";

  output << "Minimum minor allele frequency:               "
    << min_minor_allele_freq << "\n";

  output << "Size of window for quality control:           "
    << qual_window_size << "\n";

  output << "Quality control window threshold:             "
    << qual_window_threshold << "\n";

  output << "Minimum consensus overlap:                    "
    << min_consensus_overlap << "\n";

  output << "Minimum consensus length:                     "
```

```
    << min_consensus_length << "\n";

  output << "Maximum length of PCR product:                  "
    << max_pcr_product_length << "\n";

  output << "Minimum primer length:                          "
    << min_primer_length << "\n";

  output << "Minimum number of bases upstream of the SNP:    "
    << min_margin_upstream << "\n";

  output << "Minimum number of bases downstream of the SNP:  "
    << min_margin_downstream << "\n";

  output << "Size of quality window to print:                "
    << print_qual_window_size << "\n";

  output << "\n";
}
```

The Single Nucleotide Polymorphism Detector (SNPD) / Project.H

```
#ifndef PROJECT_H

#include <bool.h>
#define PROJECT_H


// Debug mode toggle
#define DEBUG_MODE TRUE


// Define some constants
#define WRITE_MODE 2


// Define various delimiters to be read from standard input.

// Delimiters for contig sequence
#define CONTIG_LEFT_DELIMITER '{'
#define CONTIG_RIGHT_DELIMITER '}'

// Delimiters for fragment sequence to be aligned to contig sequence
#define FRAGMENT_LEFT_DELIMITER '['
#define FRAGMENT_RIGHT_DELIMITER ']'

// Delimiters for the front fragment index
#define FRONT_INDEX_LEFT_DELIMITER '('
#define FRONT_INDEX_RIGHT_DELIMITER ')'

// Delimiters for the back fragment index
#define BACK_INDEX_LEFT_DELIMITER '<'
#define BACK_INDEX_RIGHT_DELIMITER '>'

// Delimiters for the cluster/contig name
#define CLUSTER_NAME_LEFT_DELIMITER ':'
#define CLUSTER_NAME_RIGHT_DELIMITER ';'

// Delimiters for the fragment name
#define FRAGMENT_NAME_LEFT_DELIMITER 47
#define FRAGMENT_NAME_RIGHT_DELIMITER 92

// Delimiter to signal that all the fragments have been received
#define CLUSTER_END 0


// Some constants for signal purpose used by the "deterministic finite
// automatom" which parses the data received from standard input and
// the main loop.
#define DEFAULT 0
#define CONTIG_RECEIVED 1
#define FRAGMENT_RECEIVED 2
#define CLUSTER_END_RECEIVED 3
```

```
// Define some suffixes to be attached to output files
#define SUFFIX_CONFLICT_FILE ".conf"
#define SUFFIX_POLYMORPHISM_FILE ".poly"
#define SUFFIX_ANALYSIS_FILE ".analy"
#define SUFFIX_ALL_FILE ".all"

#endif
```

The Single Nucleotide Polymorphism Detector (SNPD) / Project.C

```
#include "project.H"
#include "analysis.H"
#include "parameter.H"
#include "sequence.H"
#include "utility.H"
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <string.h>


// Prototyping for several procedures

int ReadInput(istream& input, Sequence** contigSeq, Sequence** fragSeq);
void PrintOutfiles(AlignedCluster* alignedCluster, char* filename);
bool IsBasePair(char c);
void Test(void);


// Declaration and Initialization of global variables

Parameter* gParameters;
Sequence* gContig = 0;
Sequence* gFragment = 0;
AlignedCluster* gAlignedCluster = 0;

char gClusterName[64];
char gFragName[64];
int gFrontIndex = 0;
int gBackIndex = 0;

int gNumClusterProcessed = 0;
int gNumSNPs = 0;
int gNumClusterWithSNP = 0;
int gNumClusterWithSNPConsensus = 0;


// Main program

main(int argc, char *argv[])
//
// Effects:  Continues to receive sequences and alignment data via
//           standard input.  Analyze each cluster as it is received.
//           Output potential single nucleotide polymorphism data and
//           other information necessary for primer picking by PRIMER
//           in Boulder IO format.
//
{
  // Create a new Parameter object.
  gParameters = new Parameter;

  // Parse commandline arguments.
```

```
gParameters->parseParameters(argc, argv);

// Print settings to standard error.
gParameters->printParameters(cerr);
cerr << "\n\n";

// Loop for reading data from standard input.  Read until there is
// no more data to be read.

while (!cin.eof() && !cin.fail()) {
  // Parse the input using the ReadInput procedure which implements
  // a deterministic finite automaton
  int command = ReadInput(cin, &gContig, &gFragment);

  // Execute command according to command return by ReadInput.
  switch (command) {

  case DEFAULT:
    break;

  case CONTIG_RECEIVED:
    // Received a contig sequence.  Create a new empty sequence assembly.
    // Delete old assembly if it exists.
    if (gAlignedCluster)
     delete gAlignedCluster;
    gAlignedCluster = new AlignedCluster(gParameters, gContig, gClusterName);
    break;

  case FRAGMENT_RECEIVED:
    // Received a sequence.  Add the sequence to the sequence alignment
    // assembly.
    if ((gContig->length() > 0) && (gFragment->length() > 0)) {
     gAlignedCluster->addFragment(gFragment, gFragName,
                                  gFrontIndex-1, gBackIndex-1);
    }
    break;

  case CLUSTER_END_RECEIVED:
    // Have received all the sequences.  Start analysis.
    cerr << "JAMALAH processing cluster <" << gClusterName << ">...\n";
    cerr << "LENGTH OF CONTIG is " << gContig->length() << " bases\n";
    cerr << "CLUSTER contains "
     << gAlignedCluster->clusterSize() << " sequences\n";

    // Check to make sure that the number of sequences received is
    // greater than 0, and that the length of the contig sequence is
    // also greater than 0.  Otherwise, it's meaningless to perform
    // the analysis.
    if ((gContig->length() > 0) && (gAlignedCluster->clusterSize() > 0)) {
     gAlignedCluster->analyzeAllPositions(gParameters->use_high_qual_only);

     // Analyze all positions.
     gAlignedCluster->printPolymorphicPositions(gParameters->use_high_qual_only, 0);

     // Output result of analysis in BoulderIO format to standard output.
     gAlignedCluster->outputPrimerBoulderInput(cout);

     // Print analysis files.
     PrintOutfiles(gAlignedCluster, gClusterName);
    }
    cerr << "Done.\n\n";

    // Keep track of number of clusters processed.
    gNumClusterProcessed++;
```

```
      break;

    default:
      break;
    }
  }

  // Print summary of analysis in this session.
  ofstream outfile("SUMMARY", WRITE_MODE);
  outfile << "Number of clusters:                                          "
    << gNumClusterProcessed << "\n";
  outfile << "Number of SNPs:                                              "
    << gNumSNPs << "\n";
  outfile << "Number of clusters with SNPs:                                "
    << gNumClusterWithSNP << "\n";
  outfile << "Number of clusters with SNPs and ideal consensus sequences:  "
    << gNumClusterWithSNPConsensus << "\n";
  outfile.close();
}


void PrintOutfiles(AlignedCluster* alignedCluster, char* filename)
//
// Effects:  Print analysis files of the results from the single
//           nucleotide polymorphism analysis.  Use <filename> as the
//           filename and attach proper suffixes to it.
//
{
  // Determine the length of <filename> + length of ".".
  char* newFilename;
  int filenameLen = strlen(filename) + 1;

  if (!gParameters->no_file_conflict) {
    // Create a new char array for <filename> + the suffix.
    newFilename = new char[filenameLen + strlen(SUFFIX_CONFLICT_FILE)];

    // Copy to the array <filename> appended by the proper suffix.
    strcpy(newFilename, filename);
    strcat(newFilename, SUFFIX_CONFLICT_FILE);

    // Create a new file for writing.
    ofstream outfile(newFilename, WRITE_MODE);

    // Print settings.
    gParameters->printParameters(outfile);
    outfile << "\n\n";

    // Print information on all conflicting nucleotides.
    alignedCluster->printConflictPositions(gParameters->use_high_qual_only,
                                           outfile);
    outfile.close();
  }

  if (!gParameters->no_file_polymorphism) {
    // Create a new char array for <filename> + the suffix.
    newFilename = new char[filenameLen + strlen(SUFFIX_POLYMORPHISM_FILE)];

    // Copy to the array <filename> appended by the proper suffix.
    strcpy(newFilename, filename);
    strcat(newFilename, SUFFIX_POLYMORPHISM_FILE);

    // Create a new file for writing.
    ofstream outfile(newFilename, WRITE_MODE);
```

```
    // Print settings.
    gParameters->printParameters(outfile);
    outfile << "\n\n";

    // Print information on all polymorphic nucleotides.
    alignedCluster->printPolymorphicPositions(gParameters->use_high_qual_only,
                                              outfile);
    outfile.close();
  }

  if (!gParameters->no_file_all) {
    // Create a new char array for <filename> + the suffix.
    newFilename = new char[filenameLen + strlen(SUFFIX_ALL_FILE)];

    // Copy to the array <filename> appended by the proper suffix.
    strcpy(newFilename, filename);
    strcat(newFilename, SUFFIX_ALL_FILE);

    // Create a new file for writing.
    ofstream outfile(newFilename, WRITE_MODE);

    // Print settings.
    gParameters->printParameters(outfile);
    outfile << "\n\n";

    // Print information on all nucleotides.
    alignedCluster->printAllPositions(outfile);
    outfile.close();
  }
}


int ReadInput(istream& input, Sequence** contigSeq, Sequence** fragSeq)
//
// Effects:  Parse the standard input by looking for specific delimiter
//           pairs <left_delimiter> and <right_delimiter>.  Each kind of
//           data--contig sequence, fragment seq, etc.--is enclosed inside
//           a pair of uniquely defined delimiters.  If
//
{
  static char prevDelimiter = 0;
  static Sequence* selectedSeq = 0;

  static char string[64];
  static char* stringPtr = string;

  char c = input.get();
  switch (c) {
  case CLUSTER_NAME_LEFT_DELIMITER:
    if ((prevDelimiter == CONTIG_LEFT_DELIMITER) ||
        (prevDelimiter == FRAGMENT_LEFT_DELIMITER) ||
        (prevDelimiter == FRONT_INDEX_LEFT_DELIMITER) ||
        (prevDelimiter == BACK_INDEX_LEFT_DELIMITER) ||
        (prevDelimiter == FRAGMENT_NAME_LEFT_DELIMITER)) {
      exit(1);
    }
    prevDelimiter = CLUSTER_NAME_LEFT_DELIMITER;
    return DEFAULT;
    break;
  case CLUSTER_NAME_RIGHT_DELIMITER:
    if ((prevDelimiter != CLUSTER_NAME_LEFT_DELIMITER) && prevDelimiter) {
      exit(1);
    }
    *stringPtr = 0;
```

```
  strcpy(gClusterName, string);
  stringPtr = string;
  prevDelimiter = CLUSTER_NAME_RIGHT_DELIMITER;
  return DEFAULT;
  break;
case CONTIG_LEFT_DELIMITER:
  if ((prevDelimiter == CONTIG_LEFT_DELIMITER) ||
      (prevDelimiter == FRAGMENT_LEFT_DELIMITER) ||
      (prevDelimiter == FRONT_INDEX_LEFT_DELIMITER) ||
      (prevDelimiter == BACK_INDEX_LEFT_DELIMITER) ||
      (prevDelimiter == FRAGMENT_NAME_LEFT_DELIMITER)) {
    exit(1);
  }
  *contigSeq = selectedSeq = new Sequence;
  prevDelimiter = CONTIG_LEFT_DELIMITER;
  return DEFAULT;
  break;
case CONTIG_RIGHT_DELIMITER:
  if ((prevDelimiter != CONTIG_LEFT_DELIMITER) && prevDelimiter) {
    exit(1);
  }
  prevDelimiter = CONTIG_RIGHT_DELIMITER;
  return CONTIG_RECEIVED;
  break;
case FRAGMENT_LEFT_DELIMITER:
  if ((prevDelimiter == CONTIG_LEFT_DELIMITER) ||
      (prevDelimiter == FRAGMENT_LEFT_DELIMITER) ||
      (prevDelimiter == FRONT_INDEX_LEFT_DELIMITER) ||
      (prevDelimiter == BACK_INDEX_LEFT_DELIMITER) ||
      (prevDelimiter == FRAGMENT_NAME_LEFT_DELIMITER)) {
    exit(1);
  }
  *fragSeq = selectedSeq = new Sequence;
  prevDelimiter = FRAGMENT_LEFT_DELIMITER;
  return DEFAULT;
  break;
case FRAGMENT_RIGHT_DELIMITER:
  if ((prevDelimiter != FRAGMENT_LEFT_DELIMITER) && prevDelimiter) {
    exit(1);
  }
  prevDelimiter = FRAGMENT_RIGHT_DELIMITER;
  return FRAGMENT_RECEIVED;
  break;
case FRONT_INDEX_LEFT_DELIMITER:
  if ((prevDelimiter == CONTIG_LEFT_DELIMITER) ||
      (prevDelimiter == FRAGMENT_LEFT_DELIMITER) ||
      (prevDelimiter == FRONT_INDEX_LEFT_DELIMITER) ||
      (prevDelimiter == BACK_INDEX_LEFT_DELIMITER) ||
      (prevDelimiter == FRAGMENT_NAME_LEFT_DELIMITER)) {
    exit(1);
  }
  prevDelimiter = FRONT_INDEX_LEFT_DELIMITER;
  return DEFAULT;
  break;
case FRONT_INDEX_RIGHT_DELIMITER:
  if ((prevDelimiter != FRONT_INDEX_LEFT_DELIMITER) && prevDelimiter) {
    exit(1);
  }
  *stringPtr = 0;
  gFrontIndex = atoi(string);
  stringPtr = string;
  prevDelimiter = FRONT_INDEX_RIGHT_DELIMITER;
  return DEFAULT;
  break;
```

```
  case BACK_INDEX_LEFT_DELIMITER:
    if ((prevDelimiter == CONTIG_LEFT_DELIMITER) ||
        (prevDelimiter == FRAGMENT_LEFT_DELIMITER) ||
        (prevDelimiter == FRONT_INDEX_LEFT_DELIMITER) ||
        (prevDelimiter == BACK_INDEX_LEFT_DELIMITER) ||
        (prevDelimiter == FRAGMENT_NAME_LEFT_DELIMITER)) {
      exit(1);
    }
    prevDelimiter = BACK_INDEX_LEFT_DELIMITER;
    return DEFAULT;
    break;
  case BACK_INDEX_RIGHT_DELIMITER:
    if ((prevDelimiter != BACK_INDEX_LEFT_DELIMITER) && prevDelimiter) {
      exit(1);
    }
    *stringPtr = 0;
    gBackIndex = atoi(string);
    stringPtr = string;
    prevDelimiter = BACK_INDEX_RIGHT_DELIMITER;
    return DEFAULT;
    break;
  case FRAGMENT_NAME_LEFT_DELIMITER:
    if ((prevDelimiter == CONTIG_LEFT_DELIMITER) ||
        (prevDelimiter == FRAGMENT_LEFT_DELIMITER) ||
        (prevDelimiter == FRONT_INDEX_LEFT_DELIMITER) ||
        (prevDelimiter == BACK_INDEX_LEFT_DELIMITER) ||
        (prevDelimiter == FRAGMENT_NAME_LEFT_DELIMITER)) {
      exit(1);
    }
    prevDelimiter = FRAGMENT_NAME_LEFT_DELIMITER;
    return DEFAULT;
    break;
  case FRAGMENT_NAME_RIGHT_DELIMITER:
    if ((prevDelimiter != FRAGMENT_NAME_LEFT_DELIMITER) && prevDelimiter) {
      exit(1);
    }
    *stringPtr = 0;
    strcpy(gFragName, string);
    stringPtr = string;
    prevDelimiter = FRAGMENT_NAME_RIGHT_DELIMITER;
    return DEFAULT;
    break;
  case CLUSTER_END:
      return CLUSTER_END_RECEIVED;
    break;
  default:
    if (((prevDelimiter == CONTIG_LEFT_DELIMITER) ||
         (prevDelimiter == FRAGMENT_LEFT_DELIMITER)) && IsBasePair(c)) {
      selectedSeq->putChar(c);
    }
    else if ((prevDelimiter == FRONT_INDEX_LEFT_DELIMITER) ||
             (prevDelimiter == BACK_INDEX_LEFT_DELIMITER) ||
             (prevDelimiter == FRAGMENT_NAME_LEFT_DELIMITER) ||
             (prevDelimiter == CLUSTER_NAME_LEFT_DELIMITER)) {
      *(stringPtr++) = c;
    }
    return DEFAULT;
    break;
  }
}


bool IsBasePair(char c)
//
```

```
// Effects:  Return TRUE if character <c> is one of the following:
//           G/g - nucleotide guanine
//           A/a - nucleotide adenine
//           T/t - nucleotide thymine
//          C/c - nucleotide cytosine
//          N/n - no call
//          * - padding character
//
//          Otherwise, return FALSE.
//
{
  if ((c == 'G') || (c == 'A') || (c == 'T') || (c == 'C') || (c == 'N') ||
      (c == 'g') || (c == 'a') || (c == 't') || (c == 'c') || (c == 'n') ||
      (c == '*'))
    return TRUE;

  return FALSE;
}
```

The Single Nucleotide Polymorphism Detector (SNPD) / Sequence.H

```
#ifndef SEQUENCE_H

#include "project.H"
#include <iostream.h>
#define SEQUENCE_H



class Sequence {

// The Sequence class is similar to a typical string class.  Each object is
// capable of storing an array of characters.  The class provides a variety
// of methods by which characters can be added to the growing sequence and
// by which the exisiting sequence can be accessed.

public:

  Sequence(void);
  // Effects:  Constructor for the class.  Creates a new empty sequence.


  Sequence(char* seq);
  // Effects:  Constructor for the class.  Creates a new sequence containing
  //           the characters in string <seq>.  The string <seq> must be
  //           nil-terminated.


  Sequence(char* seq, const long len);
  // Effects:  Constructor for the class.  Creates a new sequence containing
  //           the characters in string <seq>.  Up to <len> characters will
  //           be copied from <seq> to the new sequence.


  Sequence(Sequence& srcSeq);
  // Effects:  Copy constructor for the class.  An exact copy of the <srcSeq>
  //           will be created.


  ~Sequence(void);
  // Effects:  Destructor for the class.


  long length(void) { return pSEQUENCE_LEN; };
  // Effects:  Return the length of the sequence.


  long setReadCursor(const long index);
  // Effects:  Set the read cursor to position <index> in the sequence.
  //           Characters in the sequence are indexed starting at 0.  Next
  //           call with Sequence::getChar will return the character at
  //           position <index>.  If <index> is lower than 0, then set the
  //           read cursor to 0.  If <index> is greater than length of
  //           sequence-1, set the read cursor to point to the last
```

```
  //            character in the sequence.  Return the position the read
  //            cursor is set to.


  char getChar(void) { return pSEQUENCE[pREAD_CURSOR++]; };
  // Effects:  Return the character currently being pointed by the read
  //           cursor.  Advance the read cursor by 1.


  char getChar(long index) {
    if ((index >= pSEQUENCE_LEN) || (index < 0)) return 0;
    return pSEQUENCE[index];
  };
  // Effects:  Return the character at position <index>.


  char* getSequence(void);
  // Effects:  Return the complete sequence in a newly allocated string.


  long putChar(const char c);
  // Effects:  Add the character <c> to the position currently being pointed
  //           by the write cursor.  Return the length of updated sequence.


  long putString(const char* str, long len = 0);
  // Effects:  Append the string of characters in <str> to the existing
  //           sequence.  Return the length of updated sequence.


  long putSequence(const Sequence* seq, long startIndex = 0, long len = 0);
  // Effects:  Append the string of <len> characters in <seq> starting at
  //           position <startIndex> to the existing sequence.  Return the
  //           length of updated sequence.


  void printSequence(char CR = TRUE, ostream& output = cout);
  // Effects:  Print the complete sequence to <output>.  Terminate the
  //           printing with a newline character if <CR> is TRUE.


  char operator==(const Sequence& seq2);
  // Effects:  Return TRUE if <seq1> and <seq2> contain exactly the same
  //           sequences.  Return FALSE otherwise.


  char operator!=(const Sequence& seq2) { return !(*this == seq2); };
  // Effects:  Return TRUE if <seq1> and <seq2> contain different sequences.
  //           Return FALSE otherwise.

protected:

private:

  // Pointer to the array of characters.  If NIL, the sequence is empty.
  char* pSEQUENCE;

  // Variable to keep track of the length of character array allocated.
  long pALLOC_LEN;

  // Variable to keep track of the length of sequence.
  long pSEQUENCE_LEN;

  // Variable to keep track of the current read position.
```

```
  long pREAD_CURSOR;

  // Variable to keep track of the current write position.
  long pWRITE_CURSOR;

};


ostream& operator<<(ostream& output, Sequence& seq);
// Effects:  Print the complete sequence to <output>.  Printing not terminated
//           with a newline character.

#endif
```

The Single Nucleotide Polymorphism Detector (SNPD) / Sequence.C

```
#include "sequence.H"
#include <string.h>

#define ALLOC_BLOCK_SIZE 64

Sequence::Sequence(void)
//
// Effects:  Constructor for the class.  Creates a new empty sequence.
//
{
  pSEQUENCE = new char[ALLOC_BLOCK_SIZE];
  pALLOC_LEN = ALLOC_BLOCK_SIZE;
  pSEQUENCE_LEN = pREAD_CURSOR = pWRITE_CURSOR = 0;
}


Sequence::Sequence(char* seq) : pREAD_CURSOR(0)
//
// Effects:  Constructor for the class.  Creates a new sequence containing
//           the characters in string <seq>.  The string <seq> must be
//           nil-terminated.
//
{
  pSEQUENCE_LEN = 0;
  if (seq) {

    // Adjust write cursor according to the length of <str>.
    pWRITE_CURSOR = pSEQUENCE_LEN = strlen(seq);

    // Calculate size of char array to allocate.
    pALLOC_LEN = pSEQUENCE_LEN -
      (pSEQUENCE_LEN % ALLOC_BLOCK_SIZE) + ALLOC_BLOCK_SIZE;
    pSEQUENCE = new char[pALLOC_LEN];

    // Copy <seq> to newly allocated character array.
    for (long i=0; i<pSEQUENCE_LEN && *seq; i++, seq++)
      pSEQUENCE[i] = *seq;
  }
}


Sequence::Sequence(char* seq, const long len) : pREAD_CURSOR(0)
//
// Effects:  Constructor for the class.  Creates a new sequence containing
//           the characters in string <seq>.  Up to <len> characters will
//           be copied from <seq> to the new sequence.
//
{
  if (seq && len) {
    // Calculate the size char array to allocate.
    pALLOC_LEN = len - (len % ALLOC_BLOCK_SIZE) + ALLOC_BLOCK_SIZE;
    pSEQUENCE = new char[pALLOC_LEN];
```

```
    // Copy <len> characters of <seq> to newly allocated character array,
    // if possible.  Otherwise, copy as much as possible.
    for (pSEQUENCE_LEN=0; pSEQUENCE_LEN<len && *seq; pSEQUENCE_LEN++, seq++)
      pSEQUENCE[pSEQUENCE_LEN] = *seq;

    // Adjust write cursor according to the amount copied.
    pWRITE_CURSOR = pSEQUENCE_LEN;
  }
}


Sequence::Sequence(Sequence& srcSeq)
//
// Effects:  Copy constructor for the class.  An exact copy of the <srcSeq>
//           will be created.
//
{
  // If this sequence already contains some sequence.  Delete the sequence.
  if (pSEQUENCE) delete [] pSEQUENCE;

  // Allocate a new char array for the new sequence to be copied.
  pSEQUENCE = new char[pALLOC_LEN = srcSeq.pALLOC_LEN];

  // Copy the sequence.
  memcpy(pSEQUENCE, srcSeq.pSEQUENCE,
         (pSEQUENCE_LEN = pSEQUENCE_LEN) * sizeof(char));

  // Update the sequence cursors.
  pREAD_CURSOR = srcSeq.pREAD_CURSOR;
  pWRITE_CURSOR = srcSeq.pWRITE_CURSOR;
}


Sequence::~Sequence(void)
//
// Effects:  Destructor for the class.
//
{
  // Delete the char array.
  delete [] pSEQUENCE;
}


void Sequence::printSequence(char CR, ostream& output)
//
// Effects:  Print the complete sequence to <output>.  Terminate the
//           printing with a newline character if <CR> is TRUE.
//
{

  for (long i=0; i<pSEQUENCE_LEN; i++)
    output << pSEQUENCE[i];
  if (CR)
    output << "\n";
}


long Sequence::setReadCursor(const long index)
//
// Effects:  Set the read cursor to position <index> in the sequence.
//           Characters in the sequence are indexed starting at 0.  Next
//           call with Sequence::getChar will return the character at
//           position <index>.  If <index> is lower than 0, then set the
//           read cursor to 0.  If <index> is greater than length of
```

```
//            sequence-1, set the read cursor to point to the last
//            character in the sequence.  Return the position the read
//            cursor is set to.
//
{
  pREAD_CURSOR = index;

  // Check index against the sequence's upper and lower bounds.
  if (index < 0)
    pREAD_CURSOR = 0;
  else if (index > pSEQUENCE_LEN)
    pREAD_CURSOR = pSEQUENCE_LEN - 1;

  return pREAD_CURSOR;
}


char* Sequence::getSequence(void)
//
// Effects:  Return the complete sequence in a newly allocated string.
//
{
  // Allocate a char array.
  char* new_alloc = new char[pSEQUENCE_LEN + 1];

  // Copy sequence to the array.
  for (long i=0; i<pSEQUENCE_LEN; i++)
    new_alloc[i] = pSEQUENCE[i];
  new_alloc[pSEQUENCE_LEN] = 0;

  // Return the location of the char array.
  return new_alloc;
}


long Sequence::putChar(const char c)
//
// Effects:  Add the character <c> to the position currently being pointed
//           by the write cursor.  Return the length of updated sequence.
//
{
  // Check to see if another character can be added to the allocated
  // char array.  If not, allocate a bigger array, and copy the old
  // stuf to the new array.  Delete the old char array.
  if (pWRITE_CURSOR >= pALLOC_LEN) {
    char* new_alloc = new char[pALLOC_LEN + ALLOC_BLOCK_SIZE];
    pALLOC_LEN += ALLOC_BLOCK_SIZE;
    memcpy(new_alloc, pSEQUENCE, pALLOC_LEN);
    delete [] pSEQUENCE;
    pSEQUENCE = new_alloc;
  }

  // Insert the character <c>.
  pSEQUENCE[pWRITE_CURSOR++] = c;
  pSEQUENCE_LEN++;

  return pSEQUENCE_LEN;
}


long Sequence::putString(const char* str, long len)
//
// Effects:  Append the string of characters in <str> to the existing
//           sequence.  Return the length of updated sequence.
```

```
//
{
  // Check length of <str>
  if (!len)
    return pSEQUENCE_LEN;

  // Check see if <len> characters in <str> can be added to the allocated
  // char array.  If not allocate one big enough, and copy the old stuff
  // to the new arra.  Delete the old array.  Update variables, if
  // necessary.
  if ((pWRITE_CURSOR + len) > pALLOC_LEN) {
    long alloc_size = pALLOC_LEN + len -
      (len % ALLOC_BLOCK_SIZE) + ALLOC_BLOCK_SIZE;
    char* new_alloc = new char[alloc_size];
    pALLOC_LEN = alloc_size;
    memcpy(new_alloc, pSEQUENCE, pSEQUENCE_LEN);
    delete [] pSEQUENCE;
    pSEQUENCE = new_alloc;
  }

  // Copy <len> characters of <str> to the sequence character array.
  // Update variables.
  memcpy(&pSEQUENCE[pWRITE_CURSOR], str, len);
  pWRITE_CURSOR += len;
  pSEQUENCE_LEN += len;

  // Return length of new sequence.
  return pSEQUENCE_LEN;
}


long Sequence::putSequence(const Sequence* seq, long startIndex, long len)
//
// Effects:  Append the string of <len> characters in <seq> starting at
//           position <startIndex> to the existing sequence.  Return the
//           length of updated sequence.
{
  // Check to make sure <startIndex> is within bound.
  if (startIndex >= seq->pSEQUENCE_LEN)
    return 0;
  else if (startIndex < 0)
    startIndex = 0;

  // Get length of <src_seq> and check length.
  char* src_seq = &seq->pSEQUENCE[startIndex];
  if (!len)
    len = seq->pSEQUENCE_LEN - startIndex;
  else if (len > (seq->pSEQUENCE_LEN - startIndex))
    len = seq->pSEQUENCE_LEN - startIndex;

  // Call putString to actually do the copying.
  return putString(src_seq, len);
}


char Sequence::operator==(const Sequence& seq2)
//
// Effects:  Return TRUE if <seq1> and <seq2> contain exactly the same
//           sequences.  Return FALSE otherwise.
//
{
  // Compare the length of the two sequences.  If not equal, return
  // FALSE.
  if (this->pSEQUENCE_LEN != seq2.pSEQUENCE_LEN)
```

```
    return FALSE;

  // Compare the sequences.  If they are different, return FALSE.
  for (long i=0; i<this->pSEQUENCE_LEN; i++)
    if (this->pSEQUENCE[i] != seq2.pSEQUENCE[i])
      return false;

  return TRUE;
}


ostream& operator<<(ostream& output, Sequence& seq)
//
// Effects:  Print the complete sequence to <output>.  Printing not terminated
//           with a newline character.
//
{
  seq.printSequence(FALSE, output);
  return output;
}
```

The Single Nucleotide Polymorphism Detector (SNPD) / Utility.H

```
#ifndef UTILITY_H

#include <iostream.h>
#define UTILITY_H


class ArrayBlock {

// This is the structure of the list elements in class Array.

public:

  // The void pointer that can be type-casted.
  void* OBJECT;

  // Pointer to previous element
  ArrayBlock* prevOBJECT;

  // Pointer to next element
  ArrayBlock* nextOBJECT;

protected:

private:

};


class Array {

// This is an implementation of a linked list structure, in which every
// element in the list contains a void pointer that can be easily type-
// casted for different purposes.  This particular implementation allows
// a new element to be added to or deleted from the beginning of the list.
// Similarly, a new element can be added to or deleted from the end of the
// list.  The elements are indexed (0 .. LIST_SIZE-1).

public:

  Array(void) { firstArrayBlock = lastArrayBlock = 0; };
  // Effects:  Constructor for the class.


  ~Array(void);
  // Effects:  Destructor for the class.  Destroy all elements in the
  //           list/array.  Objects to which the pointers were referring
  //           are not destroyed, however.


  void* element(int index);
  // Effects:  Return the value of the void pointer of element <index>.
  //           Elements are indexed starting at 0.  If access is out-of-bound,
```

```
  //           print error message.


  int size(void);
  // Effects:  Return the size of the list/array.


  void addElementHigh(void* element);
  // Effects:  Add a new element to the end of the list.  Assign the value
  //           <element> to the void pointer of the new element.


  void addElementLow(void* element);
  // Effects:  Add a new element to the beginning of the list.  Assign the
  //           value <element> to the void pointer of the new element.


  void deleteElementHigh(void);
  // Effects:  Delete the element at the end of the list from the list.
  //           Object to which the void pointer of the element was referring
  //           is not destroyed.


  void deleteElementLow(void);
  // Effects:  Delete the element at the beginning of the list from the list.
  //           Object to which the void pointer of the element was referring
  //           is not destroyed.

protected:

private:

  // Pointer to the first element in the list/array.  Is NIL if the list/
  // array is empty.
  ArrayBlock* firstArrayBlock;

  // Pointer to the last element in the list/array.  Is NIL if the list/
  // array is empty.
  ArrayBlock* lastArrayBlock;

};

#endif
```

The Single Nucleotide Polymorphism Detector (SNPD) / Utility.C

```
#include "utility.H"


Array::~Array(void)
//
// Effects:  Destructor for the class.  Destroy all elements in the
//           list/array.  Objects to which the pointers were referring
//           are not destroyed, however.
//
{
  ArrayBlock* currentBlock = firstArrayBlock;

  // Iterate through the linked list and delete all ArrayBlock objects.
  while (currentBlock) {
    ArrayBlock* blockToBeDeleted = currentBlock;
    currentBlock = currentBlock->nextOBJECT;
    delete blockToBeDeleted;
  }
}


void* Array::element(int index)
//
// Effects:  Return the value of the void pointer of element <index>.
//           Elements are indexed starting at 0.  If access is out-of-bound,
//           print error message.
//
{
  // Check to make sure index is greater or equal to 0.
  if (index < 0)
    cerr << "Array::element(int index):  access out of bound\n";

  int counter = 0;
  ArrayBlock* currentBlock = firstArrayBlock;

  // Iterate through the linked list until the end of the list is
  // reached, or the list ends.
  while (currentBlock && (counter < index)) {
    counter++;
    currentBlock = currentBlock->nextOBJECT;
  }

  // If the list ends on an element, return the value of the void pointer.
  // Otherwise, it might have been an out-of-bound access.  Print error
  // message.
  if (currentBlock)
    return currentBlock->OBJECT;
  else
    cerr << "Array::element(int index):  access out of bound\n";

  return 0;
}
```

```
int Array::size(void)
//
// Effects:  Return the size of the list/array.
//
{
  // Iterate through the linked element until the end of the list, a NIL
  // element, is reached.  Increment the counter by 1 for each element
  // traversed.
  int counter = 0;
  ArrayBlock* currentBlock = firstArrayBlock;

  while (currentBlock) {
    counter++;
    currentBlock = currentBlock->nextOBJECT;
  }

  return counter;
}


void Array::addElementHigh(void* element)
//
// Effects:  Add a new element to the end of the list.  Assign the value
//           <element> to the void pointer of the new element.
//
{
  ArrayBlock* newArrayBlock = new ArrayBlock;
  newArrayBlock->OBJECT = element;

  // Check to see if the list is empty.  If it is empty, create a new
  // element and update <firstArrayBlock> and <lastArrayBlock> appropriately.
  if (!firstArrayBlock && !lastArrayBlock) {
    newArrayBlock->prevOBJECT = 0;
    newArrayBlock->nextOBJECT = 0;
    firstArrayBlock = newArrayBlock;
    lastArrayBlock = newArrayBlock;
  }

  // If the list is not empty, create a new element and update <lastArrayBlock>
  // pointers of previously last element appropriately.
  else {
    lastArrayBlock->nextOBJECT = newArrayBlock;
    newArrayBlock->prevOBJECT = lastArrayBlock;
    newArrayBlock->nextOBJECT = 0;
    lastArrayBlock = newArrayBlock;
  }
}


void Array::addElementLow(void* element)
//
// Effects:  Add a new element to the beginning of the list.  Assign the
//           value <element> to the void pointer of the new element.
//
{
  ArrayBlock* newArrayBlock = new ArrayBlock;
  newArrayBlock->OBJECT = element;

  // Check to see if the list is empty.  If it is empty, create a new
  // element and update <firstArrayBlock> and <lastArrayBlock> appropriately.
  if (!firstArrayBlock && !lastArrayBlock) {
    newArrayBlock->prevOBJECT = 0;
```

```
    newArrayBlock->nextOBJECT = 0;
    firstArrayBlock = newArrayBlock;
    lastArrayBlock = newArrayBlock;
  }

  // If the list is not empty, create a new element and update
  // <firstArrayBlock> pointers of previously first element appropriately.
  else {
    firstArrayBlock->prevOBJECT = newArrayBlock;
    newArrayBlock->prevOBJECT = 0;
    newArrayBlock->nextOBJECT = firstArrayBlock;
    firstArrayBlock = newArrayBlock;
  }
}


void Array::deleteElementHigh(void)
//
// Effects:  Delete the element at the end of the list from the list.
//           Object to which the void pointer of the element was referring
//           is not destroyed.
//
{
  // Delete the last element in the linked list and update pointers
  // appropriately.
  lastArrayBlock = lastArrayBlock->prevOBJECT;
  delete lastArrayBlock->nextOBJECT;
  lastArrayBlock->nextOBJECT = 0;
}


void Array::deleteElementLow(void)
//
// Effects:  Delete the element at the beginning of the list from the list.
//           Object to which the void pointer of the element was referring
//           is not destroyed.
//
{
  // Delete the first element in the linked list and update pointers
  // appropriately.
  firstArrayBlock = firstArrayBlock->nextOBJECT;
  delete firstArrayBlock->prevOBJECT;
  firstArrayBlock->prevOBJECT = 0;
}
```

The Single Nucleotide Polymorphism Detector (SNPD) / Makefile for MAKE

```
#!/bin/make
# -*-Mode:text;-*-


# The machine architecture (currently supports SUN4, MIPS, and ALPHA)
ARCH=$(HOSTTYPE)


# The top-level directory of this package.
prefix= .


BINFILE = ../jamalah.$(ARCH)
SHELL = /bin/sh
MAKE = make
CC = g++
CC_OPTS = -g -O -fno-for-scope -D$(ARCH)
CFLAGS = $(CC_OPTS)

INSTALL = install -c
INSTALL_PROGRAM = $(INSTALL) -m 755
INSTALL_DATA = $(INSTALL) -m 644


# Where to install binaries.
bindir=$(prefix)/bin


SOURCES = analysis.C project.C sequence.C parameter.C utility.C
OBJECTS = $(SOURCES:.C=.o)
.SUFFIXES:     .C $(SUFFIXES)


LDFLAGS =
OUTPUT_OPTION = -o $*.o


COMPILE.C = $(CC) $(CFLAGS) -c
LINK.C = $(CC) $(CFLAGS) $(LDFLAGS)


.C.o:
       $(COMPILE.C) $(INCLUDES) $(OUTPUT_OPTION) $<
.C:
       $(LINK.C) $@ $< $(LDLIBS)


all: $(BINFILE)

install: all
       $(INSTALL_PROGRAM) $(BINFILE) $(bindir)
clean:
       -rm -f *.o $(BINFILE)
```

```
TAGS: $(SOURCES)
	etags --typedefs $(SOURCES)

$(BINFILE): $(OBJECTS)
	$(LINK.C) -o $(BINFILE) $(OBJECTS) $(LDLIBS)


project.o:  project.H project.C sequence.H analysis.H parameter.H utility.H
sequence.o:  project.H sequence.C sequence.H
analysis.o:  project.H analysis.C analysis.H sequence.H utility.H
parameter.o:  parameter.C parameter.H
utility.o:  utility.C utility.H
```

The Primer Order and SNP Data Processor / OrderDataProcessor.pl

```
#!/usr/local/bin/perl
# -*-Mode: perl;-*-

# Print things out as they come.
$|=1;

# Include some important functions.
require "boulderio.pl" || die "$0: $!\n";
require "ctime.pl" || die "$0: $!\n";

# We don't want to send boulderio data out, so we turn that feature off.
$boulder'passthru = 0;

# Some configuration variables.
$complement{"G"}="C";
$complement{"C"}="G";
$complement{"T"}="A";
$complement{"A"}="T";
$complement{"N"}="N";

# Extra primer tail attached to forward primer for dye primer sequencing.
$forward_primer_tail = "TGTAAAACGACGGCCAGT";

# Initialize more variables.
$number_primer_pair = 0;
$number_base_forward = 0;
$number_base_reverse = 0;
$max_prod_len = 0;

# Whitehead/MIT CGR internal index
$DWU_index_start = 2582;

# Write SNP data to file.
open(DATAFILE, ">Data_File");

# Read stdin into a boulder record, one at a time.
while (%r = &read_record) {

    # Check to see if primer pairs have have been picked..
    if ($r{FORWARD_PRIMER}) {
        $number_primer_pair++;

        # Find forward primer region
       $r{FORWARD_PRIMER}=~/([0-9]+),\s*([0-9]+)/;
       $left_start = $1;
       $left_length = $2;

        # Find reverse primer region
       $r{REVERSE_PRIMER}=~/([0-9]+),\s*([0-9]+)/;
       $right_start = $1;
       $right_length = $2;

        # Figure out forward primer sequence and attach tail to it.
```

```
        $forward_primer = substr($r{SEQUENCE},$left_start,$left_length);
        $forward_primer = $forward_primer_tail . $forward_primer;

        # Figure out reverse primer sequence.
        $reverse_primer = "";
        for ($i = $right_start; $i>=($right_start - $right_length + 1); $i--) {
            $reverse_primer .= $complement{substr($r{SEQUENCE},$i,1)};
        }

        # Generate modified cluster names.
        $new_name = $number_primer_pair . $r{MARKER_NAME};

        # Generate Whitehead/MIT CGR internal names.
        $DWU_index = $DWU_index_start + $number_primer_pair - 1;
        print "DWU-";
        print $DWU_index;
        print "F\t";
        print "$forward_primer\t";
        print "DWU-";
        print $DWU_index;
        print "R\t";
        print "$reverse_primer\n";

        # Print info to data file as tab-delimited columns.
        print DATAFILE "DWU-";
        print DATAFILE $DWU_index;
        print DATAFILE "\t$r{MARKER_NAME}\t$r{PRODUCT_SIZE}\t";

        for ($i = 0; $i < length($r{SEQUENCE}); $i++) {
            $index_string = "POLYMORPHISM_" . $i;
            if ($r{$index_string}) {
                $polymorphism_string = $r{$index_string};
                print DATAFILE "[";
                for ($j = 0; $j < (length($polymorphism_string) - 1); $j++) {
                    print DATAFILE substr($polymorphism_string, $j, 1);
                    print DATAFILE "/";
                }
                print DATAFILE substr($polymorphism_string,
                                      length($polymorphism_string) - 1, 1);
                print DATAFILE "]";
            }
            else {
                print DATAFILE substr($r{SEQUENCE}, $i, 1);
            }
        }
        print DATAFILE "\t";

        print DATAFILE "$forward_primer\t$r{PRIMER_FORWARD_TM}\t";
        print DATAFILE "$reverse_primer\t$r{PRIMER_REVERSE_TM}\n";

        # Total numbers of bases for forward primers and for reverse primers.
        $number_base_forward += length($forward_primer);
        $number_base_reverse += length($reverse_primer);
    }
    %r = ();
}

print "$number_primer_pair\t$number_base_forward\t\t$number_base_reverse\n";
print DATAFILE "$number_primer_pair\n";

# close data file.
close(DATAFILE);
```