ACCURATE, PARALLEL CLUSTERING OF EST (GENE) SEQUENCES

by

Kevin Thomas Pedretti

A thesis submitted in partial fulfillment of the
requirements for the Master of Science
degree in Electrical and Computer Engineering
in the Graduate College of
The University of Iowa

May 2001

Thesis supervisor: Professor Thomas L. Casavant

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

MASTER'S THESIS

This is to certify that the Master's thesis of

Kevin Thomas Pedretti

has been approved by the Examining Committee for the thesis requirement for the Master of Science degree in Electrical and Computer Engineering at the May 2001 graduation.

Thesis committee: _____
Thesis supervisor

_____
Member

_____
Member

ACKNOWLEDGEMENTS

I would like to convey my gratitude to those who helped me on this project, including my advisor, Thomas L. Casavant, and my fellow colleagues, Terry A. Braun, Todd E. Scheetz, and Chad A. Roberts. This work would not have been possible without all of your insightful ideas and feedback.

TABLE OF CONTENTS

Page

LIST OF FIGURES vi

CHAPTER

1 INTRODUCTION 1

2 BACKGROUND 5

2.1 Biological Background 5
2.1.1 Gene Structure 5
2.1.2 Alternative Splicing 8
2.1.3 Genome Sequencing 8
2.1.4 EST Sequencing 10
2.1.5 EST Clustering 12
2.2 Survey of Other EST Clustering Tools 13
2.2.1 NCBI UniGene 13
2.2.2 TIGR Gene Index 14
2.2.3 ICAtools 15
2.2.4 SANBI STACK 15

3 PROBLEM STATEMENT 17

4 APPROACH 20

4.1 Fundamentals 20
4.1.1 Program Parameters 22
4.1.2 Organization of Outputs 27
4.2 Optimization 28
4.2.1 Hashing 28
4.2.2 Global Hash Table 30
4.2.3 Parallel Execution 30

5 IMPLEMENTATION 32

5.1 Common Implementation Details 32
5.1.1 High Level Solution Structure 33
5.1.2 Comparing Sequences 36
5.1.3 Hashing 40
5.2 TLcluster (Version 1) 42
5.3 UIcluster 2.0 46
5.3.1 Global Hash Table 46
5.3.2 Extended Search 49
5.3.3 Reverse Complement Checking 49
5.3.4 Additional Minor Changes 50
5.4 UIcluster 3.0 51
5.4.1 Parallel Execution 51
5.4.2 Virtual Primaries 52
5.5 Running UIcluster 3.0 54
5.5.1 Compiling 55
5.5.2 Command Line Options and Usage 56
5.5.3 Output File Format 58

6 RESULTS 60

6.1 EST Sequencing Novelty Assessment 60
6.2 Generation of Gene Indices 62
6.3 Accuracy Assessment 64
6.3.1 Comparisons to NCBI UniGene 64
6.3.2 Analysis of Cluster Assemblies 66
6.4 Performance Assessment 67
6.4.1 Execution Time 67
6.4.2 Memory Usage 70
6.4.3 Parameter Variation 71

7 CONCLUSION AND FUTURE WORK 74

7.1 Alternative Transcript Identification 74
7.1.1 Without Genomic Sequence 75
7.1.2 With Genomic Sequence 75
7.2 Confirming Gene Predictions 76
7.3 Manual Curation 76
7.4 Cluster Merging 77
7.5 Long Transcribed Sequences 79

7.6 Automatic Calculation of ζ' and λ 80
7.7 ExtendMatch Improvements 80

APPENDIX UICLUSTER 3.0 SOURCE CODE 81

A.1 Header Files 81
A.1.1 uicluster.h 81
A.1.2 cluster.h 82
A.1.3 compare.h 83
A.1.4 fasta.h 83
A.1.5 incremental.h 84
A.1.6 memory.h 84
A.1.7 options.h 84
A.1.8 qsort.h 85
A.1.9 utils.h 85
A.1.10 bl2seq.h 86
A.2 Source Files 86
A.2.1 main.c 86
A.2.2 cluster.c 89
A.2.3 compare.c 105
A.2.4 fasta.c 111
A.2.5 incremental.c 114
A.2.6 memory.c 118
A.2.7 options.c 119
A.2.8 qsort.c 123
A.2.9 utils.c 125
A.2.10 bl2seq.c 129

REFERENCES 133

LIST OF FIGURES

Figure Page

2.1 DNA Double Helix (adapted from [33]) 6

2.2 Gene Structure (adapted from [23]) 7

2.3 Alternative Splicing 8

2.4 Insertion, deletion and misread errors 9

2.5 High-level overview of EST sequencing 11

2.6 Example EST Sequence 13

5.1 High Level Data Flow 33

5.2 Basic Flow of Execution 35

5.3 Expanded Clustering Control Flow (line 4c from figure 5.2) 36

5.4 Example of `ScoreMatch` Execution 39

5.5 Example of Hashing a Sequence 42

5.6 Primary and Secondary Data Structures 43

5.7 Global Hash Table 47

5.8 UNIX Commands for Compiling `UIcluster` 56

5.9 `UIcluster 3.0` Command-line Interface 57

6.1 Incremental Library Novelty 61

6.2 Comparing Clusters 65

6.3 TLcluster vs. UIcluster 68

6.4 Parallel Speedup 69

6.5 Parallel Memory Scaling 70

6.6 Effects of clustering options on execution time 72

7.1 Cluster Viewer 78

CHAPTER 1
INTRODUCTION

Sequencing of cDNA (complementary DNA) is influenced by an additive random process which increases the potential for errors over DNA sequencing. In addition, the process by which cDNA clones are selected for sequencing introduces redundancy. For these reasons, automated software tools are necessary to classify large data-sets of cDNAs into groups that roughly correspond to genes. Thus, a crucial tool needed for this is a computer application to form clusters based on sequence similarity from the raw cDNA sequence data. This thesis presents the design and evolution of a program that has been created to accomplish this task with the characteristics of flexibility, efficiency, and accuracy. Although there are several existing software tools [27] [20] [2] [21] available that perform genetic sequence clustering accurately, this program is unique in its high degree of flexibility and in its computational efficiency. Furthermore, the program is non-proprietary and may be freely obtained from our project web site (http://genome.uiowa.edu).

An EST, or *expressed sequence tag* [1], is most generally a sequence obtained by performing a single read of a random complementary DNA (cDNA) clone. A specific type of EST sequenced from the 3' end of a cDNA has the unique property that it is analogous to a finger print – it can readily be used as a unique identifier for

a gene. Thus, high-throughput gene discovery projects generate large numbers of 3' ESTs, in an effort to find new genes.

Clustering is the process of partitioning a set of elements into meaningful groups (clusters) so that members of each group are more similar to each other than to members of any other group. In the context presented here, ESTs and other forms of genetic sequence are the elements being clustered and cluster membership is determined based on sequence similarity. The ultimate goal is to partition the elements so that each cluster represents all known genetic information for a single gene or gene family. The importance of this result bears on several aspects, but the principle of these is creating non-redundant indices of genes. These indices are an essential tool for assessing novelty rates and guiding future gene discovery efforts.

An additional important use of clustering is to identify EST sequences that have a high potential of being derived from an alternative transcript of a known (or unknown) gene. A gene, as contained in genomic DNA, can often encode the information necessary to produce multiple proteins. The genomic DNA is processed by the cell into a messenger RNA (mRNA) transcript that in turn produces a protein. Since ESTs are derived from post-spliced mRNA, they provide a convenient way to identify different gene transcripts. Analysis of the consistency of the sequences in a cluster can identify candidates that possibly represent alternative transcripts. Further sequencing and genomic sequence data can then provide more thorough verification.

A brute-force, exhaustive solution to the clustering problem is not, however,

computationally feasible. An $O(n^2)$, where n is the number of sequences, computation is required to identify all sequence similarities. This may be a sufficient approach for 100s or even a few 1000s of ESTs, but it cannot possibly scale to data sets of millions of ESTs. For data-sets of that size, heuristics must be employed to simplify the computation. The task is to simplify the computation enough to be practical while retaining sufficient accuracy to provide meaningful results.

The clustering program discussed in this thesis has been implemented and proven to achieve both accuracy and performance. It has been developed over the course of four years and has had three major releases. Each release has built upon the prior by incorporating new functionality and increased performance. The first version of the tool [26] was developed by Professor Thomas Casavant and released in Fall 1998. The two subsequent versions have been developed by Kevin Pedretti under the supervision of Professor Casavant. The robustness of this program has been demonstrated by its daily use in the production pipeline of large-scale gene discovery projects under way at the University of Iowa. Its use has resulted in the estimated discovery of more than 40,000 new genes in three mammalian species (human, mouse, and rat) [10].

Chapter 2 provides the biological background necessary to understand EST clustering. Chapter 3 is comprised of a concise problem statement. Chapter 4 discusses the high-level approach used in the three releases of the program. Chapter 5 gives specific implementation details of each release. Chapter 6 presents results

obtained by using the program including accuracy and performance measurements. Finally, Chapter 7 concludes with an outline of directions for future development. The appendix lists the source code for the latest release, UICluster 3.0.

CHAPTER 2
BACKGROUND

This chapter will present the biological background necessary to understand the basis for expressed sequence tag (EST) sequencing [1] and why the clustering of ESTs is important. The last section of the chapter will present a survey of other EST clustering programs.

2.1 Biological Background

2.1.1 Gene Structure

Current definitions of genes are inadequate and ambiguous in describing heritable units of a genome. Here, a gene is defined as a well structured and localized region in the genome that encodes the information necessary for producing one or more proteins. A gene is the basic unit of heredity, passing along traits such as eye color and diseases such as cystic fibrosis. Having an extra gene, missing a gene, or having a mutated gene are some of the mechanisms by which genetic diseases can manifest themselves. However, disease inheritance is complicated and is not yet fully understood. There are higher level interactions among genes and other structures in the genome that play significant roles. Disease expression is also influenced by the environment. Understanding every gene in the human genome is an anticipated by-product of the current sequencing efforts, however this goal will take decades to

achieve.

The genome consists of DNA, which is the double-helix molecule located in cell nuclei. An organism's genome is located in the nucleic DNA of each of its cells. Figure 2.1 shows the structure of a partial double-stranded region of DNA. The double-helix structure can be thought of as taking a ladder and twisting it. Each rung is composed of one purine base, adenine (A) or guanine (G), and one pyrimidine base, cytosine (C) or thymine (T). Adenine exclusively pairs with thymine and guanine exclusively pairs with cytosine. Each rung in the double-helix structure is commonly referred to as a base pair (bp).

Figure 2.1: DNA Double Helix (adapted from [33])

The human genome is made up of 23 DNA molecules, called chromosomes, containing an estimated $3x10^9$ base pairs. These chromosomes are currently estimated to contain between 30,000–40,000 protein coding genes [17]. An abstract view of a gene's structure is shown in figure 2.2.

A cell processes this structure by transcribing it (from the transcription start to

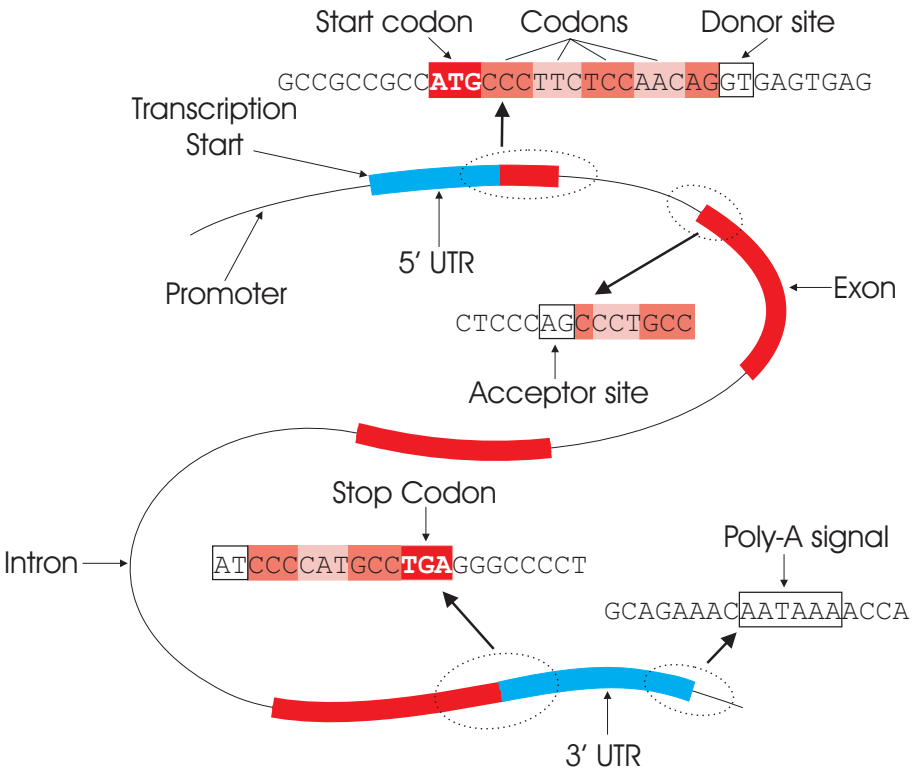

Figure 2.2: Gene Structure (adapted from [23])

the Poly-A signal) into mRNA (messenger RNA), and then processing the mRNA by concatenating the highlighted regions called exons. The final product represents the information necessary to synthesize a single protein molecule. The removed regions, represented in the figure by thin lines are called introns. Acceptor and donor sites, located in the intronic regions and flanking each exon are the signals used by the cellular machinery to identify exon boundaries and aid in the concatenation process.

2.1.2 Alternative Splicing

Concatenation appears to be a probabilistic process and exons are sometimes skipped by the cell's machinery, being left out of the final mRNA product. This phenomenon is a mechanism by which a single gene can be translated into multiple transcripts, each coding for a different protein. Approximately 30-40% of the genes in the human genome are thought to have multiple transcripts [17]. Figure 2.3 shows some examples of alternative splicing.

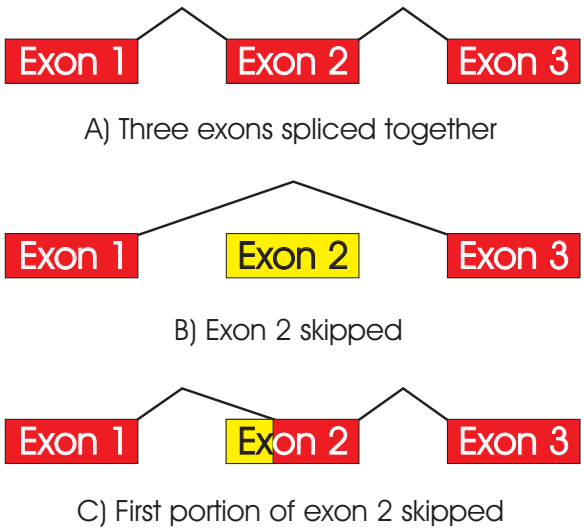

Figure 2.3: Alternative Splicing

2.1.3 Genome Sequencing

Genome sequencing is the process of identifying the base pair sequence of every chromosome in an organism's genome. The exact details of this process [24] are not important for understanding this thesis. However, it is important to understand the

general nature of this process.

Automated sequencing machines have been developed to enable genome-level, high-throughput sequencing projects to be feasible. These machines carry out many sequencing reactions (a chemical reaction) in parallel. Current state-of-the-art technology allows for roughly 500-1000bp to be obtained in each sequencing reaction. However, these reactions are error prone and tend to become even more error-prone as sequencing lengths increase. There are three errors that can occur: bases can be inserted, deleted, or misread. Examples of these three events are shown in figure 2.4. Sequencing errors sometimes occur in groups, such as a run of multiple bases being inserted/deleted. Also, the error-rates at the beginning and end of a sequencing reaction are relatively higher than the error rates in the middle of a reaction (e.g. the first 100bp and last 100bp of a reaction will have more errors).

Figure 2.4: Insertion, deletion and misread errors

For this reason, a given region of the genome must be sequenced many times before a high-quality consensus sequence can be formed. The current standard of the Human Genome Project is to have each base sequenced eight times (8x coverage)

before calling it finished.

2.1.4 EST Sequencing

An EST (expressed sequence tag) is a special type of sequence that is useful for high-throughput gene discovery. Genome level sequencing produces the base pair sequence of an organism's genome but does nothing to identify where the genes are located. Since less than 5% of the human genome codes for genes [17], identifying the genes amounts to finding a needle in a haystack. Gene prediction programs such as Genscan [7] and GRAIL [14] can be used to locate and computationally predict gene structure (where the exons are), however they are limited in their accuracy. EST sequencing provides a shortcut to accurately and efficiently identifying genes directly by sequencing the complimentary DNA (cDNA). cDNA has the intronic regions removed and contains only the concatenated transcript of a gene.

Figure 2.5 shows a high level overview of EST sequencing [1]. To prepare for EST sequencing, mRNA molecules are extracted from cells and converted into cDNA through reverse transcription [5] [34]. The cDNAs are then cloned into vectors, and electroporated into bacteria for growth, amplification, and storage. A collection of such cDNAs is refereed to as a library. Each cDNA library potentially contains many unique and previously undiscovered gene transcripts. However significant redundancy within a library (multiple copies of the same cDNA) and between libraries is normal.

High throughput EST sequencing for gene discovery involves sequencing the 3' untranslated region (UTR) of randomly chosen cDNA transcripts from a cDNA

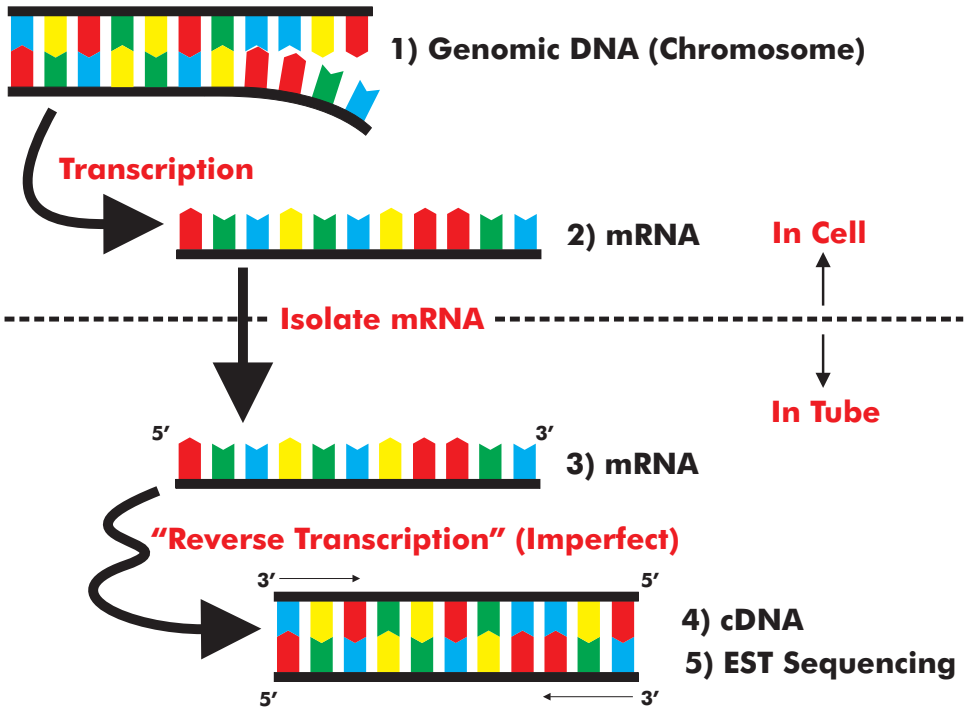

Figure 2.5: High-level overview of EST sequencing

library. The sequence is obtained by performing a single sequencing reaction, not the multiple reactions as discussed in the previous section. Empirical studies have shown that the error rate for EST sequencing is approximately 5% for misread errors, and 1-2% for insertion/deletion errors [6].

The 3' UTR sequence is the most divergent [1], and thus the most useful portion of a transcript for identification purposes. It is on average 750bp long. The probablility of another gene having the same 3' UTR is extremely low. The use of a poly-T primer during reverse transcription allows for the preferential selection of cDNAs with a poly-A tail at the 3' ends. The presence of this feature allows for sequencing to usually start from a known position (at the poly-A site).

2.1.5 EST Clustering

The massive number of EST sequences generated by high-throughput gene discovery projects need to be clustered into groups based on sequence similarity. Ideally, each cluster will exclusively contain all of the sequenced ESTs for a particular gene. The results of doing this are used to assess the novelty rate of new sequences and provide feedback information to the sequencing pipeline. If the novelty rates reported by clustering (roughly equal to the gene discovery rate) fall below a certain threshold, laboratory procedures can be performed to filter out already sequenced transcripts from a cDNA library [5].

Comparing pairs of ESTs and looking for similarity is the basic operation to clustering. This comparison is complex because of the single-pass nature of EST sequencing. As was already mentioned, bases can be inserted, deleted, or misread. This means that some form of edit distance calculation is required to optimally derive the similarity between two sequences.

From the computational perspective, an EST is a character string made up of letters from the alphabet A, C, T, G, X, N where A, C, T, and G represent DNA bases and X and N represent masked and ambiguous regions, respectively. A typical EST sequence is shown in figure 2.6. Masked regions denote bases that have been identified to be repetitive or contain low complexity. At least 45% of the human genome consists of repetitive elements [17]. If masking were not performed, spurious sequence similarities would be found. Ambiguous regions denote bases that could not

be accurately determined by the sequencing machine or base calling program. There are examples of both of these types of regions in figure 2.6. 3' ESTs are typically between 400–1000bp in length. This is a limitation of the current gene sequencing technology and the lengths may grow larger in the future.

```
>UI-R-A0-ae-b-09-0-UI
TTTTTTTTTTTTTTTTTTGATTTTCAATGATAAACTTTTATTCTGAATATACTGTTTTTGCACAAGATTTA
ACACAACATTTTCTGGGXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXCAAAATGTGTTCA
TCCGACTAGTTAATTTCCACAAAAGTGTCCAGAGAACATAATAAGGGGGAGAAAAAAAATCTGTTGTTCA
CAAAAGCCACTTGGCGTTTTGCTTGATGCACAATGAGCATTTCATGAAGAGAATCCCTAAAACATGATCC
CACAGTCATACCGCACAAGGAAAGAACAGCTTGGCCAGGTCACATTGGAAACTCAATTGGCATTTACACC
GGACAGCATGCCAGGAGTCTCAGTGGAATTTCCATGGTTCTTTTTTGTGTGAACTAGAAACAAGGTATAC
GAAACCTCCCGTAACAGCAATCTATTTCTGCAAAATTCTGGCCATTTTCATGACCTGATAGTTCTGTTTT
AGTGATTTGCTCTTTACAGAAATATACACCAGATAGTGACCATATCAACATTCTGCCATGGAGAACAATG
CAAGTTCCAGCGAATGATAAAATAA
```

Figure 2.6: Example EST Sequence

2.2 Survey of Other EST Clustering Tools

This section briefly discusses other EST clustering tools.

2.2.1 NCBI UniGene

UniGene [27] is an experimental system used at the National Center for Biotechnology Information (NCBI) for automatically partitioning EST and other sequences into non-redundant sets of gene-oriented clusters. Ideally, each UniGene cluster contains sequences that represent a unique gene, as well as related information such as the tissue types in which the gene has been expressed, and map location.

The UniGene clustering procedure is broken down into multiple steps, with each stage adding less reliable data to the results of the proceeding stage. At each

stage, an essentially NxN sequence comparison is performed to generate a weighted graph where the vertices are sequences and the edges are weighted according to sequence similarity. Sequences with edges exceeding a threshold are merged into the same cluster. A detailed description of the stages is given at (http://www.ncbi.nlm.nih.gov/UniGene/build.html).

While the build procedure is public, the actual scripts and tools used are not readily attainable nor are they flexible enough to be used in an environment outside of NCBI. In addition, it appears as if the procedure is started from the beginning each time the UniGene index is built. However, NCBI UniGene is probably the most widely used gene index and is one of the standards to which our clustering procedure will be compared.

2.2.2 TIGR Gene Index

The TIGR Human Gene Index [2] uses a strict assembly method to group highly related sequences into clusters. The method disregards inconsistent matches when forming clusters in favor of confidence based on perfect or near perfect sequence overlaps. This design choice means that sequences possibly representing alternative splice forms will not be considered for cluster membership. However, the benefit is high confidence that the sequences in any given cluster are truely related. The drawback is that *underclustering* will occur (i.e. to many clusters) and the number of genes will be over estimated.

2.2.3 ICAtools

The ICAtools [21] are a set of programs that are designed for doing medium scale cDNA sequence clustering. The program inputs files of cDNA sequence information and produces an index file which links similar sequences together in clusters. ICA is an acronym for Incremental Clustering Algorithm which describes the way the program builds its index one sequence at a time. The incremental nature of the program is very desirable since it eliminates the need to start from the beginning when only a few new sequences need to be clustered. The program also uses the notion of small, exact matches between sequences in a filtering step before doing a comparison. The default size for the exact match length is six bases. Two sequences are only compared if at least one of these short regions is in common between them. This saves by avoiding sequence comparisons, a time-consuming operation, that have no chance of being similar.

The ICAtools are freely available from the Internet (http://www.hgmp.mrc.ac.uk/Registered/Option/icatools.html).

2.2.4 SANBI STACK

The STACK [20] clustering system aims to cluster ESTs and full-length cDNA sequences into high-quality clusters. The difference between STACK and UniGene is that STACK attempts to generate consensus sequences for each cluster using the phrap [13] program. These consensus sequences can be used to detect alternative transcripts of the same gene. Also, the developers claim that the STACK gene index

is generated more carefully and has a greater degree of error checking than UniGene. The `d2_cluster` [8] program is used to form what they describe as a "loose" clustering based on the total number and multiplicity of (possibly discontiguous) matching 6-base words, rather than sequence alignment. The goal of loosely clustering is to preserve information about alternative splice forms. A post processing step, performed by another program called `craw` [9], is used to identify the possible multiple transcripts contained in a cluster. Their analysis [20] shows that their clustering is between 13–16% less fragmented than UniGene clusters. Fragmentation occurs when two distinct clusters exist that should actually be a single cluster.

The STACK tool is available freely for academic use from the Internet (http://www.sanbi.ac.za/CODES).

CHAPTER 3
PROBLEM STATEMENT

Large-scale gene discovery projects require rapid and accurate clustering of EST sequences for maximum efficiency [11]. Novelty rate estimates (i.e. the number of clusters divided by the total number of ESTs sequenced) provided by clustering are a key part of the feedback loop to a gene discovery sequencing pipeline. This information is used to decide when to perform serial-subtractions, which have been shown to dramatically increase novelty rates [5]. High overall novelty rate is the primary goal of these projects. Furthermore, clustering results can provide valuable insights into gene family relationships and clues to the identification of alternative splicing sites.

These important uses of clustering make it imperative that the technique chosen be both efficient and accurate. If done in a naive fashion, such as a NxN comparison, the problem is intractable for any reasonably sized data set. On current PC hardware (e.g., 600MHz Pentium III), benchmarks have shown that a Smith-Waterman [29] comparison of two EST sequences requires on average 5 milliseconds. For a data set of 1 million ESTs, an $O(n^2)$ clustering would require approximately 80 years. However, typical data sets will be highly redundant and the number of clusters will be much less than the number of sequences clustered. A better approach

would be an algorithm that scales proportionately to the number of clusters. Such an algorithm would still be $O(n^2)$ for the worst case (i.e., every sequence is a cluster) but would be much faster in practice. Clearly, there are significant opportunities to utilize heuristics and other optimization techniques to speed this computation. However, careful evaluation is necessary to be confident that the approximated clustering results match as closely as possible the solution that would have resulted from an exhaustive approach.

There are several existing EST clustering solutions in use at different labs around the world. Principle among these are NCBI UniGene [27], ICATools [21], TIGR Human Gene Index [2], and SANBI STACK [20]. These tools have already been discussed briefly in the previous chapter. While these tools are useful, they are often not flexible enough to be generally useful to outside laboratories. There is a need for a clustering program that combines the strengths of these programs, but is flexible enough to be useful in a wide-array of applications and laboratory environments. Furthermore, there is a need for a program that has higher performance and is more scalable than the currently available tools. Parallel execution, distributing both computation and memory, along with improved heuristics, are methods that could be used to achieve this.

The task of this thesis is to describe the design and implementation of a high-performance, accurate, and flexible clustering software application. The algorithms employed have been chosen to optimize the trade-off between performance and accu-

racy. Of particular importance is the ability to handle large data sets (more than 1 million ESTs) with reasonable computation time on commodity PC hardware. Furthermore, the application has been made flexible by using carefully chosen run-time parameters. A novel goal of the software package is to be easily adaptable to the clustering needs of other projects.

CHAPTER 4
APPROACH

The first section of this chapter presents the fundamental approach we have taken to the clustering problem. The important characteristics of our solution are discussed from a high-level standpoint. The second section of the chapter discusses the optimization techniques we have employed in our solution.

The clustering application that implements our approach has evolved over the course of four years and has been released to the public as three major versions. Each successive release has built upon the previous and implemented more of the details discussed in this chapter. When necessary, a note will be made of what version of the application a particular feature was first incorporated.

4.1 Fundamentals

The definition of a cluster and the criteria for cluster membership are fundamental parameters that first need to be determined. NCBI UniGene [27], for example, defines a cluster as containing all known genetic information for a given gene. This includes alternative splice forms of a gene. Cluster membership is determined in a multi-stage, graph-based approach by which clusters are formed based on sequence similarity and known annotations. Essentially an NxN comparison is performed to construct a graph where edges between sequences are weighted with the similarity

score. SANBI STACK [20] takes a similar approach, first generating a graph of sequence similarities. However, instead of using a traditional sequence alignment to determine similarity, the criteria used is the multiplicity and number of 6 base words in common between sequences. Both of these approaches are valid and useful. However, we take a different approach that avoids the $O(n^2)$ comparisons for typical data sets. The advantage is substantially improved performance enabling the ability to run effectively on commodity PC hardware. The risk is generating less accurate results, although our analysis has shown this not to be the case.

Instead of pre-computing all sequence similarities before forming the clusters, we take a more dynamic, incremental approach to the problem. Sequences are read one at a time from an input file and compared against one representative sequence from every existing cluster. These representative sequences are called *primaries*. The non-primary sequences of a cluster are called *secondaries*. The sequence being clustered is added to a cluster if it is found to be similar to the cluster's primary. If the sequence is similar to no existing clusters, it becomes the primary sequence of a new cluster.

As with existing clustering applications, the computation becomes more complex as the data sets clustered grow larger. However, for our approach the computation scales proportionally to the number of clusters rather than the number of sequences since only the cluster primaries are compared against. This produces a large benefit because EST data sets typically have significant redundancy, meaning that

the number of clusters will be much lower than the number of sequences clustered.

4.1.1 Program Parameters

This section discusses the parameters and optional features of our approach that are configurable by the user. These parameters afford the user a large degree of flexibility when performing a clustering. Different option sets are appropriate for obtaining different types of results. Additionally, the application can be executed with different parameters several times for the same data set and the results can be compared.

4.1.1.1 Incremental Clustering

A key feature that has been incorporated since the earliest version is the ability to perform incremental clustering. In this mode of operation, one or more files containing the results of previous clusterings can be input to the application and used when performing a new clustering. This is more efficient than reprocessing all of the data from the beginning as do the graph-based approaches of UniGene [27], STACK [20], and TIGR [2] gene index. Notably, ICAtools [21] provides similar incremental clustering functionality but requires that each cluster be stored in a separate file. In addition, incremental clustering allows other analytical processes, such as tracking cluster growth over time. Our approach is to allow each file to contain more than one cluster, simplifying the administrative tasks of running the application. Previous output files of our clustering program can be directly input back into the

program when performing a new clustering. This feature can also be used as a crude form of check-pointing by splitting a large data set up into pieces and clustering each piece incrementally.

4.1.1.2 Similarity Criteria

The similarity criteria is specified by the user as N out of M, meaning that at least one matching window of M bases with no more than $M - N$ errors must be present for two sequences to be considered similar. If the number of errors permitted is relatively modest (95% identity), this criteria can be evaluated more quickly than the optimal (semi-optimal) alignment methods of UniGene, ICAtools, and TIGR Gene Index. Still, evaluating this criteria is a time consuming operation and should be avoided as much as possible. This is the goal of our optimization schemes discussed in section 4.2.

For estimating the number of genes represented in a data set, the N out of M criteria should be chosen to allow for enough errors so that true similarities are not missed while being rigorous enough that false similarities are not found. Unfortunately, there is no pre-determined method to select N and M and empirical investigations by expert biologists are necessary to determine which values to use. However, one may wish to use the clustering application for purposes other than generating gene indices. A "looser" clustering (*underclustering*), similar to that produced by STACK [20], can be performed by allowing more errors when one is looking for evidence of alternative splice forms. Conversely, a "tight" clustering *overclustering*)

can be performed by allowing fewer or no errors. This is useful when one wishes to be highly confident that members of a cluster are related, with the consequence of missing some true similarities. The strict alignment based clustering of TIGR's gene index strict alignment based clustering is an example of this mode of operation.

4.1.1.3 Repicking Primaries

A potential drawback of the cluster primary concept is that the chosen primary may not be a good representative for the cluster as a whole. For homogeneous 3' EST data, the best representative is most often the longest sequence in the cluster since each sequence theoretically starts from the same position. However, by default our algorithm uses the first discovered member of a cluster as the primary, disregarding longer sequences that are added to the cluster later. This works well as long as the input sequences are all approximately the same size or if they are pre-sorted into descending order by size. Since this is usually not the case, an option is available to repick the primary every time a sequence is added to the cluster. If the sequence being added is longer than the existing primary, it becomes the new primary for the cluster. In such a case, all of the existing secondaries are compared against the new primary. Sequences not matching the new primary, called *orphans*, are made note of in the application's output. Over time, orphan sequences can be *re-adopted* as new primaries are picked. At the end of clustering, any remaining orphans are intended to be examined by a human.

4.1.1.4 Virtual Primaries

As a more extensive attempt to address the limitations of the cluster primary concept, the latest version of the program contains an option to generate a *virtual primary* for each cluster. Every time a sequence is added to a cluster, a check is performed to see if the virtual primary can be extended. There are five possible cases that are considered: an internal hit, front extension, tail extension, tail and front extension, and no extension. An internal hit is when the added secondary is completely contained in the virtual primary. An extension occurs when the added secondary can make the virtual primary longer at its front, tail, or both. For this to occur, there can be no non-matching regions (taking into account some error) of the overlap between the virtual primary and secondary. If there are non-matching regions, then the sequence is added to the cluster but the virtual primary is not changed. The sequence should be inspected later to determine the cause of the non-consistent hit. Such sequences may be good candidates for representing an alternative transcript of the gene the cluster represents.

If the virtual primary parameter is enabled, all sequence comparisons are performed against the virtual primary instead of the primary. However, the cluster primary is also maintained and updated to reflect the longest sequence in the cluster if the repick primary parameter is enabled.

4.1.1.5 Reverse Complement Checking

A rare error that occurs when doing EST sequencing is that the opposite strand of a cDNA transcript is sequenced in the wrong direction. It is necessary to reverse complement such a sequence before it can be compared to other sequences that were sequenced in the correct orientation. To identify these errors, an optional parameter was added to the second version of the program that checks the reverse compliment of an input sequence if no match to a cluster primary is found in its original form.

An additional important use of this feature is for clustering 5' EST and full-length cDNA sequences along with 3' ESTs. These types of sequences must be reverse complimented before any overlaps with 3' EST sequences can be ruled out.

The performance implication of this feature is that the computation time may double, since the cluster space is potentially searched twice. In practice, the penalty is not this severe since a reverse compliment match is a relatively rare occurrence.

4.1.1.6 Extended Search

A parameter that was added in the second version of our clustering application is the ability to do an extended search of the cluster primaries for each sequence being clustered. By default, a sequence being clustered is greedily added to the first cluster primary that it is found to match. Enabling this parameters makes the search of the cluster primaries exhaustive. If any matches are found, the sequence is added to the cluster with the best matching cluster primary and all other matches are noted

in an output file. This feature can be useful for identifying potential alternative transcripts. Two separate clusters that are linked together by multiple sequences have a high potential of either being alternative transcripts of the same gene or members of the same gene family. Alternatively, the linking sequences could be chimeric–"false" sequences containing partial regions of separate transcripts caused by library construction errors. Human inspection of such cases is necessary to resolve any ambiguity.

The number of sequence comparisons required when performing an extended search is potentially much greater than performing the greedy search. However, the global hash table discussed in section 4.2.2 is effective in filtering the search space by eliminating primaries that can not possibly meet the similarity criteria. This lessens the overhead substantially.

4.1.2 Organization of Outputs

The main output of applying our clustering approach to a set of sequences is a file containing the identified clusters. This file contains both the sequence data and cluster hierarchy. Depending on the parameters chosen, other files are also output. There is a "reject" file that contains sequences that have been identified to be too low quality to cluster. There is an "orphan" file that lists orphaned sequences if the *repick primary* option is enabled. The "hits" file lists all of the cluster primaries hit for each sequence clustered if the *extended search* option has been enabled. Finally, there is an output file written for each cluster file input into the application when

performing an incremental clustering.

4.2 Optimization

A unique aspect of our application is its adaptability to the computing hardware that is being used to run it. A drawback of the NxN comparison methods employed by STACK and UniGene is that high-end computers are required for running them. The `d2_cluster` [8] application used by STACK was originally designed to run on a MasPar super-computer, and has recently been ported to other platforms including SGI Origin2000 and Linux PC clusters. They report that a 126 CPU SGI Origin2000 is used for building the STACK gene indices [20]. Our hash optimization schemes allow the user to intelligently balance the amount of memory used and computation time required by configuring run-time parameters appropriately. The program is able to run efficiently on commodity hardware with modest amounts of memory. In addition, multiple processors can be taken advantage of to distribute the memory and computational requirements of clustering if required.

4.2.1 Hashing

A key optimization of our sequence comparison functions is to initially search for short, exact matches by looking for hashes in common between the input sequence and the cluster primaries. A *hash* is an integer that uniquely represents a string of bases. The length of the string of bases to use, ζ, is specifiable by the user at run-time. For example, the length $\zeta = 8$ string of bases {GCCACTTG} may be represented by the integer 48406. A sequence is *hashed* by generating a unique integer for every length

ζ window of the sequence. The hashing optimization is based on the principle that it is faster to compare integers than to perform a string comparison. Integer comparisons are primitive operations implemented in hardware for every modern CPU, while string comparisons are usually implemented as library calls. In addition, hashes only need to be generated once for each sequence but are used many times. This amortizes the cost of generating the hashes over the program's execution. The trade-off is that memory usage increases because the hash lists for each cluster primary need to be stored in memory. Memory usage will scale proportional to the total number of bases contained in the cluster primaries, since a hash needs to be stored for each base position.

When performing a sequence comparison, both sequences are first hashed. Next, identical hashes between the two sequences are located. The base regions of the two sequences corresponding to the matching hash are a potential seed for a longer match, hopefully meeting the N out of M similarity criteria set by the user. At this point, a more exhaustive search is performed, taking into account errors. If the similarity criteria is not met in this comparison, the procedure moves onto the next hash in common between the two sequences and examines it. This procedure continues until a match is found or all identical hashes have been examined and discarded.

4.2.2 Global Hash Table

The GHT uses hashes to optimize the program at a higher level by filtering the entire search space into a subset of high-potential candidate primaries. The table contains an entry for each possible hash value. At each entry, there is a list of clusters that contain at least one occurrence of the entry's associated hash. When a sequence is clustered, it is hashed in the same way as described earlier. However, instead of comparing it is hashes against every primary, the GHT is traversed. Only primaries that are found by inspecting the GHT are examined. Primaries not having any hashes in common with the sequence being clustered are not examined.

A further refinement of the GHT concept is to keep a touch count for each primary in the table. Each time the primary is *"touched"*, meaning it has a hash in common with the sequence being clustered, this counter is incremented. A comparison is only performed if the touch count is incremented to become greater than a threshold that is set by the user at run-time. If the threshold is chosen too high, then some primaries meeting the user's N out of M similarity criteria may be missed. Similarly, if the touch count is chosen too low, more comparisons than necessary will be performed and performance may decrease dramatically.

4.2.3 Parallel Execution

The latest version of the clustering program has been parallelized to split up the computational and memory requirements across several computers (compute nodes). The main reason for doing this is so the program can scale to larger problem

sizes without being constrained by the memory limitations of a single computer. The increased performance is an added benefit.

In this mode of execution, each cluster is stored on exactly one compute node. A given sequence is read in from the input file and processed in parallel on each compute node. This results in a parallel search of the cluster space. Once each node has finished its search, each node's best match is collectively communicated to all compute nodes. The node with the best match stores the sequence in its memory space. If no match is found on any of the compute nodes, the input sequence becomes a new cluster and is assigned to one of the compute nodes. Clusters are balanced evenly across the compute nodes.

If the extended search option is enabled, an additional communication is performed to build a list of all matches meeting the user's similarity criteria. This list is gathered to the master compute node (the node writing the output files) and is written to a file.

CHAPTER 5
IMPLEMENTATION

This chapter presents implementation details of the three generations of the clustering applications that have been developed to date. The first version of the program, `TLcluster 1.0` was implemented by Professor Thomas Casavant in the Fall of 1998. This version was revised and expanded to produce the two subsequent major releases of the application, `UIcluster 2.0` and `UIcluster 3.0`. In this chapter, `UIcluster` will be used to collectively refer to all three of the implementations.

5.1 Common Implementation Details

Common characteristics of all versions of `UIcluster` include the high-level solution structure, the hashing algorithm, and the the sequence comparison functions. These topics will be discussed in this section and further elaborated on in the subsequent sections that specifically deal with each implementation version.

An additional implementation commonality is that each version has been written in the C programming language [18] and is intended to be run using a UNIX-based [32] [30] operating system. Appendix A lists the complete source code of the latest release (roughly $5,500$ lines). Portions of this code will be referred to throughout this chapter. The UNIX-derived Linux operating system has been used for development and testing of each version. However, an effort has been made to make the

applications as UNIX platform independent as possible.

5.1.1 High Level Solution Structure

The basic flow of data is the same for all versions of `UIcluster` and is shown in figure 5.1. Two data sources are input into the application. The first is a file (or files) containing clusters formed by previous runs of the program. These clusters are only input when performing an incremental clustering. The second data source is a file containing the sequences to be clustered. This file is formatted in the commonly used multiple FastA file format [12]. Figure 2.6 is an example of a FastA formatted sequence. The first line of a FastA sequence always begins with a "greater than" sign and is followed by the sequence name and other information. The sequence is listed after this line, and includes all lines up until the next FastA sequence record.

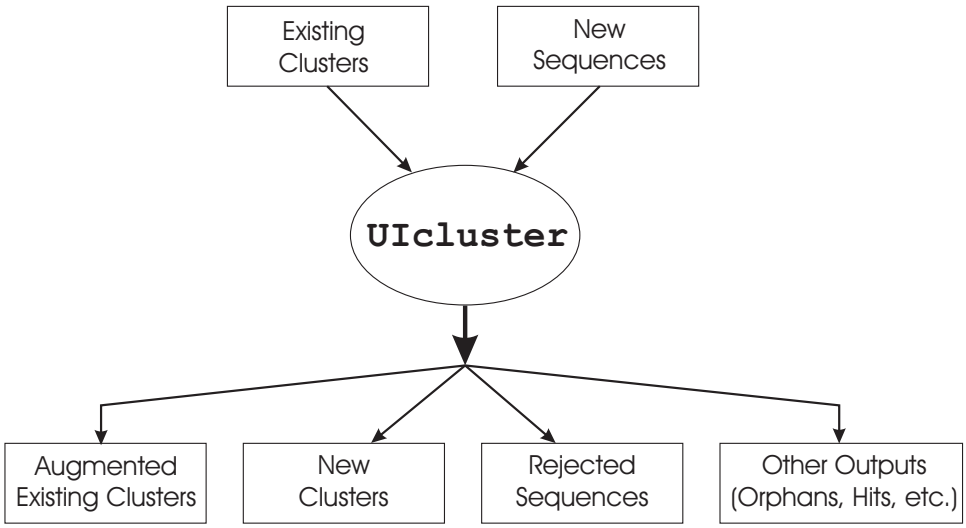

Figure 5.1: High Level Data Flow

When performing an incremental clustering, the clusters that are input into the program are processed and loaded into memory before any new sequences are clustered. New sequences are compared against these clusters in addition to any new clusters that are formed by the new sequences themselves. If a sequence being clustered is found to belong to one of the previously existing clusters, it becomes a member of that cluster.

Once the program has finished running, the clusters that have been identified are output to file. One file is output that contains all of the new clusters that were formed from clustering the input sequences. The sequences in this file were not able to be added to the previously existing clusters that were input into the application. If there were files input to the program that contained clusters to be used for incremental clustering, a new file is output for each of the files containing the modified clusters (i.e., possibly having sequences from the input set added to them). This behavior is very useful for tracking the growth of existing clusters of interest. Additionally, input sequences not meeting the user's minimum sequence length criteria (i.e., *rejected* sequences) are output to the `reject` file. Multiple other files are also output depending on which options the user has chosen to enable (e.g., a file listing all of the primaries matched, or *"hit"*, for each input sequence when the extended search option is enabled).

In addition to the data-flow, the same basic flow of execution is used for each implementation. This flow is shown in figure 5.2. Step 4 encompasses the bulk

of the program's execution. Sequences are read one at a time from the input file and clustered. If a sequence is determined to be too short based on a user-defined parameter (specified as the shortest number of consecutive valid bases allowable), it is rejected and not clustered. Rejected sequences are output to the `reject file` for later inspection.

```
1) Parse command-line
2) Allocate memory and initialize data structures
3) Read existing clusters into memory when performing an
   incremental clustering
4) While there are unprocessed sequences in the input file
        4a) Read a sequence from the input file
        4b) Determine if the sequence is a reject
        4c) If the sequence is not a reject, cluster the sequence
5) Write the clusters to the output file(s)
```

Figure 5.2: Basic Flow of Execution

If a sequence is not rejected, it is clustered in step 4c. This procedure is shown in more detail in figure 5.3. First, the sequence is "hashed", as described in section 5.1.3. These hashes are then used to search for candidate clusters that have high probability of *matching* the input sequence, based on the similarity criteria being used. Each candidate cluster's primary is compared to the input sequence to determine whether or not the similarity criteria is met. This criteria is specified by the user at run-time as N of M bases, meaning that at least one M length window containing no more than $M - N$ errors must exist between two sequences for them to be considered similar.

The two functions that are used for this comparison are described in the follow-

```
1) Hash the input sequence
2) Identify candidate primaries by searching for hash hits
        2a) When a candidate is identified, call ScoreMatch()
        2b) If score < threshold
                i)   Move on to next candidate primary
            Else
                i)   Call ExtendMatch()
                ii)  Add the input sequence to the candidate cluster
                iii) Terminate the search and move on to next
                     input sequence
3) If the input sequence is not added to any cluster, it becomes the
   primary of a new cluster
```

Figure 5.3: Expanded Clustering Control Flow (line 4c from figure 5.2)

ing section. By default, the search of the candidate clusters is *greedy*. The sequence being clustered is added to the first cluster that is found to be similar. The extended search feature, first implemented in `UIcluster 2.0` can be enabled to search the entire space of candidate clusters for each sequence that is input. In this mode of operation, an additional file is output that contains a list of matching clusters for each sequence. However, the sequence is only added to the cluster that it matches best (i.e., the longest matching subsequence measured in units of bases).

5.1.2 Comparing Sequences

Comparing sequences is the fundamental operation used by `UIcluster` to cluster sequences. Before the process used for comparing sequences is described, it is important to note that DNA is largely repetitive in nature. Before clustering is performed, it is important to mask out repetitive regions so that false similarities are not identified. Sequence similarity should only be based on base regions that are unique to a particular sequence. Another related aspect of DNA that needs to be consid-

ered is low-complexity regions such poly-A tails and simple repeats. These regions contain little information and should also not be considered as evidence of similarity when comparing sequences. The input to `UIcluster` should always be masked for low-complexity and repetitive regions using a program such as `RepeatMasker` [28]. If such preprocessing is not performed, *overclustering* (i.e. merging clusters that should be disjoint) will occur.

Our sequence comparison procedure is implemented as a two-phased operation. First, the `ScoreMatch` function is called to evaluate if the similarity criteria specified by the user has been met. If it is determined that there is a match, `ExtendMatch` is called to extend the minimal subsequence match that was found by `ScoreMatch` to its longest extent while retaining the user's criteria for the maximum number of allowable errors. Calls to `ScoreMatch`, while much less complex than a score-based sequence comparison such as the Smith-Waterman algorithm [29], will still become the computation's bottleneck if it is called too often. Avoiding unnecessary calls to this sequence comparison function is the goal of our hashing optimization discussed in section 5.1.3. The next two subsections will discuss the `ScoreMatch` and `ExtendMatch` functions specifically. The source code implementing these functions is located in appendix section A.2.3.

5.1.2.1 ScoreMatch

The `ScoreMatch` function determines if two sequences share a window of N out of M bases in common. Three error modes need to be taken into account when

doing the comparison: insertions, deletions, and mismatches. The function recursively descends an alignment until either a region of N out of M bases is found or more than $M - N$ errors are found in every possible edit path.

Figure 5.4 shows an example of comparing two sequences with `ScoreMatch`. For this example, the match criteria is $N = 6$ and $M = 7$ (Note: the tree is not truncated where the $M - N$ error limit has been exceeded so that all cases can be shown and discussed). When an error is encountered there are three cases that must be checked. The left branch corresponds to a mismatch error, the middle branch corresponds to a deleted base in the first sequence (or inserted base in second sequence), and the right branch corresponds to an inserted base in the first sequence (or deleted base in the second sequence). In this example, the first three bases match exactly. When the first difference is encountered at position four, `ScoreMatch` first checks for a mismatch error along the left branch. To do this, it calls itself advancing to the next position in each sequence. The fifth positions are found to match, but the sixth positions are different. Again, `ScoreMatch` calls itself first checking for a mismatch error. Another mismatch error is found and this branch stops because the end of both sequences is reached. The recursion falls back one level and checks for a deletion in the first sequence. This fails and the end of the second sequence is reached. Finally, the right branch is taken to check for an insertion in the first sequence. This succeeds, but the end of the first sequence is reached before a score of 6 is found. The recursion then falls back to the fourth position and checks for a deletion in the first sequence.

This corresponds to the middle tree in the figure. Again, no matches with a score of at least 6 are found. The recursion again falls back to the fourth position and checks for an insertion in the first sequence, taking the right-most branch. This search is successful, finding a match of 6/7, so `ScoreMatch` returns to the calling function with the score – 6. If no acceptable match had been found, `ScoreMatch` returns the score of the best path checked. Had a match of score 6 been found earlier (e.g. in the left branch of base position 4) the function would have returned immediately.

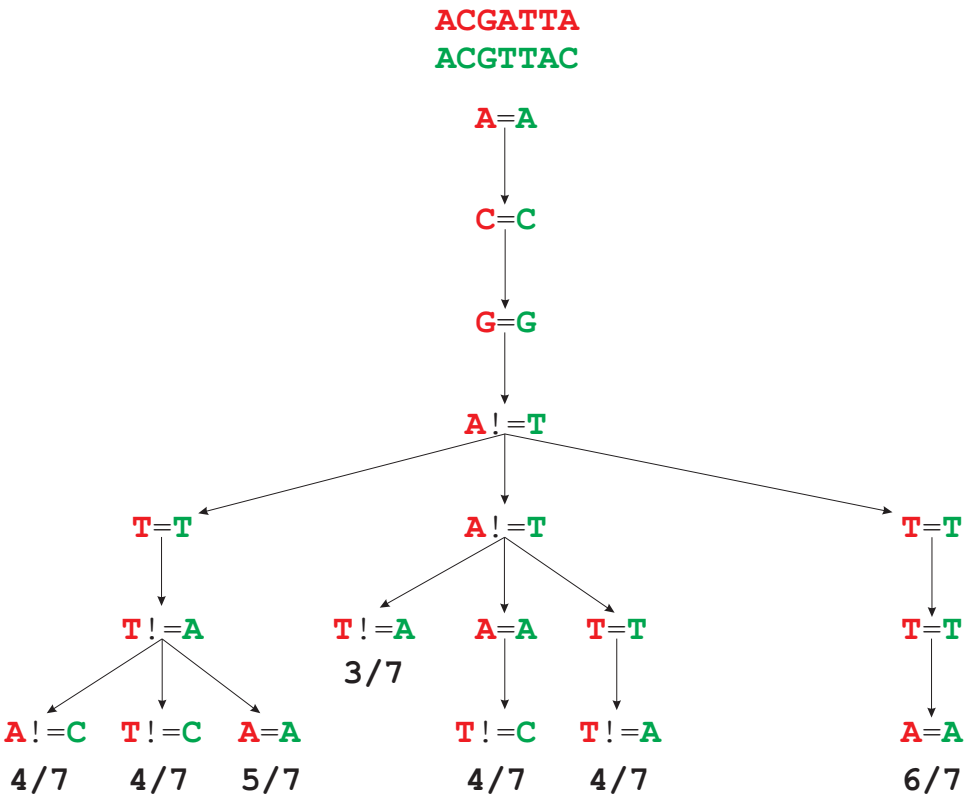

Figure 5.4: Example of `ScoreMatch` Execution

`ScoreMatch` is potentially very costly in terms of execution time, but is made

at least marginally efficient because the number of errors is bounded, allowing whole sub-trees in the search space to be eliminated. The key assumption is that the number of errors permitted will be relatively small (e.g. 95% identity for a given region). If too many errors are permitted, the search tree fans out rapidly in both the depth and width dimensions and performance suffers dramatically.

5.1.2.2 ExtendMatch

Once a match has been found by `ScoreMatch`, the `ExtendMatch` function is called to lengthen it. The approach taken is to append regions of M bases until the end of one of the sequences is reached or until a region shorter than M is can be appended. The regions are appended by repeatedly calling `ScoreMatch`. For each appended region, the users N out of M similarity criteria must hold. The extension stops once this criteria fails to be met for an appended region.

5.1.3 Hashing

All implementations of `UIcluster` use hashing techniques of various complexities to filter the search space and accelerate sequence comparisons. A *hash* is simply an integer that uniquely represents a short string of characters. In the case of DNA, the possible characters are from the alphabet $\{A, C, G, T\}$, which represent the four bases. Any sized alphabet can be used, although the maximum string length used to generate a hash may be severely limited for larger alphabets for practical purposes.

The general equation used to generate a hash is given by equation 5.1.

$$H = \sum_{i=0}^{\zeta-1} (K^i * \phi) \tag{5.1}$$

In this equation H is the generated hash value, ζ is the string length, K is the alphabet size, and ϕ is the integer value assigned to the letter at position i in the string being hashed. To generate hashes for DNA sequences, the alphabet size K is four since there are four DNA bases. Equation 5.2 gives the values assigned to each base. As a practical matter, the string length ζ that can be used to generate hashes is limited by K, and the word size of the computer. For the DNA alphabet, each base requires 2-bits to represent it ($\lceil \log_2 K \rceil$ where $K = 4$). Thus, the maximum value of ζ using a single word on a 32-bit machine is 16.

$$\phi = \begin{cases} 0 & \text{if seq[i] = A} \\ 1 & \text{if seq[i] = C} \\ 2 & \text{if seq[i] = G} \\ 3 & \text{if seq[i] = T} \end{cases} \tag{5.2}$$

When a sequence is hashed, equation 5.1 is used on every ζ length substring. Figure 5.5 shows the first six hashes generated for a sample sequence. Each of these hashes is indexed by the left-most character in the substring being hashed. There are no hashes generated for the last $\zeta - 1$ bases in a sequence. Additionally, substrings with X's or N's in them cannot be hashed. This means that for every X or N, there are at least ζ substrings that cannot be hashed. Recall, X denotes a masked base position (due to repetitive or low complexity sequence regions) and N denotes an

ambiguous base position (due to uncertainty during sequencing).

```
Sequence: GCCACTTGGCGTTTTG
Hashes:
          Hash 1: GCCACTTG = 48406
          Hash 2: CCACTTGG = 44869
          Hash 3: CACTTGGC = 27601
          Hash 4: ACTTGGCG = 39668
          Hash 5: CTTGGCGT = 59069
          ...etc.
```

Figure 5.5: Example of Hashing a Sequence

The calculation to generate the hashes for a sequence only needs to be performed once, but the hashes are accessed many times during the programs execution. This amortizes the computational overhead of generating the hashes. The actual C code that hashes a sequence is listed in section A.2.2 in function `hashSeq`.

5.2 TLcluster (Version 1)

The first implementation of the clustering program was `TLcluster`. The main features of this program were incremental clustering, repicking of primaries, and acceleration of sequence comparisons with per-primary sorted hash lists. The data structures used to represent a cluster are shown in figure 5.6. A cluster is comprised of one primary structure and zero or more secondary structures. The secondary elements are stored in a linked-list attached to the `nextS_p` pointer of the primary

structure. If the cluster is a singleton (i.e. contains no secondaries), this pointer has the value of `NULL`. The secondaries are chained together using the `nextS_p` pointer of the secondary structure. The last secondary in the linked list has this pointer set to the value `NULL`.

```
typedef struct PRIMARY {
    char             *seqName;
    char             *sequence;
    int              *hashPrefix;
    int              *hashPrefixIndex;
    struct PRIMARY   *nextP_p;
    secondary_t      *nextS_p;
} primary_t;

typedef struct SECONDARY {
    char             *seqName;
    char             *sequence;
    double           score;
    int              iP;
    int              iC;
    int              matchLength;
    struct SECONDARY *nextS_p;
} secondary_t;
```

Figure 5.6: Primary and Secondary Data Structures

The clusters are stored in memory as a linked-list of primaries. The `nextP_p` pointer of the primary structure performs the linkage. When clustering a new sequence, `TLcluster` starts at the beginning of this list and inspects every primary in order. The new sequence is added as a secondary to the first primary that it matches. If the repick primary option is enabled and the new sequence is longer than the cluster's existing primary, it becomes the new primary for the cluster. The cluster's old primary then becomes a secondary member of the cluster.

When inspecting a primary, the hashes stored in the sorted `hashPrefix` array of the primary data structure are used to determine if the sequence being clustered has any potential of being similar to the primary sequence. The hashes of the sequence being clustered are compared against the hashes of the primary sequence, and identical hash values are identified. When a matching hash is found, the indices of the corresponding hashes (i.e., the base index of the hash in the primary and the base index of the hash in the sequence being clustered) is passed to the `ScoreMatch` function. If this function determines that there is at least an N of M base match between the sequences, the `ExtendMatch` function is called and the sequence is added to the primary's list of secondaries (or is repicked as the new primary). The statistics of the match identified by the sequence comparison functions is stored in the secondary structure in the `score`, `iP`, `iC`, `matchLen` fields. These correspond to the identity score of the match (e.g., a match of 95/100 bases corresponds to 95% identity), the start base index of the match in the primary, the start base index of the match in the secondary, and the length of the match.

The hashes of a primary sequence are stored in the `hashPrefix` array in numerically ascending order. The hashes of the sequence being clustered are also stored in ascending order. Thus, to search for identical hashes between two sequences, these arrays only need to be linearly scanned once. Each array has an index counter associated with it that starts at 0 and is incremented until the last hash is inspected. When searching for identical hashes, if the hash at the current index of the primary's

hash array is less than the hash at the current index of the new sequence's hash array, then the primary's index counter is advanced. Alternatively, if hash at the current index of the primary's hash array is greater than the hash at the current index of the new sequence's hash array, then the new sequence's index counter is advanced. If the hashes being examined are identical for both the primary and the sequence being examined, then `ScoreMatch` is called. The original implementation of `TLcluster` did not store the hashes in sorted order. The entire primary array was inspected for each hash of the sequence being clustered. The last version of `TLcluster` had an order of magnitude in performance as a result of this sorting.

The length of the hash probe used is an important parameter that can significantly affect performance. Longer hash lengths will result in better performance for a given similarity criteria. It must also be chosen carefully so that potential similarities are not missed. The formula for calculating the optimal hash size is shown in equation 5.3. The rational for this equation is that for any chosen similarity criteria where M is the window size and $M - N$ is the number of permitted errors, there is at least one contiguous, error-free region of ζ bases. Thus, the comparison of two sequences can be accelerated by first searching for short exact matches of length ζ bases between the pair (i.e. searching for identical hashes). If such a match is found, a more exhaustive search that permits errors can be performed. If no length ζ hashes are identified, then the two sequences cannot possibly contain a window of M bases with N bases in common.

$$\zeta = \left\lfloor \frac{M}{M - N + 1} \right\rfloor \tag{5.3}$$

5.3 UIcluster 2.0

`TLcluster` was found to work well for moderately sized data sets (30,000 or fewer ESTs), however as the Rat EST gene discovery data sets grew, more performance was required. To accomplish this, a table was implemented in the next version of our clustering program, renamed `UIcluster 2.0`, that stores the set of cluster primaries containing any given hash value. Although this table, referred to as the *Global Hash Table*, increases memory requirements significantly (4^{ζ} lists of varying lengths proportional to the number of primaries) , it eliminates the need to sequentially traverse the list of primaries for each sequence clustered. Only primaries that contain hashes in common with the sequence being clustered are examined as candidates. Thus, performance is increased significantly. In addition to this optimization, two features were added that enable more thorough clustering – checking the reverse complement of a sequence, and performing an extended search of all primaries for each sequence being clustered. Both of these are options and can be enabled independently by the user.

5.3.1 Global Hash Table

A structural view of the global hash table (GHT) is shown in figure 5.7. In general, this table contains 4^{ζ} top level entries, each entry being a memory pointer to a linked list of cluster primaries. Only primaries discovered by indexing into this

table and traversing the corresponding linked list are considered as candidates.

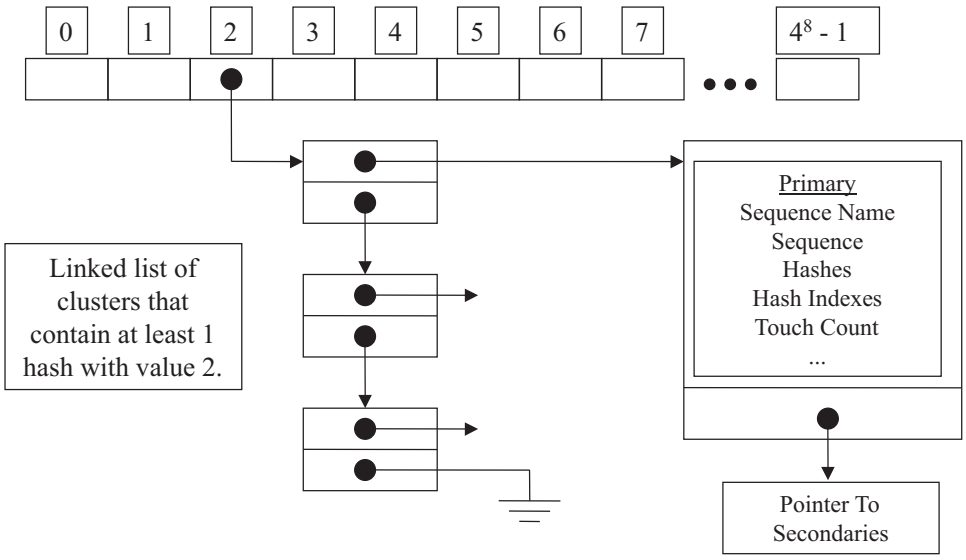

Figure 5.7: Global Hash Table

In the figure, there are 4^8 entries corresponding to a hash window size of $\zeta = 8$. On a 32-bit machine, this equates to an empty table size of 256 kilobytes (4^8 entries of 4 bytes each). In order to keep accesses to this table fast, this table is directly accessed by hash value. This is an important point since this table is accessed so frequently. Any sort of traditional hash table implementation would require extra calculations and have the potential for hash collisions. The drawback is memory usage of the table scales by 4^{ζ} for DNA sequences (alphabet size = 4). The memory required for the table when $\zeta = 16$ is 16 gigabytes. Thus, some compromises need to be made when choosing the value of ζ to use. The shorter value chosen will be referred to as ζ'. Some empirical experimentation is necessary to determine what ζ'

is most effective for a given data set and for the available memory.

Since it is typically not possible to use the optimal hash size for a given N out of M similarity criteria, a thresholding scheme was developed to lessen the trade-off in selectivity of candidates. For each sequence clustered, the GHT is traversed to identify candidate primaries that contain at least λ length ζ' hashes in common with the sequence. Instead of looking for only one ζ length hash in common as was the case in `TLcluster`, multiple shorter hashes are searched for. Equation 5.4 can be used as a guide when choosing the value of λ to use for the optimal ζ and chosen ζ'. The value of ζ' to use is a parameter defined by the user at run-time for flexibility in memory usage and performance.

$$\lambda = (M - N + 1) * (\zeta - \zeta') \tag{5.4}$$

An integer counter was added to the primary data structure to count the number of times a given primary is encountered, or *touched* while traversing the GHT for a given sequence. Only if a primary is touched more than λ times is a comparison performed against the sequence being clustered. This counter is reset to zero after every sequence processed.

Intuitively, the thresholding scheme lowers the probability that `ScoreMatch` will be called for a primary that doesn't match the sequence being clustered. Similarly, the GHT clearly has the potential to eliminate many of the failed calls to `ScoreMatch` in `TLcluster`. In practice, the GHT optimization has been highly effective – usually

improving performance by factor of at least 30 over `TLcluster`.

5.3.2 Extended Search

Due to the use of `TLcluster` in our sequence processing pipelines, it was determined that there are certain situations where checking all of the primaries for each sequence clustered is desirable. Directly modifying `TLcluster` to do this would have been relatively straight-forward, however performance would have degraded significantly. Instead of stopping on average half way through the traversal of the primary linked list for sequences that become secondaries, the remainder of the list would need to be traversed.

This property was accomplished in practice by the use of the `GHT`. Performance is only moderately increased because the search space is filtered into a relatively short list of candidate primaries for each sequence clustered.

5.3.3 Reverse Complement Checking

A common error in sequencing DNA causes the wrong strand to be sequenced in the opposite direction. To detect this, a feature was added to check the reverse complement of a sequence being clustered in addition to checking it in its original form. If the extended search option is enabled, the reverse complement of a sequence is always checked for similarity to the cluster primaries. Otherwise, it is only checked for sequences where no similarity was found in the sequence's original orientation.

To generate the reverse compliment, the character string representing the sequence being clustered is copied into a working buffer. This buffer is then reversed

in-place by successively swapping bases. After reversal, the sequence string is complemented according to DNA pairing rules (i.e $\{A \rightarrow T\}, \{T \rightarrow A\}, \{G \rightarrow C\}, \{C \rightarrow G\}$). The C code that implements these steps is found in the function `revComp` listed in appendix section A.2.9. The resulting sequence string is then hashed and processed in exactly the same way as the original sequence.

5.3.4 Additional Minor Changes

Two additional changes are worth mentioning. The Linux operating system, or rather the library that implements the `malloc` function [30], has limits on the number of memory blocks that can be allocated. This caused `UIcluster` to crash for large data sets without explanation. Considerable time was spent debugging the program in order to identify the cause of the crash. Once the `malloc` limitation was discovered, custom memory allocation routines were implemented as a solution. These functions are listed in appendix section A.2.6. The vast majority of the calls to `malloc` were identified to be of a limited number of sizes. Indeed, this is because entries in the GHT are all of the same size and are allocated separately. Other data structures in the application have this property as well. The memory allocation functions operate by allocating a large chunk of memory that is a multiple of the size in bytes of a given data structure. This, along with some bookkeeping, enables many structures to be allocated with a single call to `malloc`. This solved the crashing problem and was instructive as to the issues that arise when working with large data sets.

The second change was a result of problems encountered when clustering long

sequences. `TLcluster` was implemented to use a crude "bubble sort" algorithm for sorting hash arrays. While this was sufficient for the average EST sequence length (400-800bp), this became a bottleneck for longer sequences such as full-length cDNAs and genomic contigs. The *quicksort* algorithm was implemented to work around this problem. The algorithm used was adapted from [22] to also carry along the hash index array. The source code for this routine is listed in appendix section A.2.8.

5.4 UIcluster 3.0

The latest version of `UIcluster` improves on its predecessor by adding parallel execution capability and the notion of virtual primaries. These features are discussed in the following sections.

5.4.1 Parallel Execution

The parallel execution functionality is implemented using the MPICH MPI library [19]. The number of UNIX processes to use is a parameter specified by the user at run-time. If one process is chosen, the program operates identically to `UIcluster 2.0` if the virtual primary generation option is not enabled (discussed in the next section). If more than one process is chosen, the program's execution and memory requirements are spread across multiple UNIX processes and processors. If the computing hardware being used is an SMP machine, each of these processes can use a separate CPU. Alternatively, if the program is operating in a clustered environment, such as a Beowulf class system [4], the processes are spread across multiple distributed computers.

The only change to the clustering algorithm is a series of collective communications between processes after the GHT has been searched and the matching primaries identified. The list of matching primaries for each node is sorted by score and the best match is communicated to all other nodes. If no match is found, then the sequence being clustered becomes the primary of a new cluster and is assigned to a single process, determined by the process ID equal to the modulus of the cluster ID and the number of processes. In this way, the clusters are evenly spread across processes and the memory requirements are reduced accordingly (assuming each process also gets assigned approximately the same number of secondaries).

If a similarity to a cluster is determined, the process with the best score is determined and the sequence is added to the best matched cluster on that process.

5.4.2 Virtual Primaries

The virtual primary generation feature of the latest version has been implemented using the `bl2seq` [31] program to identify the extent of the overlap between the current virtual primary of a cluster and the sequence being added to the cluster. `bl2seq` uses the `BLAST` [3] algorithm to align two sequences and assign a score to each of the matching regions identified. It is, however, considerably slower than the sequence comparison functions in `UIcluster`. The reason for using it instead of the `ScoreMatch` and `ExtendMatch` functions is two-fold. First, it is more accurate in determining the length of a match. It reports the end base of a match for both sequences being compared, instead of a single match length parameter as does

`ScoreMatch`. This is important when aligning a new sequence to a virtual primary to see if it can be extended. `ExtendMatch`, because it uses a heuristic for speed (repeatedly calling `ScoreMatch`), and sometimes misses some bases at the end of a match. The second reason for using `bl2seq` is to identify cases where there are multiple matching regions. `ScoreMatch` and `ExtendMatch` only identify one match. This is useful for identifying sequences possibly representing alternative splice forms of a gene. Future versions of `UIcluster` may implement the necessary functionality of `bl2seq` internally. It was used for the purpose of speeding development of the virtual primary generation feature.

When a sequence is added to a cluster, the `bl2seq` function is called to compare the new sequence to the cluster's virtual primary. The source code to call `bl2seq` and parse the results is listed in appendix section A.2.10. A list of matches sorted by score is returned to the caller. Each match in the list contains the start base in the virtual primary, the start base in the new sequence, the end base in the virtual primary, the end base in the new sequence, the score of the match (match length in the virtual primary), and the direction of the match. This information is then used by the `addSecondary` function (listed in appendix section A.2.2) to determine if the virtual primary can be extended by the new sequence. Five cases are detected: bases can be added to the front of the virtual primary, bases can be added at the end of the virtual primary, bases can be added at both the front and end of the virtual primary, the new sequence is entirely contained in the virtual primary, and

the new sequence partially hits the virtual primary but contains significant regions that are inconsistent with it. If an extension is possible, the added bases from the new sequence are copied into the virtual primary buffer. If bases are added to the beginning, the existing virtual primary is shifted to the right in the buffer. The resulting sequence is then hashed and added to the GHT. The hashes of the previous virtual primary are removed before this is done.

When the virtual primary buffer is first created, twice as much memory than necessary is allocated. When an extension would cause this buffer to be exceeded, the buffer is doubled again with a call to the `realloc` function [30]. By allocating more space than necessary, the number of `realloc` calls is reduced, thus reducing memory fragmentation.

The latest release of `UIcluster` is the first version that incorporates the virtual primary generation feature. Over time, the usefulness of this feature will be evaluated more extensively than has been done to date. If it is determined to be useful, its performance will be improved in future versions of the clustering application.

5.5 Running UIcluster 3.0

This section is intended to be instructive in compiling and using the latest version of `UIcluster`. The application is available in source code form from our project web site (http://genome.uiowa.edu). The previous releases of the application, `TLcluster 1.0` and `UIcluster 2.0`, and accompanying documentation are also available from this site. The procedures for running `UIcluster 2.0` are essentially

identical to what is presented in this section with the exception of the virtual primary and parallel execution features.

5.5.1 Compiling

Once the source code to the program has been downloaded from the Internet, it needs to be compiled into an executable program before it can be used. To do this, the source distribution must be uncompressed, configured, and compiled with a C compiler. The user performs all of these steps entering commands at a UNIX command-line.

An MPI library must be installed on the UNIX computer being used before `UIcluster 3.0` can be compiled. The program was developed using the freely available MPICH MPI library. This library is available for the UNIX platforms (Linux, Sun, HP) `UIcluster` supports. Instructions for obtaining and compiling MPICH are available on the Internet (http://www-unix.mcs.anl.gov/mpi/mpich). Other MPI implementations should work, however they have not been thoroughly tested. The compilation of `UIcluster 2.0` does not require an MPI library. If the user does not need the parallel execution and virtual primary generation features of `UIcluster 3.0`, then version 2.0 should be used. Other than this requirement, the procedure for compiling the two versions is the same.

The UNIX commands necessary to build the UIcluster executable are given in figure 5.8. The first step is to decompress and un-archive the distribution by executing the first two commands in the figure at a UNIX shell prompt. The next

step is move into the main source directory by using the third command. The fourth step configures the files necessary to build the application. Finally, typing "make" builds the executable. After a successful build, the executable will be a file called `uicluster`. It may be copied to a location in the user's path so it can be executed from anywhere on the system.

```
1) gzip -d UIcluster-3.0.tar.gz
2) tar -xvf UIcluster-3.0.tar
3) cd UIcluster-3.0
4) ./configure
5) make
```

Figure 5.8: UNIX Commands for Compiling `UIcluster`

5.5.2 Command Line Options and Usage

Figure 5.9 shows the command line options input into the program. The meaning of most of these has already been discussed in this chapter, however the names may be slightly different and/or abbreviated. With the exception of the input sequence file, all parameters are optional for the user to specify. The default value for each option is shown in the right column of the figure. Both short and long option names are available for each parameter, shown in the left column of the figure.

The `--preClus` option takes as its argument a file containing a list of files, one per line, of previous clustering results to use for incremental clustering. The `--rejectCrit` is specified as the minimum number of bases required to cluster an in-

```
UIcluster 3.0.4 Usage:   uicluster [options] input_fasta_file

  Valid Options: (defaults are in parenthesis)
      -F, --preClus     specifies the preClustered infile    (none)
      -R, --rejCrit     specifies the rejection criteria  (100 bases)
      -H, --hashSize    specifies the hash size           (  8 bases)
      -S, --startSkip   specifies the start skip          ( 18 bases)
      -s, --endSkip     specifies the end skip            (  0 bases)
      -M, --matchLen    specifies the length to match     ( 40 bases)
      -E, --errLimit    specifies the error limit         (  2 bases)
      -C, --maskChar    specifies the mask character         ('X')
      -h, --hitThresh   specifies the hit threshold           (16)
      -P, --wrongPen    specifies the wrong penalty            (1)
      -p, --gapPen      specifies the gap penalty              (1)
      --repick          repick cluster primaries            (off)
      --tryRevC         check reverse compliment            (off)
      --keepGoing       perform exhaustive search           (off)
      --vPrimary        generate virtual primary            (off)
      --help            view this message
```

Figure 5.9: `UIcluster 3.0` Command-line Interface

put sequence. The `--hashSize` specifies the value of ζ' to use. `--startSkip` specifies the number of bases to disregard at the beginning of a sequence. This is useful when there is a poly-T tail still present in 3' EST data. The `--endSkip` similarly specifies the number of bases to skip at the end of a sequence. The `--matchLen` specifies match window, M, to use and `--errorLimit` specifies the number of errors to allow, $M - N$. The `--maskChar` parameter designates the character that will be used to identify low-complexity and ambiguous regions. The `--wrongPen` and `--gapPen` designate penalties to use for gaps (i.e. inserted and deleted bases) and mismatched bases in the `ScoreMatch` function. The `--repick` flag enables the repicking of primaries as discussed in section 5.2. The `--tryRevC` flag enables reverse complement checking of input sequences as discussed in section 5.3.3. The `--keepGoing` flag enables the extended search capabilities discussed in section 5.3.2. Finally, the `--vPrimary` option turns on the virtual primary generation feature for each cluster as discussed in

section 5.4.2.

If parallel execution of the program is desired, the `mpirun` program must be used to launch the executable on multiple compute nodes simultaneously. The "-np X" (where X is a number) argument of this program is used to designate how many compute nodes to use. For example, to run `UIcluster 3.0` on 8 compute nodes with the default options and an input sequence file named *seqs.fasta* the command is *"mpirun -np 8 uicluster seqs.fasta"*. To execute the program serially the command would have been simply *"uicluster seqs.fasta"*. In addition to using `mpirun` for parallel execution, all of the input files need to be available on each of the compute nodes (e.g. cross-mounted using the network file system (NFS) protocol).

5.5.3 Output File Format

The cluster files output by all version of the clustering application are formatted in essentially the same way. The new clusters are output in a file named the same as the input sequence file with an *".clus"* extension (e.g. the cluster file output for an input file named input.fasta is input.fasta.clus). In the incremental clustering mode, a file with an *".out"* extension is output each cluster file input into the program.

Each cluster file contains one or more clusters, each cluster being comprised of one or more sequences. For each cluster, the cluster primary is given first and is signified by a line starting with "`@P:`" followed directly by the sequence's name. The sequence string starts on the next line. The sequence is printed as seventy bases per line. Any secondary sequences belonging to the cluster follow directly after the

primary, and are formatted in the same way except that each sequence definition starts with "`@S:`". If the virtual primary option is enabled, it is located between the primary and the first secondary sequence. Its sequence definition starts with "`@VP:`".

In addition to the cluster output files, there are two accompanying files output. The *rejects* file (input sequence file-name with the "*.rej*" extension appended) contains sequences that were rejected from the clustering. The format of this file is a numbered list of sequences and is fairly self-explanatory. The *hits* file (input sequence file-name with the "*.hits*" extension appended) contains the list of clusters that were found to contain a match meeting the users N out of M similarity criteria for each sequence. In `UIcluster 3.0`, each line in this file is formatted as a sequence name followed by a list of tuples of the form *cluster_ID.match_score:direction* representing the matched cluster for that sequence. The direction field is either "f" or "r", indicating forward and reverse complement matches. A sample entry of this file listing two matched clusters (i.e., cluster ids 4 and 98) is "`UI-R-A0-ae-e-12-UI: 4.378:f 98.175:r`".

CHAPTER 6
RESULTS

This chapter presents results obtained by utilizing the `UIcluster` clustering application discussed in this thesis. The first two sections of the chapter discuss two important uses of the application – novelty assessment and gene index creation. The third section of the chapter compares the University of Iowa's rat gene index created by `UIcluster` and NCBI's rat UniGene index. Finally, the last section reports on the measured performance and memory usage of the various versions of `UIcluster`.

6.1 EST Sequencing Novelty Assessment

`UIcluster` was originally developed for the purpose of assessing 3' EST sequencing novelty rates, roughly corresponding to the gene discovery rate. The program has been used in the production sequencing pipelines of several projects underway in our laboratories at the University of Iowa [11] and at other institutions (KAIST, Korea, Washington University, St. Louis, UNL, Lincoln, MCW, Milwaukee, among others). Equation 6.1 states the equation used for calculating percent novelty.

$$\% \text{ Novelty} = \frac{\#\text{ clusters}}{\#\text{ sequences}} * 100 \tag{6.1}$$

This equation is utilized to calculate incremental and overall novelty rates

for individual libraries and for projects as a whole. Incremental novelty calculations are performed daily to monitor the sequencing efforts and to determine when library subtractions and/or normalizations should occur [5]. Both of these procedures have been proven to dramatically increase novelty rates. However, they are time consuming and cannot be performed on a continual basis.

Figure 6.1 shows an example of the effectiveness of these procedures for a progression of four cDNA libraries, named C0, C1, C2p, and C3. More details can be found in [25].

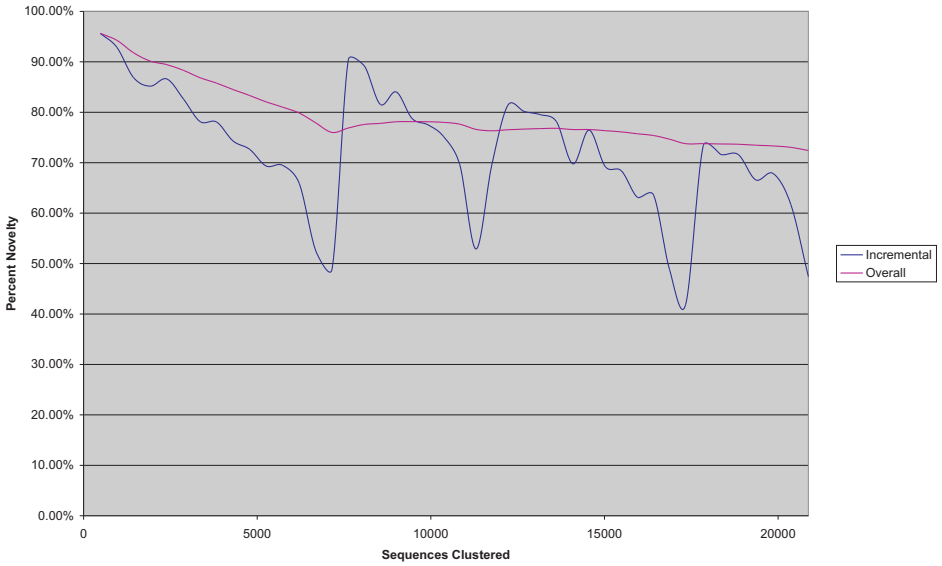

Figure 6.1: Incremental Library Novelty

C0, the first in the progression, was sequenced to obtain roughly 7,000 ESTs. This is shown in figure 6.1 as the left-most downward trend. To increase the novelty

rates, the C1 library was created by removing previously seen cDNA transcripts from the C0 library. Sequencing from this library is depicted in the second downward trend in the figure (7,500–12,000). Again, over time the sequencing from this library became too low and the C2p library was created from it. This is shown in the third downward trend in the figure (12,000–17,000). Finally, the C3 library was created from C2p to improve novelty rates (17,000–21,000). For each successive library, the incremental novelty rates steadily decrease since the redundancy removing procedures are not perfect. It should also be noted that the large drop-offs in novelty immediately before each library is a unique characteristic of this data set and will probably not be observed for other library progressions.

6.2 Generation of Gene Indices

A related task to novelty assessment is gene index creation. Ideally, running `UIcluster` on a set of sequences will assess novelty and generate a gene index (or UniGene set as it is popularly referred to). Each cluster will represent a gene, and the total number of clusters divided by the number of sequences will represent the novelty of the sequences clustered. Such indices are essential for picking cDNA clones to use in the radiation hybrid mapping efforts under-way at our laboratory.

The specific procedures that are used to generate our UniGene indices are constantly being refined. The similarity criteria that we have most recently been using is a matching window of 38/40 (i.e. $N = 38, M = 40$) bases between two sequences for them to be put into the same cluster. Over time, the repicking of

primaries and the reverse complement checking features have been incorporated into our UniGene build procedure. At the time of writing this thesis (April 9, 2001), our Rat Gene Discovery UniGene index contains 62,296 clusters, 213,372 sequences. Our Human Cancer Genome Anatomy Project UniGene index contains 29,509 clusters (40,684 sequences), and our Mouse Brain Molecular Anatomy Project contains 37,983 clusters (88,844 sequences). Current statistics on our UniGene indices can be obtained from our project web site (http://genome.uiowa.edu).

Recently, the number of genes estimated to be in the human genome has been reduced from 100,000 to 30,000-40,000 [17]. Most other higher-level mammalian organisms are expected to have similar numbers of genes. This is causing us to revise our UniGene build procedure because, for example, our 62,296 rat clusters (genes) seems to be much too high (*underclustered*). We suspect low-quality ESTs and other sequencing errors (alternative polyadenylation, internal restriction sites, and internal priming) to be the cause of this. As a first attempt at eliminating such contamination, we have formed a Rat UniGene set by only counting clusters that contain one or more sequences with both the tail and signal features present. These features are identified in an EST sequence by the `estPrep` application that our laboratory has developed [11]. When this criteria is applied, 23,902 clusters remain in our rat UniGene index. We are currently evaluating the effectiveness of this change. However, the reduced number of clusters seems to be more consistent with the revised gene estimates. An additional change planned for the future is to utilize the draft

human genomic sequence to verify clusters. This is possible because most rat genes ($> 98\%$) are also present in the human genome. If all sequences in a cluster match to a localized region of the human genome (e.g., within the same 10,000 base region) then that cluster is likely to represent the same gene. Clusters with sequences matching distant regions of the genome should be looked at with suspicion.

6.3 Accuracy Assessment

Comparisons between our rat UniGene index and NCBI's UniGene index [27] will be used to assess the accuracy of our sequence-similarity-based clustering. Additionally, the sequence assembly program phrap [13] will be used to assemble our clusters into consensus sequences. Instances where the sequences in a cluster assemble into one consensus (contig) provides additional evidence that the sequences represent the same gene.

6.3.1 Comparisons to NCBI UniGene

A set of Perl [35] scripts were developed to compare our rat clustering results to NCBI's rat UniGene (ftp://ftp.ncbi.nlm.nih.gov/repository/unigene). In order to obtain the most fair comparison, the 128,229 University of Iowa ESTs contained in this index were extracted and put into a FastA formatted file. This number is reduced from the 213,372 sequences stated earlier because NCBI requires the tail feature to be present in an EST for it to be used in their clustering. Other minor EST selection criterias also differ between our methods.

The latest version of `UIcluster` was used to cluster this data set. The param-

eters used were $N = 38, M = 40, \zeta = 8, \lambda = 15$ and the repick primary and reverse complement checking options were enabled. Running `UIcluster` resulted in 41,726 clusters. Three types of cluster relationships were then determined by the Perl script – clusters matching between the `UIcluster` clustering and NCBI's clustering and clusters that are split into one or more clusters between the two. These relationships are shown graphically in figure 6.2. The case where a `UIcluster` cluster is totally contained in a single NCBI cluster (e.g., in the figure there is an extra sequence in the NCBI cluster for the matching clusters case) is considered to be matching.

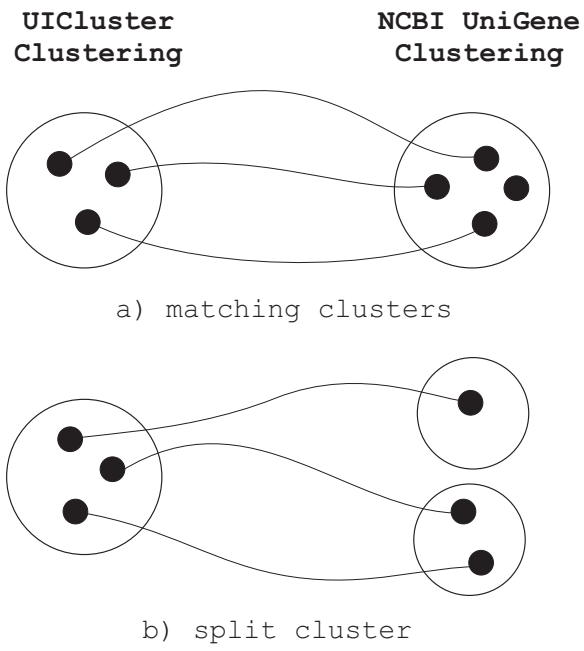

Figure 6.2: Comparing Clusters

The analysis of the 41,726 clusters generated by `UIcluster` showed that 39,165 clusters (93.9%) matched an NCBI cluster. The remaining 2,561 clusters (6.1%) were

split into multiple NCBI clusters. Performing the comparison in the opposite direction for the 41,522 NCBI clusters containing University of Iowa ESTs resulted in 38,890 (93.7%) NCBI clusters directly matching a `UIcluster` cluster. The remaining 2,632 (6.3%) NCBI clusters were split among multiple clusters in the `UIcluster` clustering.

6.3.2 Analysis of Cluster Assemblies

Another set of Perl scripts were developed to generate cluster consensus sequences for each of our clusters using the `phrap` [13] sequence assembly program. Ideally, each cluster should assemble into one contig (i.e., a consensus sequence that every sequence in the cluster aligns to) since all of the sequences should represent the same 3' UTR. In practice, clusters containing sequences from multiple transcripts of the same gene will assemble into more than one contig. Clusters containing sequences that shouldn't belong will also cause multiple contigs to be produced.

The script's analysis of our refined 23,902 cluster rat UniGene index (i.e., the index discussed at the end of section 6.2 that was generated by only using ESTs with the tail and signal features present) shows that of the 13,334 non-singleton clusters (i.e., clusters containing only one sequence), 8,362 (62%) assemble into one consensus sequence that represents all of the sequences in the cluster. These clusters are likely to represent true genes. The remaining 37% assemble into more than one consensus sequence. Automated methods for classifying the causes of these cases are currently being developed. However, hand examination is showing that the vast majority appear to be instances of multiple splice forms being present in the same

cluster. Other causes include alternative polyadenylation, internal not sites, chimeric sequences, and internal priming. This analysis is somewhat encouraging considering that it is estimated that between 30–40% of human genes contain multiple splice forms. However, splicing variations are thought to usually not occur in the 3' UTR. Thus, further inspection by expert biologists is needed to gain more understanding of the multi-consensus clusters.

6.4 Performance Assessment

The performance of the versions of `UIcluster` presented in this thesis is discussed in this section. While it is impossible to examine the entire parameter space of the program, an effort has been made to present the most important performance metrics. All of the performance results obtained in this section were obtained using a set of 16 dual 500MHz Pentium III computers. Each computer (compute node) contained either 1 gigabyte or 2 gigabytes of memory. Gigabit Ethernet (1000 megabits per second) was used for the communication network.

6.4.1 Execution Time

For serial execution, the largest performance increase was realized with the introduction of the global hash table in `UIcluster 2.0`. Figure 6.3 shows the performance difference between `TLcluster` and `UIcluster 2.0` for clustering 80,766 rat EST sequences with a similarity criteria of $N = 38, M = 40$. For this test, $\zeta = 8$ and $\lambda = 15$ were used with `UIcluster 2.0`. These are the parameters that our production pipelines currently employ. Other parameters will produce differing lev-

els of performance gain, however results similar to those presented in this figure are typically observed. The serial performance of `UIcluster 2.0` and `UIcluster 3.0` is essentially the same.

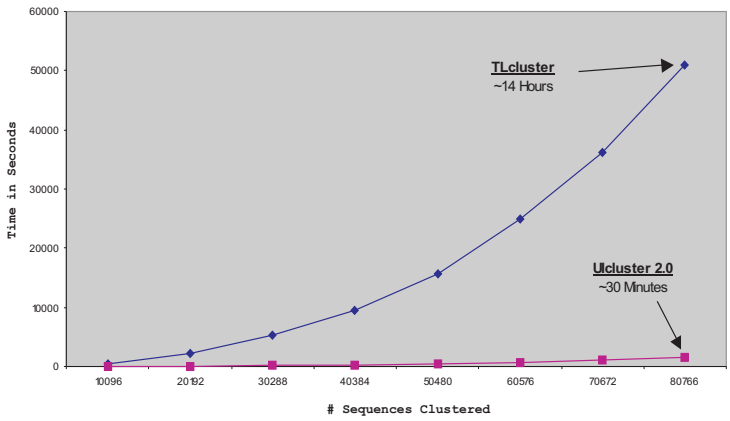

Figure 6.3: TLcluster vs. UIcluster

Figure 6.4 illustrates the parallel speedup obtained by `UIcluster 3.0`. Since the implementation uses a collective communication at the end of every sequence, the amount of computation required for each sequence is important. As the grain size increases, better performance should be observed since relatively less communication is being performed. The first curve (labeled 1) corresponds to the default parameters used in our pipeline. The second curve (labeled 2) adds the extended search option. The third curve (labeled 3) adds the reverse complement checking.

Performance actually decreases from the serial case with two compute nodes for the first case. This is probably due to the computation not being distributed

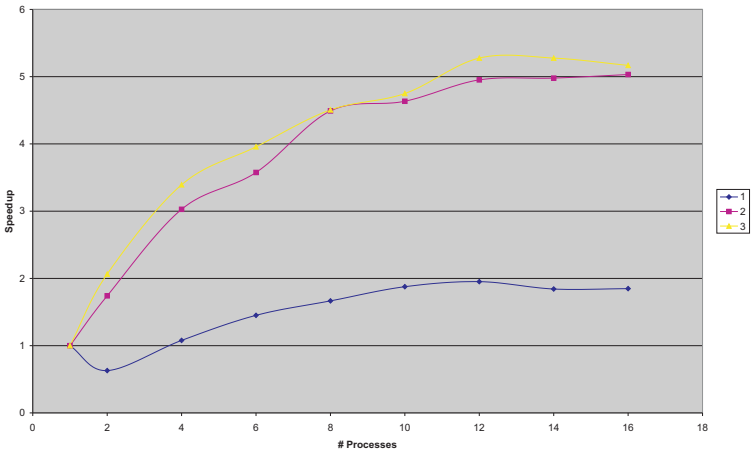

Figure 6.4: Parallel Speedup

evenly and the added cost of the communication. As the computation is spread across more compute nodes, performance increases somewhat but is never more than double the performance of the serial case. The single node execution time of this case is approximately 12 minutes and the execution time of the 16 node run is approximately 7 minutes.

Enabling the extended search of primaries significantly increases the realized speedup. The computation scales well up to 8 nodes and then levels off. For this curve, the execution time of the single node run was approximately 38 minutes and the execution time of the eight node run was approximately 8.5 minutes.

The third curve scales approximately the same as the second. The execution time of this case of the single node run is approximately 48 minutes and the execution time of the eight node run is approximately 11 minutes.

6.4.2 Memory Usage

The introduction of the global hash table in `UIcluster 2.0` greatly increased the memory requirements of the application. For a set of 82,624 ESTs, `TLcluster` required about 100 megabytes of memory. For the same data set, both versions of `UIcluster` required roughly 171 megabytes of memory.

Figure 6.5 shows how memory usage scales for the same data set with `UIcluster 3.0`. The memory requirements scale fairly linearly for increased numbers of compute nodes, which suggests that the approach of distributing an equal number of clusters to each compute node works well.

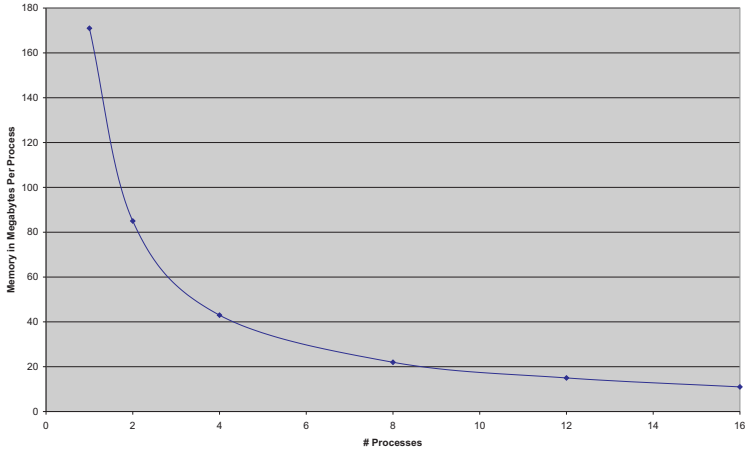

Figure 6.5: Parallel Memory Scaling

Since user programs are limited to addressing 2 gigabytes of memory with Linux, the maximum number of ESTs that can be clustered is limited to approximately 1 million sequences. For larger data sets, memory becomes an issue when

executing serially. `UIcluster 3.0` has been used to successfully cluster a data set of 1,956,525 mouse EST sequences. Performing this clustering with 16 compute nodes required approximately 300 megabytes per compute node and 18 hours, 34 minutes of compute time.

6.4.3 Parameter Variation

Figure 6.6 shows how enabling various options effects the execution time of `UIcluster 2.0` (and thus `UIcluster 3.0` when executing with a single compute node) for a data set of 82,624 ESTs. All of the cases in top half of the figure were used with the similarity criteria $N = 38$ and $M = 40$ and the parameters $\zeta = 8$ and $\lambda = 15$. The "default" option means that `UIcluster` was run without the repicking of primaries (i.e., "Repick"), reverse compliment checking (i.e., "TryRev"), extended searching (i.e., "Ext"), and virtual primaries generation (i.e., "vPrim") options enabled. The virtual primary generation option increases execution time the most. This is because calling the external `bl2seq` program, used to implement the option, involves considerable overhead. Various option combinations are also included in the table. It should be noted that the execution times of these option combinations are not simply the execution times of the individual options added. This is because options can effect one another (e.g., enabling the extended search option and the reverse complement checking option means that the reverse complement is always generated and checked for each input sequence This increases the overhead for each input sequence considerably).

Parameters	Time in Minutes
Default	11.2
Repick	15.21
TryRev	23.4
Extended	36.2
Virtual Primaries	79.1
TryRev + Ext	92.33
TryRev + Ext + Repick	118.2
TryRev + Ext + Repick + vPrim	149.75
ζ'=6 λ=21	> 24 hours
ζ'=7 λ=18	440.17
ζ'=8 λ=15	36.6
ζ'=9 λ=12	9.46
ζ'=10 λ=9	5.32
ζ'=11 λ=6	5.3
ζ'=12 λ=3	5.26
ζ=13 λ=1	5.18

Figure 6.6: Effects of clustering options on execution time

The bottom half of the figure gives the run-times of the program using different values of ζ' and λ with the same similarity criteria as before ($N = 38$, $M = 40$). The run-time using the optimal value of $\zeta = 13$ is also given (calculated by equation 5.3). For each ζ', λ is calculated by the formula given in section 5.3.1 (equation 5.4). When $\zeta' = 6$, the empty global hash table uses only 16 kilobytes of memory. However, performance is very poor because too many false candidate primaries are identified. When $\zeta = 13$, the global hash table uses 256 megabytes of memory. However, approximately the same performance can be obtained by using $\zeta' = 10$ and $\lambda = 9$. This is because for $\zeta' = 10$ or greater, the global hash table optimization is nearly 100% effective in filtering the search space down to only true candidate primaries. Thus, the `ScoreMatch` function is called the minimal number of times. For $\zeta' = 10$, the

empty global hash table uses a reasonable 4 megabytes of memory.

CHAPTER 7
CONCLUSION AND FUTURE WORK

This thesis has presented a software tool for genetic sequence clustering. It has the characteristics of high-performance, accuracy, and flexibility. `UIcluster` has proven its robustness and utility by its use in several large-scale gene discovery projects at the University of Iowa. Additionally, the flexibility of `UIcluster` has allowed it to be useful for many applications beyond its initial intent of 3' EST clustering.

However, there is still room for improvement. The following sections provide brief overviews of some of the more significant enhancements proposed for UIcluster.

7.1 Alternative Transcript Identification

Currently, sequences that may be candidates for alternative splicing are marked for later inspection by a human operator. More advanced techniques could be incorporated into `UIcluster` that examine clusters and attempt to identify exon boundaries and alternative transcripts. When available, these techniques could make use of genomic sequence data to accurately order the exons that are identified.

7.1.1 Without Genomic Sequence

Currently, the virtual primaries created for a cluster will only represent a single transcript. Sequences added to a cluster either are consistent with the virtual primary or they are flagged as problem sequences for later inspection. Such sequences may represent different transcripts of the same gene, possibly containing exons that are not present in the current virtual primary. For example, the comparison of a sequence to a virtual primary may contain a matching region, followed by a non-matching region, followed by another matching region. The non-matching region possibly represents an exon not currently in the virtual primary. This region could be inserted into the virtual primary and the boundaries could be noted as exon boundaries. The resulting virtual primary created by this type of procedure will contain all of the exons present in a cluster, but they may not be in the order that they occur in the genomic sequence.

7.1.2 With Genomic Sequence

Now that the genomes of several organisms have been completely sequenced, the is tremendous opportunity to use genomic sequence along with clustering. This information can be used to resolve ambiguous exon orderings and verify exon boundaries. For example, the virtual primary of a cluster may contain four identified exons, labeled A, B, C, and D. One transcript in the cluster may be spliced as ABD. Another transcript in the cluster may be spliced as ACD. Given these observations only, the virtual primary could be represented as ABCD or ACBD. Without genomic sequence, the order of exons B and C is ambiguous. The genomic sequence provides a means

to resolve the order of these exons. One possible approach would be to take a cluster virtual primary and find the region of the genome that contains it using a program such as BLAST [3]. This region could then be used to resolve any ambiguous exon orderings.

7.2 Confirming Gene Predictions

The exons identified by `UIcluster` in the virtual primaries can be useful for verifying the accuracy of gene prediction programs such as `GenScan` [7] and GRAIL [14]. Gene prediction programs are based on generalized models of genes and can often make mis-predictions. The empirical observation of an mRNA transcript verifies that a predicted gene actually exists and that a particular transcript of that gene is truly expressed (i.e., it is a splice form that can be produced during transcription). Additionally, the exon boundaries identified in the virtual primary verify that the predicted exon boundaries are correct.

7.3 Manual Curation

A high quality UniGene index requires human intervention to resolve ambiguities that arise during automated clustering. NCBI's UniGene indices, for example, are curated and updated by a human operator as new information becomes available. This produces a more accurate clustering and a better estimate of the number of genes discovered.

UIcluster's incremental clustering capability could provide similar capabilities by carrying curation decisions through clustering iterations. However, there are cur-

rently no user-friendly tools available to facilitate this. Ideally, such an application would incorporate all information output by the program including the match information, the extended search data, and the virtual primaries. Additionally, outside information such as genomic sequence and annotations could be incorporated. This information could all be combined to allow a human operator to resolve ambiguous situations and fix errors discovered in the clustering results. Additional tools such as `BLAST`, `phrap`, and `GenScan` could be selectively used by a human operator to make more informed curation decisions.

A cluster viewer has been implemented by the author during the course of developing `UIcluster` to better visualize the composition of clusters. This tool, written in Java [16] and shown in figure 7.1, could be extended to perform the features discussed in this section. The output of the program might be a file containing curation decisions that can be input into `UIcluster` when an incremental clustering is being performed. `UIcluster` would apply these decisions before clustering any new sequences.

7.4 Cluster Merging

If an input sequence is found to be similar to more than one cluster primary, this provides evidence that the matched clusters should possibly be merged. However, merging clusters automatically due to linkage by only a single sequence is probably not the approach to take. The cluster curation tool (discussed in section 7.3) could present such cases to an expert biologist, who could then decide if the identified clusters

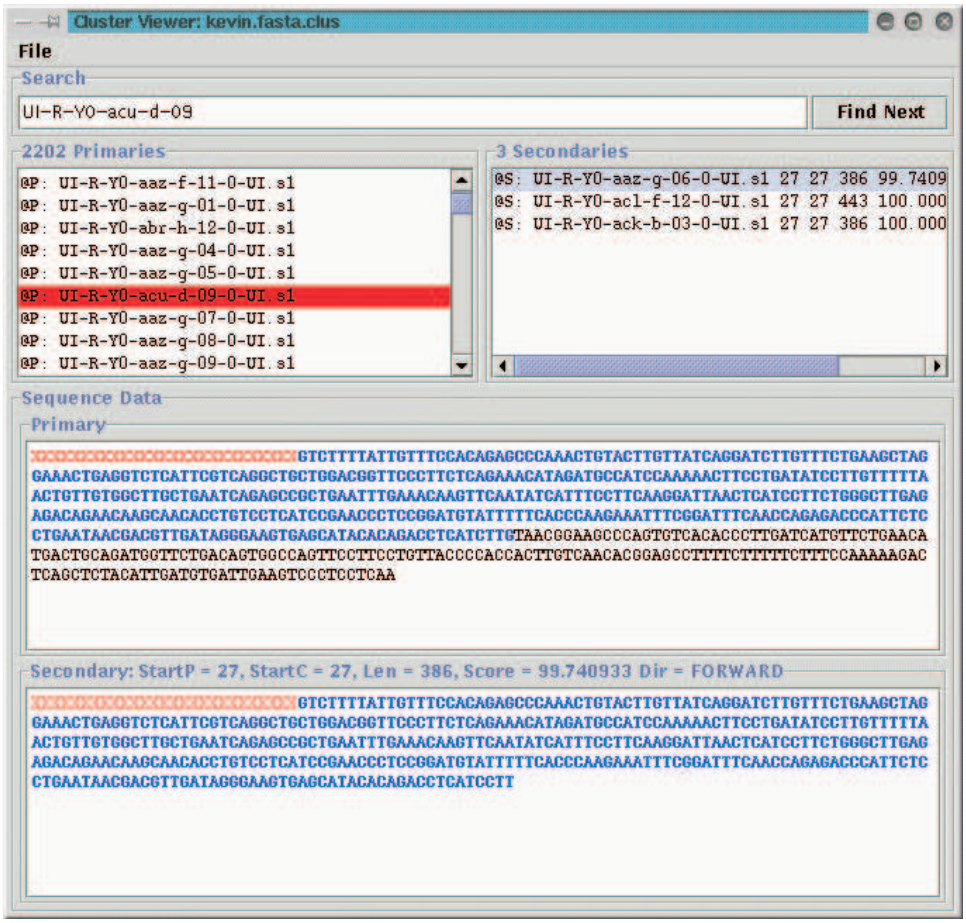

Figure 7.1: Cluster Viewer

should be merged. Alternatively, clusters that are linked by multiple input sequences might be automatically merged by `UIcluster`. A threshold for the minimum number of input sequences linking two clusters before they are automatically merged could be a parameter specified by the user at run-time. Merging clusters complicates the parallelization approach that has been utilized in `UIcluster 3.0` since clusters may need to be moved between compute nodes. A possible alternative approach is to use the incremental clustering capabilities of the application to iteratively cluster the

virtual primaries. If two virtual primaries are found to be similar, then the clusters that they represent could possibly be merged. After several iterations, steady state will be reached and no additional clusters will be merged.

7.5 Long Transcribed Sequences

The hashing techniques that we have employed are most useful for short ESTs (400–1000bp). The approach is also sufficient for full length cDNA sequences (1000–5000bp) and other long transcribed sequences. However, performance is degraded significantly when clustering such sequences. This is thought to be due to the increased probability of finding matching hashes between longer sequences. Additionally, the probability of multiple hash matches being widely separated is greater. This reduces the effectiveness of the thresholding scheme we have used (discussed in section 5.3.1). A possible solution to this problem is splitting long sequences into smaller, overlapping windows (e.g., an 5,000bp sequence could be split into 9 1000bp windows, where each successive window overlaps the previous window by 500bp). `ScoreMatch` would only be called when the user's λ threshold is met for a given window (see section 5.3.1). The sequence comparison would start at the locations of the matching windows in both sequences but the identified match may extend beyond the boundaries of these windows in the original sequences. The window size and degree of overlap should be parameters that are specified by the user at run-time. Implementing this functionality has the potential to improve the performance of the program significantly when clustering full length cDNA sequences.

7.6 Automatic Calculation of ζ' and λ

`UIcluster` could be made more user friendly by automatically calculating appropriate values for ζ' and λ for a specified similarity criteria. This would require obtaining the total amount of memory available and choosing ζ' accordingly. It should be noted that the value of ζ' that most closely approximates the performance obtained by the value calculated by equation 5.3 will differ depending on the size and novelty of the data set being clustered and the similarity criteria used. Additionally, the user may wish to obtain better performance by choosing a higher λ threshold than that calculated by equation 5.4, knowing that some sequence similarities may be missed. Therefore, one approach to automatically calculating these values would be to give the user several ζ' and λ combinations to choose from depending what his or her goals are and the type of data set being clustered.

7.7 ExtendMatch Improvements

The approach currently used by the `ExtendMatch` function (see section 5.1.2.2) does not determine the end of a matching region with enough accuracy to be useful for generating the virtual primaries. This is why the use of `bl2seq` was necessary for the virtual primary generation feature (where determining overlaps accurately is important). A member of our laboratory is currently working on an improved version of `ExtendMatch` that will be incorporated in the next release of `UIcluster`.

APPENDIX
UICLUSTER 3.0 SOURCE CODE

A.1 Header Files

A.1.1 uicluster.h

```
/***************************************************************************
                              uicluster.h
                             -----------------
    begin                : Sun Dec 12 1999
    author               : Kevin Pedretti
    email                : pedretti@eng.uiowa.edu
 ***************************************************************************/

#define NEW(TYPE) (TYPE *)emalloc(sizeof(TYPE))

enum {
  MAXFNAME = 100,/* max filename length in chars */
  MAXPRE = 100,/* max num of previously clustered files */
  MAXSNAME = 40,/* maximum sequence name */
  MAXSLEN = 2000000,/* maximum sequence length */
  FORWARD = 1,/* indicates seq in forward direction */
  REVCOMP = 2,/* indicates seq in reverse compliment dir */
  NBASESONLINE = 70,/* num of bases to print per line */
  MAXLINE = 200000,/* maximum length of any line */
  A = 0,
  C = 1,
  G = 2,
  T = 3,
  SUCCESS = 1,
  FAILURE = 0,
  TRUE = 1,
  FALSE = 0,
  FAILEDHASH = -1,/* indicates that no hash could be generated */
  NOHIT = 0,/* indicates no similar cluster was found */
  NO_MORE_SEQS = -1,/* indicates that there are no more seqs in
   input file */
  GOOD_SEQ = 0,/* indicates that input sequence is not a
   reject */
  REJECT_SEQ = 1,/* indicates that input sequence is a reject */
  INTERNAL = 0,
  BOTHEXT = 1,
  TAILEXT = 2,
  FRONTEXT = 3,
  PROBLEM = 4,
};

typedef struct Secondary Secondary;
struct Secondary {
  char *name; /* name of secondary sequence */
  char *seq; /* sequence string */
  double score; /* score of secondary (100 means perfect
   match) */
  int iP; /* match start index in primary */
```

```
  int iS; /* match start index in secondary */
  int matchLen; /* match length in number of bases */
  int dir; /* direction of match.  FORWARD or REVCOMP */
  int tovp; /* how this seq aligned to the vp */
  Secondary *next; /* Secondary in list */
};

typedef struct Primary Primary;
struct Primary {
  int clusID; /* cluster ID of this primary */
  char *name; /* name of primary sequence */
  char *seq; /* sequence string */
  int *hashes; /* hash array for sequence string */
  int *indexes; /* index array for hash positions in seq */
  int nHashes; /* number of hashes */
  int nTouched; /* used for seaching -- num times this
   primary has been examined */
  int nSecondaries; /* number of secondaries for this primary */
  Secondary *headS; /* pointer to the head secondary */
  Primary *nextCP; /* used for searching -- next primary in
   candidate list */
  Primary *next; /* Primary in list */

  /* virtual primary related variables */
  char *vp; /* virtual primary */
  int vpLen; /* cur len of the vp */
  int maxvpLen; /* max length of the vp */
  int nBothExt; /* num secondaries extending vp on front and
   tail */
  int nFrontExt; /* num secondaries extending vp on front */
  int nTailExt; /* num secondaries extending vp on tail */
  int nInternal; /* num secondaries totally contained in vp */
  int nProblems; /* num secondaries not hitting well to vp */
  int tovp; /* how this seq aligned to the vp */
};
```

A.1.2 cluster.h

```
/*************************************************************************
                              cluster.h
                             ---------------
    begin                : Sun Dec 12 1999
    author               : Kevin Pedretti, Tom Casavant
    email                : pedretti@eng.uiowa.edu
 *************************************************************************/

typedef struct GHTEntry GHTEntry;
typedef GHTEntry *GHTEntry_p;
struct GHTEntry {
  Primary *primary; /* pointer to the primary for this entry */
  GHTEntry *next; /* GHTEntry in list */
};

typedef struct Hit_str Hit;
typedef Hit *Hit_p;
struct Hit_str {
  int clusID; /* the clusID hit */
  int score; /* the score of the hit */
  int dir; /* direction of the hit */
};

void cluster(int, int, FILE *, FILE *, Options, Primary **, Primary *,
     int *, int *, int *, int *, int *, int *, int *, int *);

void writeClusters(FILE *, Primary *, int, int *, char *[], int[], int,
```

```
   int);

void addPrimary(int, FASTASeq *, Primary **, Primary **,
int[], int[], int, GHTEntry_p[], int);

void addSecondary(FASTASeq *, Primary *,
  int, int, int, int, int,
  int, int, GHTEntry_p[], int[], int[], int,
  Options, int *, int *, int *);

void addtoGHT(GHTEntry_p *, Primary *, int);

void remfromGHT(GHTEntry_p *, Primary *, int);

int hashSeq(char *, int, int, int, int[], int[]);

int compareSeqs(int, int,
Primary *, char *,
int *, int *, int,
int, int, int,
int *, int *, int, int, int *, int *, int *);

void sortHitList(Hit_p hits, int nHits);
```

A.1.3 compare.h

```
/***************************************************************************
                              compare.h
                            --------------
    begin                : Mon Dec 13 1999
    author               : Tom Casavant - modified by Kevin Pedretti
    email                : pedretti@eng.uiowa.edu
 ***************************************************************************/

int ScoreMatch(char *strPattern, int strPatternIndex,
       char *strSubject, int strSubjectIndex,
       int iLengthToMatch,
       int iWrongLimit, int iGapLimit, int iWrongOrGapLimit,
       int iWrongPenalty, int iGapPenalty,
       int iRecursiveFlag,
       int *nWrong_p, int *nMissing_p, int *nInserted_p);

void extendMatch(char *pSeq, char *cSeq,
 int topScore, int lengthToMatch, int hashSize,
 int radix, int skip, char maskedChar,
 int wrongLimit, int gapLimit, int wrongOrGapLimit,
 int topPIndex, int topCIndex,
 int wrongPenalty, int gapPenalty,
 int *totalErrors, int *bestLength);
```

A.1.4 fasta.h

```
/***************************************************************************
                              fasta.h
                            -------------
    begin                : Sun Dec 12 1999
    author               : Kevin Pedretti
    email                : pedretti@eng.uiowa.edu
 ***************************************************************************/

typedef struct FASTASeq FASTASeq;
struct FASTASeq {
  char name[MAXSNAME]; /* name of sequence */
  char *seq; /* sequence string */
  FASTASeq *next; /* FASTASeq in list */
```

```
};

void readSeqs(FILE *, FILE *, int, FASTASeq **, int *, int *);
int readSeq(FILE *, FASTASeq *, FILE *, int, int);
void printSeq(FILE *, char *, int, int);
```

A.1.5 incremental.h

```
/*************************************************************************
                              incremental.h
                             ------------------
    begin                : Wed Jan 12 2000
    author               : Kevin Pedretti
    email                : pedretti@eng.uiowa.edu
 *************************************************************************/

int readClusFiles(char *c, Primary **, Primary **, char *[], int[]);
int parseClusFile(FILE *, Primary **, Primary **, int *, int *);
```

A.1.6 memory.h

```
/*************************************************************************
                              memory.h
                             --------------
    begin                : Tue Dec 14 1999
    author               : Kevin Pedretti
    email                : pedretti@eng.uiowa.edu
 *************************************************************************/

enum {
  GHTBLOCK = 1000000,/* block size 'GHTEntry' preallocations */
  PRIMEBLOCK = 300000,/* block size for 'Primary' preallocations */
  SECNDBLOCK = 1000000,/* block size for 'Secondary' preallocations */
  FASTABLOCK = 1000000,/* block size for 'FASTASeq' preallocations */
  SEQBLOCK = 1000000,/* block size for prallocations for sequence
   data */
};

GHTEntry *getGHTEntry();
Primary *getPrimary();
Secondary *getSecondary();
FASTASeq *getFASTASeq();
```

A.1.7 options.h

```
/*************************************************************************
                              options.h
                             ---------------
    begin                : Sun Dec 12 1999
    author               : Kevin Pedretti
    email                : pedretti@eng.uiowa.edu
 *************************************************************************/

/* Structure containing all user-defined parameters. */
typedef struct Options Options;
struct Options {
  char inFile[MAXFNAME];
  char preCFile[MAXFNAME];
  int rejCrit;
  int hashSize;
  int startSkip;
  int endSkip;
  int lenToMatch;
  int errLimit;
  char maskChar;
```

```
  int repick;
  int tryRevC;
  int hitThresh;
  int wrongPen;
  int gapPen;
  int keepGoing;
  int vPrimary;
};

/* Called from main to get user-defined parameters from the command-line */
int getopts(Options *, int, char **);

/* Prints out the user-define parameters parsed from the comman-line */
void printopts(FILE *, Options);

/* print command-line arguments and usage instructions */
void printUsage();
```

A.1.8 qsort.h

```
/*************************************************************************
                              qsort.h
                             -------------
    begin                : Mon Dec 13 1999
    author               : Kevin Pedretti
    email                : pedretti@eng.uiowa.edu
 *************************************************************************/

enum {
  M = 7,/* threshold list size for abandoning qsort */
  NSTACK = 50,/* stack size, may have to increase */
  NR_END = 1,/* sentinel */
};

#define SWAP(a,b) temp=(a);(a)=(b);(b)=temp;
#define FREE_ARG char*

void nrerror(char error_text[]);
int *ivector(long nl, long nh);
void free_ivector(int *v, long nl, long nh);

/* Do a quick sort on data[] while maintaining consistancy with index[].
   NOTE: This sort routine sorts data[1..n] NOT data[0..n-1].
         The caller should account for this.
*/
void qsortWIndx(unsigned long n, int data[], int index[]);
```

A.1.9 utils.h

```
/*************************************************************************
                              utils.h
                             -------------
    begin                : Sun Dec 12 1999
    author               : Kevin Pedretti
    email                : pedretti@eng.uiowa.edu
 *************************************************************************/

#define min2(X, Y)  ((X) < (Y) ? (X) : (Y))
#define max2(X, Y)  ((X) > (Y) ? (X) : (Y))

void setprogname(char *str);
char *progname(void);
void setversion(char *str);
char *getversion(void);
extern void eprintf(char *, ...);
```

```
extern void wprintf(char *, ...);
extern char *estrdup(char *);
extern void *emalloc(size_t);
extern void *ecalloc(size_t, size_t);
extern void *erealloc(void *, size_t);
extern char *progname(void);
extern void setprogname(char *);
int nmallocs();
char *chomp(char *);
int countBases(char *);
int ipower(int, int);
void revComp(char *in, int len);
char *printTime(time_t, time_t);
```

A.1.10 bl2seq.h

```
/***************************************************************************
                          bl2seq.h  -  description
                             -------------------
    begin                : Sun Mar 18 2001
    copyright            : (C) 2001 by Kevin Pedretti
    email                : pedretti@eng.uiowa.edu
 ***************************************************************************/

/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/
enum {
  BL2SEQ_MAXLINE = 1024,
  STATE_INIT = 0,
  STATE_STARTS = 1,
  STATE_LENS = 2,
  STATE_STRANDS = 3,
  PLUS = 0,
  MINUS = 1,
  PP = 0,
  PM = 1,
  MP = 2,
  MM = 3,
};

struct bl2seq_hit_str {
  int sb1; /* start base in sequence 1 */
  int sb2;
  int eb1; /* end base in sequence 1 */
  int eb2;
  int dir; /* direction of hit */
};

typedef struct bl2seq_hit_str bl2seq_hit;

void bl2seq(char *seq1, int seq1Len, char *seq2, int seq2Len,
    bl2seq_hit * hits, int *nHits);

int call_bl2seq(char *seq1, char *seq2, char *out);
```

A.2 Source Files

A.2.1 main.c

```
/***************************************************************************
```

```
                        main.c  -  UIcluster clustering program
                        ---------------------------------------
    begin               : Sun Dec 12 09:43:22 CST 1999
    author              : Kevin Pedretti
    email               : pedretti@eng.uiowa.edu
 ***************************************************************************/

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "mpi.h"
#include "uicluster.h"
#include "utils.h"
#include "options.h"
#include "fasta.h"
#include "cluster.h"
#include "incremental.h"

int main(int argc, char *argv[])
{
  int myRank; /* this procs rank in MPI_COMM_WORLD */
  int nProcs; /* num procs in MPI_COMM_WORLD */
  time_t startTime; /* start time of program execution */
  time_t stopTime1; /* stop time before writing output file */
  time_t stopTime2; /* stop top after writing output file */
  Options opts; /* Contains user configurable options */
  FASTASeq *inseqs = NULL; /* linked list of input sequences */
  Primary *primaries = NULL; /* linked list of primaries */
  Primary *tail = NULL; /* pointer to the last element in the primary
   linked list */
  int nCF = 0; /* number of previously clustered files to
   read in */
  char *inFiles[MAXPRE]; /* the names of the input previously
   clustered files */
  int div[MAXPRE]; /* indexes to last primary of each
   pre-clustered file */
  char outFN[MAXFNAME],/* output file name */
   rejFN[MAXFNAME],/* reject file name */
   logFN[MAXFNAME],/* log file name */
   orphanFN[MAXFNAME]; /* orphan file name */
  FILE *fd_fasta,/* input FASTA file */
  *fd_out,/* output file containing clusters */
  *fd_rej,/* file containing rejected sequences */
  *fd_log,/* log file */
  *fd_orphan; /* file containing orphans */
  int nSeqs = 0,/* num of input sequences */
   nRej = 0,/* num of input seqs rejected */
   nPrime = 0,/* num primaries after clustering */
   nSecnd = 0,/* num secondaries after clustering */
   nOrph = 0,/* num of orphans */
   nOrphE = 0,/* num orphan events */
   nReadptE = 0,/* num readopt events */
   nRepick = 0,/* num repick events */
   nMatchRev = 0; /* num seq matches in the REVCOMP direction */
  int stat;
  int i;

  /* initialize MPI */
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
```

```
MPI_Comm_size(MPI_COMM_WORLD, &nProcs);

/* record the start time of clustering */
startTime = time(NULL);
setprogname("UIcluster");
setversion("3.0.5");

/* only the rank 0 proc outputs to screen */
if (myRank == 0) {
  printf("\n%s %s\n", progname(), getversion());
  for (i = 0; i <= (strlen(progname()) + strlen(getversion())); i++) {
    printf("-");
  }
  printf("\n\n");
}

/* parse the command line */
stat = getopts(&opts, argc, argv);
if (stat == FAILURE) {
  eprintf(" COULDN'T PARSE COMMAND LINE.");
}
if (myRank == 0) {
  printf("Running with %d processes.\n", nProcs);
  printopts(stdout, opts);
}

/* open in input file */
fd_fasta = fopen(opts.inFile, "r");
if (fd_fasta == NULL) {
  eprintf("can't open %s:", opts.inFile);
}

/* build the name of this procs output file */
if (nProcs == 1) {
  sprintf(outFN, "%s.clus", opts.inFile);
} else {
  sprintf(outFN, "%s-%d.clus", opts.inFile, myRank);
}

/* open the output file */
fd_out = fopen(outFN, "wc");
if (fd_out == NULL) {
  eprintf("can't open %s", outFN);
}

/* only process 0 outputs rejects */
if (myRank == 0) {
  sprintf(rejFN, "%s.rej", opts.inFile);
  /* open the reject file */
  fd_rej = fopen(rejFN, "wc");
  if (fd_rej == NULL)
    eprintf("can't open %s", rejFN);
}

/* read in previously clustered file */
if (strcmp(opts.preCFile, "none") != 0) {
  if (myRank == 0) {
    printf("\nReading previously clustered files...\n");
  }
  nCF = readClusFiles(opts.preCFile, &primaries, &tail, inFiles, div);
}

/* perform clustering */
if (myRank == 0) {
  printf("\nBegin Clustering...\n");
```

```
    printf
("\n  *status given as .num_seqs_clustered:num_new_clusters.\n\n");
  }
  if (myRank == 0) {
    /* only the master output rejects to the reject file */
    cluster(myRank, nProcs, fd_fasta, fd_rej, opts, &primaries, tail,
    &nPrime, &nSecnd, &nRepick, &nOrphE, &nReadptE, &nMatchRev,
    &nSeqs, &nRej);
  } else {
    /* set fd_rej to null so nothing is output to it */
    cluster(myRank, nProcs, fd_fasta, NULL, opts, &primaries, tail,
    &nPrime, &nSecnd, &nRepick, &nOrphE, &nReadptE, &nMatchRev,
    &nSeqs, &nRej);
  }

  /* record the time up until just after clustering */
  stopTime1 = time(NULL);

  /* finish up - write primaries to file and display summary stats */
  if (myRank == 0) {
    printf("\n\nWritting output...\n");
  }
  writeClusters(fd_out, primaries, NBASESONLINE, &nOrph, inFiles, div, nCF,
opts.vPrimary);

  /* record the stop time of the clustering */
  stopTime2 = time(NULL);

  if (myRank == 0) {
    printf("\n");
    printf("Total # Input:      %7d\n", nSeqs);
    printf("Num Rejects:        %7d\n", nRej);
    printf("# Seqs Clustered:   %7d\n", nSeqs - nRej);
    printf("Num Clusters:       %7d\n", nPrime);
    printf("Num Secondaries:    %7d\n", nSecnd);
    if (opts.repick) {
      printf("Num Repick Events:  %7d\n", nRepick);
      printf("Num Orphan Events:  %7d\n", nOrphE);
      printf("Num ReAdopt Events: %7d\n", nReadptE);
      if (nOrph > 0)
printf("Total # Orphans:    %7d\n", nOrph);
    }
    if (opts.tryRevC)
      printf("Num Matched Rev:    %7d\n", nMatchRev);
    printf("Total # of mallocs: %7d\n", nmallocs());
    printf("Tot Elapsed time  :    %s\n", printTime(startTime, stopTime2));
    /* printf(        "Output time       :    %s\n", printTime(stopTime1,
       stopTime2)); */
  }

  /* finish up */
  MPI_Finalize();
  return EXIT_SUCCESS;
}
```

A.2.2 cluster.c

```
/*****************************************************************************
                 cluster.c  -  routines to cluster sequences
                 -------------------------------------------
    begin                : Sun Dec 12 1999
    copyright            : Kevin Pedretti, Tom Casavant
    email                : pedretti@eng.uiowa.edu
 *****************************************************************************/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mpi.h"
#include "uicluster.h"
#include "utils.h"
#include "options.h"
#include "fasta.h"
#include "cluster.h"
#include "compare.h"
#include "qsort.h"
#include "memory.h"
#include "bl2seq.h"

void cluster(int myRank, int nProcs, FILE * fd_fasta, FILE * fd_rej,
     Options opts, Primary ** head, Primary * tail, int *nPrime,
     int *nSecnd, int *nRepick, int *nOrphE, int *nReadptE,
     int *nMatchRev, int *nSeqs, int *nRejects)
{
  Primary *clhead; /* candidate list head */
  Primary *tmpP; /* temporary pointer to a Primary */
  FASTASeq *curC; /* current candidate */
  Primary *curP; /* current primary */
  GHTEntry_p *ght; /* global hash table */
  GHTEntry *tmp; /* temporary variable */
  int *cHashes; /* hashes for cur */
  int *cIndexes; /* indexes for cur */
  int nhashes; /* num hashes for cur */
  char revC[MAXSLEN]; /* holds the reverse */
  int *crHashes; /* hashes for revcomp of cur */
  int *crIndexes; /* hashes for revcomp of cur */
  int nrhashes; /* num hashes for revcomp of cur */
  int iter = 0;
  int hit; /* indicates if cur matched any primaries */
  int status; /* stores return codes of function calls */
  int i;
  int stopSearch; /* boolean indicating if the primary search
   should be stopped */
  struct {
    int score;
    int rank;
  } myBestHit, bestHit;
  struct Hit_str allMyHits[10000];
  struct Hit_str allHits[10000];
  int clusID = 0;
  MPI_Datatype MPI_HIT;
  int *rcounts;
  int *displs;
  int matchLen,/* initial match length */
   extLen,/* extended match length */
   nerrors; /* num errors in match */
  int topP, topC, nwrong, nmiss, ninsert;
  int bestExtLen, bestTopP, bestTopC, bestHitDir, bestNErrors;
  Primary *bestP;
  FILE *hits; /* file containing list of candidates and the
   primaries they match */
  char hitsName[MAXFNAME];
  int hitCount; /* the number of clusters an incoming seq
   hits */

  cHashes = (int *) emalloc(MAXSLEN * sizeof(int));
  cIndexes = (int *) emalloc(MAXSLEN * sizeof(int));
  crHashes = (int *) emalloc(MAXSLEN * sizeof(int));
  crIndexes = (int *) emalloc(MAXSLEN * sizeof(int));
  nhashes = nrhashes = 0;
```

```
  /* Define MPI_HIT datatype */
  MPI_Type_contiguous(3, MPI_INT, &MPI_HIT);
  MPI_Type_commit(&MPI_HIT);

  /* Allocate rcounts and displs */
  if (opts.keepGoing && (myRank == 0)) {
    rcounts = (int *) malloc(nProcs * sizeof(int));
    displs = (int *) malloc(nProcs * sizeof(int));
  }

  /* initialize memory for the global hash table */
  ght =
      (GHTEntry_p *) ecalloc(ipower(4, opts.hashSize), sizeof(GHTEntry_p));

  /* hash all of the already existing primaries */
  if (*head != NULL) {
    printf("  creating hashes for pre-existing primaries.\n");
    tmpP = *head;
    while (tmpP != NULL) {
      nhashes =
  hashSeq(tmpP->seq, opts.hashSize, opts.startSkip, opts.endSkip,
  cHashes, cIndexes);
      qsortWIndx((unsigned long) nhashes, cHashes - 1, cIndexes - 1);
      tmpP->nHashes = nhashes;
      tmpP->hashes = (int *) emalloc(nhashes * sizeof(int));
      tmpP->indexes = (int *) emalloc(nhashes * sizeof(int));
      for (i = 0; i < nhashes; i++) {
tmpP->hashes[i] = cHashes[i];
tmpP->indexes[i] = cIndexes[i];
addtoGHT(ght, tmpP, cHashes[i]);
      }
      tmpP = tmpP->next;
    }
  }

  /* open a file for the list of candidates and the primaries they match */
  if (opts.keepGoing && (myRank == 0)) {
    strcpy(hitsName, opts.inFile);
    strcat(hitsName, ".hits");
    hits = fopen(hitsName, "w");
    if (hits == NULL) {
      eprintf("can't open %s: ", hitsName);
    }
  }

  /* allocate memory for curC */
  curC = getFASTASeq();
  curC->seq = (char *) emalloc(MAXSLEN * sizeof(char));

  /* cluster each input seq one by one... */
  status = readSeq(fd_fasta, curC, fd_rej, opts.rejCrit, *nRejects);
  while (status != NO_MORE_SEQS) {
    ++(*nSeqs);
    if (status == REJECT_SEQ) {
      ++(*nRejects);
    } else {
      if (myRank == 0) {
if (iter % 100 == 0) {
  printf("%d:%d.", iter, *nPrime);
  fflush(stdout);
}
++iter;
      }
```

```
      if (opts.keepGoing && (myRank == 0)) {
/* print the current candidate name to the hits file */
fprintf(hits, "%s ", curC->name);
      }

      nhashes =
  hashSeq(curC->seq, opts.hashSize, opts.startSkip, opts.endSkip,
  cHashes, cIndexes);
      qsortWIndx((unsigned long) nhashes, cHashes - 1, cIndexes - 1);

      clhead = NULL;
      hit = NOHIT;
      hitCount = 0;
      bestExtLen = 0;

      /* search the GHT with the forward hashes */
      i = 0;
      stopSearch = FALSE;
      while ((i < nhashes) && (stopSearch != TRUE)) {
if (cHashes[i] != FAILEDHASH) {
  /* get the first link at entry i of the GHT */
  tmp = ght[cHashes[i]];
  while (tmp != NULL) {
    curP = tmp->primary;
    /* only check primary when it exactly hits the threshold */
    /* prevents checking it more than once if keepGoing flag is on */
    if (curP->nTouched == opts.hitThresh) {
      matchLen =
  compareSeqs(opts.lenToMatch,
      opts.lenToMatch - opts.errLimit, curP,
      curC->seq, cHashes, cIndexes, nhashes,
      opts.errLimit, opts.errLimit, opts.errLimit,
      &topP, &topC, opts.wrongPen, opts.gapPen,
      &nwrong, &nmiss, &ninsert);
      if (matchLen >= (opts.lenToMatch - opts.errLimit)) {

extendMatch(curP->seq, curC->seq,
    matchLen, opts.lenToMatch, opts.hashSize, 4,
    opts.startSkip, opts.maskChar, opts.errLimit,
    opts.errLimit, opts.errLimit, topP, topC,
    opts.wrongPen, opts.gapPen, &nerrors, &extLen);

hit = FORWARD;

/* add the hit to the hit list */
allMyHits[hitCount].clusID = curP->clusID;
allMyHits[hitCount].score = extLen;
allMyHits[hitCount].dir = FORWARD;
++hitCount;

/* see if this is the best hit so far */
if (extLen > bestExtLen) {
  bestExtLen = extLen;
  bestTopP = topP;
  bestTopC = topC;
  bestHitDir = FORWARD;
  bestNErrors = nerrors;
  bestP = curP;
}

/* stop the search if the keepGoing flag is off */
if (opts.keepGoing == 0) {
  stopSearch = TRUE;
}
      }
```

```
    }

    if (curP->nTouched == 0) {
      /* insert primary at head of candidate list */
      curP->nextCP = clhead;
      clhead = curP;
    }

    ++(curP->nTouched);
    tmp = tmp->next;
  } /* end while (tmp != NULL) */
} /* end if (cHashes[i] != FAILEDHASH) */
++i; /* move on to next hash */
      } /* end while ((i < nhashes) && (stopSearch !=
   TRUE)) */

      /* check in the reverse direction if the tryRevC option is on and
         either the extended search option is on or no hit has been found yet */
      if (((opts.keepGoing == 1) && (opts.tryRevC == 1))
  || ((hit == NOHIT) && (opts.tryRevC == 1))) {
/* reset candidate list, zeroing touchcounts */
curP = clhead;
while (curP != NULL) {
  curP->nTouched = 0;
  curP = curP->nextCP;
}
clhead = NULL;

/* reverse compliment the input sequence */
strcpy(revC, curC->seq);
revComp(revC, strlen(revC));

/* generate and sort hashes for the reverse complement */
nrhashes =
    hashSeq(revC, opts.hashSize, opts.startSkip, opts.endSkip,
    crHashes, crIndexes);
qsortWIndx((unsigned long) nrhashes, crHashes - 1, crIndexes - 1);

/* search the GHT with the reverse hashes */
i = 0;
stopSearch = FALSE;
while ((i < nrhashes) && (stopSearch != TRUE)) {
  if (crHashes[i] != FAILEDHASH) {
    /* get the first link at entry i of the GHT */
    tmp = ght[crHashes[i]];
    while (tmp != NULL) {
      curP = tmp->primary;
      /* only check primary when it exactly hits the threshold */
      /* prevents checking it more than once if keepGoing flag is on */
      if (curP->nTouched == opts.hitThresh) {
matchLen =
    compareSeqs(opts.lenToMatch,
opts.lenToMatch - opts.errLimit, curP,
revC, crHashes, crIndexes, nrhashes,
opts.errLimit, opts.errLimit,
opts.errLimit, &topP, &topC, opts.wrongPen,
opts.gapPen, &nwrong, &nmiss, &ninsert);
if (matchLen >= (opts.lenToMatch - opts.errLimit)) {

  extendMatch(curP->seq, revC,
      matchLen, opts.lenToMatch, opts.hashSize, 4,
      opts.startSkip, opts.maskChar, opts.errLimit,
      opts.errLimit, opts.errLimit, topP, topC,
      opts.wrongPen, opts.gapPen, &nerrors,
      &extLen);
```

```
  hit = REVCOMP;

  /* add the hit to the hit list */
  allMyHits[hitCount].clusID = curP->clusID;
  allMyHits[hitCount].score = extLen;
  allMyHits[hitCount].dir = REVCOMP;
  ++hitCount;

  /* see if this is the best hit so far */
  if (extLen > bestExtLen) {
    bestExtLen = extLen;
    bestTopP = topP;
    bestTopC = topC;
    bestHitDir = REVCOMP;
    bestNErrors = nerrors;
    bestP = curP;
  }

  /* stop the search if the keepGoing flag is off */
  if (opts.keepGoing == 0) {
    stopSearch = TRUE;
  }
}
      }

      if (curP->nTouched == 0) {
/* insert primary at head of candidate list */
curP->nextCP = clhead;
clhead = curP;
      }

      ++(curP->nTouched);
      tmp = tmp->next;
    } /* end while (tmp != NULL) */
  } /* end if (crHashes[i] != FAILEDHASH) */
  ++i; /* move on to next hash */
} /* end while ((i < nrhashes) && (stopSearch
   != TRUE)) */
      }

      /* figure out what my best hit is */
      if (hitCount > 0) {
/* sort the hit list by score */
sortHitList(allMyHits, hitCount);

/* best hit will be at position 0 */
myBestHit.score = allMyHits[0].score;
      } else {
myBestHit.score = 0;
      }
      myBestHit.rank = myRank;

      /* communicate the best hit to all procs */
      MPI_Allreduce(&myBestHit, &bestHit, 1, MPI_2INT, MPI_MAXLOC,
    MPI_COMM_WORLD);

      if (bestHit.score == 0) { /* cur is a new primary */
++(*nPrime);

/* check if this proc should get assigned this cluster */
if ((*nPrime % nProcs) == myRank) {
  addPrimary(clusID, curC, head, &tail, cHashes, cIndexes, nhashes,
     ght, opts.vPrimary);
  ++clusID;
```

```
}

/* master proc writes to hits file if keepGoing flag is on */
if ((myRank == 0) && (opts.keepGoing == 1)) {
  fprintf(hits, "NO HITS\n");
}
      } else { /* cur gets added someplace */
++(*nSecnd);

/* gather a list of all hits to the master proc if the keepGoing flag
   is on */
if (opts.keepGoing) {
  /* gather list of all hits to root node to put in the hits file */
  MPI_Gather(&hitCount, 1, MPI_INT, rcounts, 1, MPI_INT, 0,
     MPI_COMM_WORLD);
  if (myRank == 0) {
    displs[0] = 0;
    for (i = 1; i < nProcs; i++) {
      displs[i] = displs[i - 1] + rcounts[i - 1];
    }
  }
  MPI_Gatherv(allMyHits, hitCount, MPI_HIT, allHits, rcounts,
      displs, MPI_HIT, 0, MPI_COMM_WORLD);


  /* only master proc writes to the file */
  if (myRank == 0) {
    int nAllHits = displs[nProcs - 1] + rcounts[nProcs - 1];
    /* sort the hit list before writing it */
    sortHitList(allHits, nAllHits);

    for (i = 0; i < nAllHits; i++) {
      fprintf(hits, "%d.%d:", allHits[i].clusID, allHits[i].score);
      if (allHits[i].dir == FORWARD) {
fprintf(hits, "f ");
      } else {
fprintf(hits, "r ");
      }
    }
    fprintf(hits, "\n");
  }
}

/* add the input sequence to the cluster it hit the best */
if (bestHit.rank == myRank) {
  addSecondary(curC, bestP, bestTopP, bestTopC, bestExtLen,
       bestNErrors, bestHitDir, opts.repick, opts.vPrimary,
       ght, cHashes, cIndexes, nhashes, opts, nRepick,
       nOrphE, nReadptE);
  // addSecondary(curC, curP, topP, topC, extLen, nerrors, hit,
  // opts.repick, opts.vPrimary, ght, cHashes, cIndexes, nhashes,
  // opts,
  // nRepick, nOrphE, nReadptE);
}
      }

      /* Reset candidate list */
      curP = clhead;
      while (curP != NULL) {
curP->nTouched = 0;
curP = curP->nextCP;
      }


    } /* end if (status != REJECT_SEQ) */
```

```
    /* read the next input sequence */
    status = readSeq(fd_fasta, curC, fd_rej, opts.rejCrit, *nRejects);
  } /* end main clustering loop */
}

void addPrimary(int clusID, FASTASeq * new, Primary ** head,
Primary ** tail, int hashes[], int indexes[], int nhashes,
GHTEntry_p ght[], int vPrimary)
{

  int i;

  Primary *p = getPrimary();
  p->clusID = clusID;
  p->name = (char *) emalloc((strlen(new->name) + 1) * sizeof(char));
  p->seq = (char *) emalloc((strlen(new->seq) + 1) * sizeof(char));
  strcpy(p->name, new->name);
  strcpy(p->seq, new->seq);

  /* only do the following only if virtual primary option is enabled */
  if (vPrimary == 1) {
    /* vp starts out as seq of first sequence added */
    p->vp = (char *) emalloc((strlen(p->seq) * 2) + 1);
    strcpy(p->vp, p->seq);
    p->vpLen = strlen(p->seq);
    p->maxvpLen = strlen(p->seq) * 2;
    p->nFrontExt = 0;
    p->nTailExt = 0;
    p->nBothExt = 0;
    p->nInternal = 0;
    p->nProblems = 0;
    p->tovp = INTERNAL;
  }
  p->hashes = (int *) emalloc(nhashes * sizeof(int));
  p->indexes = (int *) emalloc(nhashes * sizeof(int));

  /* enter the hashes into the GHT */
  for (i = 0; i < nhashes; i++) {
    p->hashes[i] = hashes[i];
    p->indexes[i] = indexes[i];
    addtoGHT(ght, p, hashes[i]);
  }

  /* initialize variables in primary structure */
  p->nHashes = nhashes;
  p->nSecondaries = 0;
  p->nTouched = 0;
  p->headS = NULL;
  p->nextCP = NULL;
  p->next = NULL;

  /* Add primary to the main primary list */
  if (*head != NULL) {
    (*tail)->next = p;
    *tail = p;
  } else {
    *head = *tail = p;
  }
}

void addSecondary(FASTASeq * new, Primary * p,
  int iP, int iS, int matchLen, int errors, int dir,
  int repick, int vPrimary, GHTEntry_p ght[], int hashes[],
  int indexes[], int nhashes, Options opts, int *nRepick,
```

```
  int *nOrphanE, int *nReadptE)
{
  Secondary *tmpS;
  Secondary *s = getSecondary();
  int lenOld, lenNew;
  int i;

  char strTmp[MAXSLEN];
  int nlhashes;
  int *cHashes;
  int *cIndexes;
  int lmatchLen;
  int topP, topC, nwrong, nmiss, ninsert;
  int nerrors;
  int extLen;
  bl2seq_hit bl2Hits[1000];
  int nbl2Hits;

  ++(p->nSecondaries);
  s->next = p->headS;
  p->headS = s;

  /* determine if the secondary should become the primary for the cluster */
  if (repick) {
    lenOld = countBases(p->seq);
    lenNew = countBases(new->seq);
  }

  if (lenNew > lenOld) {
    /* repick */
    ++(*nRepick);

    s->name = p->name;
    s->seq = p->seq;
    s->iP = 0;
    s->iS = 0;
    s->matchLen = 0;
    s->score = 0.0;
    s->dir = FORWARD;

    p->name = (char *) emalloc((strlen(new->name) + 1) * sizeof(char));
    p->seq = (char *) emalloc((strlen(new->seq) + 1) * sizeof(char));
    strcpy(p->name, new->name);
    strcpy(p->seq, new->seq);

    if (vPrimary == 0) {
      /* remove old primary from ght */
      for (i = 0; i < p->nHashes; i++) {
remfromGHT(ght, p, p->hashes[i]);
      }
      free(p->hashes);
      free(p->indexes);

      /* add the new primaries hashes to the ght */
      p->hashes = (int *) emalloc(nhashes * sizeof(int));
      p->indexes = (int *) emalloc(nhashes * sizeof(int));
      for (i = 0; i < nhashes; i++) {
p->hashes[i] = hashes[i];
p->indexes[i] = indexes[i];
addtoGHT(ght, p, hashes[i]);
      }
      p->nHashes = nhashes;

      /* recluster all of the secondaries */
      cHashes = (int *) emalloc(MAXSLEN * sizeof(int));
```

```
      cIndexes = (int *) emalloc(MAXSLEN * sizeof(int));
      tmpS = p->headS;
      for (i = 0; i < p->nSecondaries; i++) {
if (tmpS->dir == FORWARD) {
  strcpy(strTmp, tmpS->seq);
} else {
  strcpy(strTmp, tmpS->seq);
  revComp(strTmp, strlen(strTmp));
}

nlhashes =
    hashSeq(strTmp, opts.hashSize, opts.startSkip, opts.endSkip,
    cHashes, cIndexes);
qsortWIndx((unsigned long) nlhashes, cHashes - 1, cIndexes - 1);
matchLen = topP = topC = 0;
nwrong = nmiss = ninsert = 0;

lmatchLen =
    compareSeqs(opts.lenToMatch, opts.lenToMatch - opts.errLimit,
p, strTmp, cHashes, cIndexes, nlhashes,
opts.errLimit, opts.errLimit, opts.errLimit, &topP,
&topC, opts.wrongPen, opts.gapPen, &nwrong, &nmiss,
&ninsert);


if (lmatchLen >= (opts.lenToMatch - opts.errLimit)) {
  nerrors = extLen = 0;
  extendMatch(p->seq, strTmp,
      lmatchLen, opts.lenToMatch, opts.hashSize, 4,
      opts.startSkip, opts.maskChar, opts.errLimit,
      opts.errLimit, opts.errLimit, topP, topC,
      opts.wrongPen, opts.gapPen, &nerrors, &extLen);


  /* Update the match info */
  if ((tmpS->score == 0) && (i != 0))
    ++(*nReadptE);
  tmpS->iP = topP;
  tmpS->iS = topC;
  tmpS->score =
      (((double) (lmatchLen - nerrors)) / lmatchLen) * 100.0;
  tmpS->matchLen = extLen;
} else {
  ++(*nOrphanE);
  tmpS->iP = 0;
  tmpS->iS = 0;
  tmpS->score = 0;
  tmpS->matchLen = 0;
}

tmpS = tmpS->next;
      }
      free(cHashes);
      free(cIndexes);
    } /* end if (vPrimary == 0) */
  } else {
    /* no repick, add secondary */
    s->name = (char *) emalloc((strlen(new->name) + 1) * sizeof(char));
    s->seq = (char *) emalloc((strlen(new->seq) + 1) * sizeof(char));
    strcpy(s->name, new->name);
    strcpy(s->seq, new->seq);

    s->iP = iP;
    if (dir == FORWARD)
      s->iS = iS;
```

```
    else
      s->iS = strlen(s->seq) - iS;
    s->matchLen = matchLen;
    if (dir == REVCOMP)
      s->matchLen *= -1;

    s->score = (((double) (matchLen - errors)) / matchLen) * 100.0;
    s->dir = dir;
  }

  /* Try to extend the vp */
  if (vPrimary == 1) {
    int seqLen = strlen(s->seq);
    int lag = 4; /* acceptable error for overlaps */
    int maxMultiHit = 40; /* maximum length in bases of non-best hits
   before * sequence is considered a
   "problem" */
    int minBestHitLen = 50; /* minimum length of a best hit in order to
   extend vp */
    int done = FALSE;
    int ext = FALSE;

    /* call bl2seq.  returns sorted hit list and number of hits */
    bl2seq(p->vp, p->vpLen, s->seq, seqLen, bl2Hits, &nbl2Hits);

    if (nbl2Hits > 0) {
      if (nbl2Hits > 1) {
/* There was more than one hit */
int lenOf2Hit = bl2Hits[1].eb1 - bl2Hits[1].sb1;
if (lenOf2Hit > maxMultiHit) {
  ++(p->nProblems);
  done = TRUE;
}
      }

      if (done == FALSE) {
/* see if the vp can be extended */
/* best hit will be in bl2Hits[0] */
int sb1 = bl2Hits[0].sb1; /* start base in vp */
int sb2 = bl2Hits[0].sb2; /* start base in new seq */
int eb1 = bl2Hits[0].eb1; /* end base in vp */
int eb2 = bl2Hits[0].eb2; /* end base in new seq */
int dir = bl2Hits[0].dir; /* direction of hit -- see bl2seq.c */
int len1 = p->vpLen; /* current length of the vp */
int len2 = seqLen; /* length of the new sequence */

int lenOfHit = eb1 - sb1; /* len of the best hit */
int ff1 = sb1; /* front fringe */
int ef1 = len1 - eb1 - 1; /* end fringe */
int ff2 = sb2;
int ef2 = len2 - eb2 - 1;

if ((lenOfHit > minBestHitLen) && (dir == PP)) {
  /* Check for internal hit */
  if ((ff2 < lag) && (ef2 < lag)) {
    /* hit identified as internal */
    ++(p->nInternal);
  }
    else if ((ff1 < lag) && (ef1 < lag) && (ff2 > lag)
     && (ef2 > lag)) {
    /* hit identified to extend vp on both front and end */
    ++(p->nBothExt);
    ext = TRUE;
    if (len2 > p->maxvpLen) {
      free(p->vp);
```

```
      p->maxvpLen = len2 * 2;
      p->vp = (char *) emalloc(p->maxvpLen);
    }
    p->vpLen = len2;

    /* new sequence becomes new vp */
    strcpy(p->vp, s->seq);
  } else if ((ef1 < lag) && (ff2 < lag) && (ef2 > lag)) {
    /* hit identified to extend tail of vp */
    int sb; /* base to start copying at */
    int lenApp; /* length of appended region */

    ++(p->nTailExt);
    ext = TRUE;
    sb = eb2 + (len1 - eb1);
    lenApp = strlen(s->seq + sb);

    /* allocate more space for vp if necessary */
    if (len1 + lenApp) {
      p->maxvpLen = (len1 + lenApp) * 2;
      p->vp = erealloc(p->vp, p->maxvpLen);
    }
    strcpy(p->vp + len1, s->seq + (eb2 + (len1 - eb1)));
  } else if ((ff1 < lag) && (ef2 < lag) && (ff2 > lag)) {
    /* hit identified to extend front of vp */
    int sb; /* base to start copying at */
    int lenApp; /* length of appended region */

    ++(p->nFrontExt);
    ext = TRUE;
    sb = sb2 - sb1;
    lenApp = sb2 - sb1;
    /* allocate more space for vp if necessary */
    if (len1 + lenApp) {
      p->maxvpLen = (len1 + lenApp) * 2;
      p->vp = erealloc(p->vp, p->maxvpLen);
    }
    /* shift current vp to right */
    memmove(p->vp + lenApp, p->vp, len1 + 1);
    memcpy(p->vp, s->seq, lenApp);
  } else {
    /* problem hit */
    ++(p->nProblems);
  }

  /* if extention was made, remove old hashes from GHT *  and add new
     hashes to it */
  if (ext == TRUE) {
    int *newHashes = (int *) emalloc(MAXSLEN * sizeof(int));
    int *newIndexes = (int *) emalloc(MAXSLEN * sizeof(int));

    /* remove old primary from ght */
    for (i = 0; i < p->nHashes; i++) {
      remfromGHT(ght, p, p->hashes[i]);
    }
    free(p->hashes);
    free(p->indexes);

    /* add the new vps hashes to the ght */
    p->nHashes =
hashSeq(p->vp, opts.hashSize, opts.startSkip, opts.endSkip,
newHashes, newIndexes);
    qsortWIndx((unsigned long) p->nHashes, newHashes - 1,
       newIndexes - 1);
    p->hashes = (int *) emalloc(p->nHashes * sizeof(int));
```

```
    p->indexes = (int *) emalloc(p->nHashes * sizeof(int));
    for (i = 0; i < (p->nHashes); i++) {
      p->hashes[i] = newHashes[i];
      p->indexes[i] = newIndexes[i];
      addtoGHT(ght, p, newHashes[i]);
    }
    free(newHashes);
    free(newIndexes);
  }
} else {
  printf("best hit dir = %d\n", dir);
  printf("len of hit = %d\n", lenOfHit);
}
      }
    } else {
      /* There were no bl2seq hits */
      printf("THIS SHOULD NEVER HAPPEN\n");
    }
  }
}

void addtoGHT(GHTEntry_p * ght, Primary * primary, int hash)
{

  GHTEntry *tmp;

  if (hash != FAILEDHASH) {
    if (ght[hash] != NULL) {
      if (ght[hash]->primary != primary) {
/* The primary isn't in the list. Always insert at head. */
tmp = getGHTEntry();
tmp->primary = primary;
tmp->next = ght[hash];
ght[hash] = tmp;
      }
    } else {
      ght[hash] = getGHTEntry();
      ght[hash]->primary = primary;
      ght[hash]->next = NULL;
    }
  }
}

void remfromGHT(GHTEntry_p * ght, Primary * primary, int hash)
{
  GHTEntry *tmp, *last = NULL;

  if (hash != FAILEDHASH) {
    tmp = ght[hash];
    while (tmp != NULL) {
      if (tmp->primary == primary) {
if (last == NULL) {
  ght[hash] = tmp->next;
} else {
  last->next = tmp->next;
}
return;
      }
      last = tmp;
      tmp = tmp->next;
    }
  }
}
```

```
int hashSeq(char *seq, int hs, int ss, int es, int hsh[], int idx[])
{
  int len; /* holds length of seq */
  int i, j; /* standard counters */
  int h; /* the current hash is built in h */
  int nhashes = 0; /* num hashes generated counter */

  len = strlen(seq);
  for (i = ss; i < (len - es); i++) {
    if (i < (len - (hs - 1))) {
      /* Calculate the hash for this base */
      h = j = 0; /* initialize hash and loop counter */
      while ((j < hs) && (h != FAILEDHASH)) {
if (seq[i + j] == 'A') {
  h += (A * ipower(4, j));
} else if (seq[i + j] == 'C') {
  h += (C * ipower(4, j));
} else if (seq[i + j] == 'G') {
  h += (G * ipower(4, j));
} else if (seq[i + j] == 'T') {
  h += (T * ipower(4, j));
} else {
  h = FAILEDHASH;
}
++j;
      }
      hsh[i - ss] = h;
      idx[i - ss] = i;
      ++nhashes;
    }
  }
  return (nhashes);
}

int compareSeqs(int iLengthToMatch, int iGoodScore,
Primary * curP_p, char *strCSeq,
int *iaCHashes, int *iaCIndexes, int iCNumHashes,
int wrongLimit, int gapLimit, int wrongOrGapLimit,
int *topPIndex, int *topCIndex,
int iWrongPenalty, int iGapPenalty,
int *nFoundWrong, int *nFoundMissing, int *nFoundInserted)
{

  int topScore, score;
  int iP, iC, iCStart;
  int iPNumHashes;
  int iPSeqLen, iCSeqLen;
  int i, numMasked;

  iP = iC = 0;
  while (curP_p->hashes[iP] == FAILEDHASH)
    iP++;
  while (iaCHashes[iC] == FAILEDHASH)
    iC++;
  iPNumHashes = curP_p->nHashes;
  iPSeqLen = strlen(curP_p->seq);
  iCSeqLen = strlen(strCSeq);

  // printf("numHashesP = %d, numHashesC = %d\n", iPNumHashes, iCNumHashes);

  topScore = score = 0;
  while ((iP < iPNumHashes) && (iC < iCNumHashes)) {

    if (curP_p->hashes[iP] == iaCHashes[iC]) {
      iCStart = iC;
```

```
      while ((curP_p->hashes[iP] == iaCHashes[iC]) && (iP <= iPNumHashes)) {
while ((curP_p->hashes[iP] == iaCHashes[iC])
       && (iC <= iCNumHashes)) {
  if ((curP_p->indexes[iP] <= (iPSeqLen - iLengthToMatch))
      && (iaCIndexes[iC] <= (iCSeqLen - iLengthToMatch))) {

    /* Only call score match if the match has the potential to be
       more left than the previous best match */
    if (
((curP_p->indexes[iP] + iaCIndexes[iC]) <
 (*topPIndex + *topCIndex)) || (topScore == 0)) {

      /* Only call score match if there are not more than wrongLimit
         masked characters in a iLengthToMatch Region */
      numMasked = 0;
      for (i = iaCIndexes[iC];
   i < (iaCIndexes[iC] + iLengthToMatch); i++) {
if (strCSeq[i] == 'X')
  ++numMasked;
      }

      if (numMasked <= wrongLimit) {

score = ScoreMatch(curP_p->seq, curP_p->indexes[iP],
   strCSeq, iaCIndexes[iC], iLengthToMatch,
   wrongLimit, gapLimit, wrongOrGapLimit,
   iWrongPenalty, iGapPenalty, 0,/* recursiveFlag
 */
   nFoundWrong, nFoundMissing,
   nFoundInserted);


if (score >= iGoodScore) {
  topScore = score;
  *topPIndex = curP_p->indexes[iP];
  *topCIndex = iaCIndexes[iC];
}
      }
    }
  }
  iC++;
}
iC = iCStart;
iP++;
      }
    } else { /* the hashes dont match */
      if (curP_p->hashes[iP] > iaCHashes[iC]) {
iC++;
      } else {
iP++;
      }
    }
  } /* end while !foundCluster && iP && iC */

  return (topScore);
}


void writeClusters(FILE * fd, Primary * head, int bpl, int *nOrph,
   char *inFiles[], int div[], int nCF, int vPrimary)
{
  Primary *p;
  Secondary *s;
  int orph;
  int nblanks = 1; /* num blank lines to put between sequences */
```

```
  int i, j;
  FILE *fd_out;
  char newCF[MAXFNAME + 1];


  p = head;
  orph = 0;

  /* write new versions of previously clustered files */
  if (nCF > 0) {
    j = 0;
    for (i = 0; i < nCF; i++) {

      if (strlen(inFiles[i]) > (MAXFNAME - 5)) {
eprintf(" output filename too long.\n");
      }
      strcpy(newCF, inFiles[i]);
      strcat(newCF, ".out");

      printf("  Writing %s...\n", newCF);
      fd_out = fopen(newCF, "wc");
      if (fd_out == NULL)
eprintf("can't open %s:", newCF);

      while (j < div[i]) {
fprintf(fd_out, "@P: %s %d\n", p->name, p->clusID);
printSeq(fd_out, p->seq, bpl, nblanks);
if (vPrimary == 1) {
  fprintf(fd_out, "@VP: %d %d %d %d %d\n", p->nInternal,
  p->nBothExt, p->nFrontExt, p->nTailExt, p->nProblems);
  printSeq(fd_out, p->vp, bpl, nblanks);
}
s = p->headS;
while (s != NULL) {
  fprintf(fd_out, "@S: %s %d %d %d %f ", s->name, s->iP, s->iS,
  s->matchLen, s->score);
  if (s->dir == FORWARD)
    fprintf(fd_out, "%s", "FORWARD ");
  else
    fprintf(fd_out, "%s", "REVCOMP ");
  if (s->score == 0) {
    fprintf(fd_out, "%s", "ORPHAN");
    ++orph;
  }
  fprintf(fd_out, "\n");
  printSeq(fd_out, s->seq, bpl, nblanks);
  s = s->next;
}

p = p->next;
++j;
      }
      fclose(fd_out);
    }
  }

  while (p != NULL) {
    fprintf(fd, "@P: %s\n", p->name);
    printSeq(fd, p->seq, bpl, nblanks);
    if (vPrimary == 1) {
      fprintf(fd, "@VP: %d %d %d %d %d\n", p->nInternal, p->nBothExt,
      p->nFrontExt, p->nTailExt, p->nProblems);
      printSeq(fd, p->vp, bpl, nblanks);
    }
```

```
    s = p->headS;
    while (s != NULL) {
      fprintf(fd, "@S: %s %d %d %d %f ", s->name, s->iP, s->iS,
      s->matchLen, s->score);
      if (s->dir == FORWARD)
fprintf(fd, "%s", "FORWARD ");
      else
fprintf(fd, "%s", "REVCOMP ");
      if (s->score == 0) {
fprintf(fd, "%s", "ORPHAN");
++orph;
      }
      fprintf(fd, "\n");
      printSeq(fd, s->seq, bpl, nblanks);
      s = s->next;
    }
    p = p->next;
  }
  *nOrph = orph;
}


/* simple insertion sort to sort hit list */
void sortHitList(Hit_p hits, int nHits)
{
  int i;
  Hit_p a = hits;
  int l = 0;
  int r = nHits - 1;

  for (i = r; i > l; i--) {
    if (a[i].score > a[i - 1].score) {
      Hit t = a[i - 1];
      a[i - 1] = a[i];
      a[i] = t;
    }
  }
  for (i = l + 2; i <= r; i++) {
    int j = i;
    Hit v = a[i];
    while (v.score > a[j - 1].score) {
      a[j] = a[j - 1];
      j--;
    }
    a[j] = v;
  }
}
```

A.2.3 compare.c

```
/*************************************************************************
                 compare.c - routines to compare sequences
                 -----------------------------------------
    begin                : Mon Dec 13 1999
    author               : Tom Casavant, modified by Kevin Pedretti
    email                : pedretti@eng.uiowa.edu
 *************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "uicluster.h"
#include "options.h"
#include "fasta.h"
#include "cluster.h"
#include "compare.h"
```

```
#include "utils.h"

int ScoreMatch(char *strPattern, int strPatternIndex,
       char *strSubject, int strSubjectIndex,
       int iLengthToMatch,
       int iWrongLimit, int iGapLimit, int iWrongOrGapLimit,
       int iWrongPenalty, int iGapPenalty,
       int iRecursiveFlag,
       int *nWrong_p, int *nMissing_p, int *nInserted_p)
{

  int _iScore, _iMissingScore, _iWrongScore, _iInseredScore;
  static int iLocalWrong, iLocalMissing, iLocalInserted;

  /* Initialize */
  if (!iRecursiveFlag) {
    iLocalWrong = iLocalMissing = iLocalInserted = 0;
    *nWrong_p = iWrongLimit;
    *nMissing_p = *nInserted_p = iGapLimit;
  }

  if ((iLengthToMatch == 0) || (iLocalWrong > iWrongLimit)
      || ((iLocalMissing + iLocalInserted) > iGapLimit)
      || ((iLocalMissing + iLocalInserted + iLocalWrong) >
  iWrongOrGapLimit)) {
    /* base of recursion */
    if ((iLengthToMatch == 0) && (iLocalWrong <= iWrongLimit)
&& ((iLocalMissing + iLocalInserted) <= iGapLimit)
&& ((iLocalMissing + iLocalInserted + iLocalWrong) <=
    iWrongOrGapLimit)) {
      if ((iLocalWrong + iLocalMissing + iLocalInserted) <
  ((*nMissing_p) + (*nInserted_p) + (*nWrong_p))) {
*nWrong_p = iLocalWrong;
*nMissing_p = iLocalMissing;
*nInserted_p = iLocalInserted;
      }
      _iScore = 0;
      return (_iScore);
    } else {
      _iScore = 0;
      return (_iScore);
    }
  } else { /* recursion  */
    if (strPattern[strPatternIndex] == strSubject[strSubjectIndex]) {

      _iScore =
  1 + ScoreMatch(strPattern, strPatternIndex + 1, strSubject,
 strSubjectIndex + 1, iLengthToMatch - 1,
 iWrongLimit, iGapLimit, iWrongOrGapLimit,
 iWrongPenalty, iGapPenalty, TRUE,/* iRecursiveFlag
 */
 nWrong_p, nMissing_p, nInserted_p);
      return (_iScore);
    } else { /* test for a possible missing, inserted or
   wrong base */
      iLocalWrong += iWrongPenalty;
      _iWrongScore =
  ScoreMatch(strPattern, strPatternIndex + 1, strSubject,
     strSubjectIndex + 1, iLengthToMatch - 1, iWrongLimit,
     iGapLimit, iWrongOrGapLimit, iWrongPenalty,
     iGapPenalty, TRUE,/* iRecursiveFlag */
     nWrong_p, nMissing_p, nInserted_p);
      iLocalWrong -= iWrongPenalty;
      iLocalMissing += iGapPenalty;
      _iMissingScore =
```

```
  ScoreMatch(strPattern, strPatternIndex + 1, strSubject,
     strSubjectIndex, iLengthToMatch - 1, iWrongLimit,
     iGapLimit, iWrongOrGapLimit, iWrongPenalty,
     iGapPenalty, TRUE,/* iRecursiveFlag */
     nWrong_p, nMissing_p, nInserted_p);
      iLocalMissing -= iGapPenalty;
      iLocalInserted += iGapPenalty;
      _iInseredScore =
  ScoreMatch(strPattern, strPatternIndex, strSubject,
     strSubjectIndex + 1, iLengthToMatch, iWrongLimit,
     iGapLimit, iWrongOrGapLimit, iWrongPenalty,
     iGapPenalty, TRUE,/* iRecursiveFlag */
     nWrong_p, nMissing_p, nInserted_p);
      iLocalInserted -= iGapPenalty;

      if ((_iWrongScore >= _iMissingScore)
  && (_iWrongScore >= _iInseredScore)) {
_iScore = _iWrongScore;
      } else {
if (_iMissingScore >= _iInseredScore) {
  _iScore = _iMissingScore;
} else { /* (_iInseredScore > _iMissingScore) */
  _iScore = _iInseredScore;
}
      }
      return (_iScore);
    }
  } /* end of recursion subcase */
} /* end of ScoreMatch */
```

```
static int *tmpCHashPrefix = NULL;
static int *tmpCHashPrefixIndex = NULL;
static int *tmpPHashPrefix = NULL;
static int *tmpPHashPrefixIndex = NULL;
```

```
void extendMatch(char *pSeq, char *cSeq,
 int topScore, int lengthToMatch, int hashSize,
 int radix, int skip, char maskedChar,
 int wrongLimit, int gapLimit, int wrongOrGapLimit,
 int topPIndex, int topCIndex,
 int wrongPenalty, int gapPenalty,
 int *totalErrors, int *bestLength)
{

  int growTryLength;
  int minGrowWindowC, minGrowWindowP, growWindowC, growWindowP;
  int numCHashes, numPHashes;
  int growing, foundGrowStart;
  int maskedIndex;
  int prevCandPIndex, prevCandCIndex, candCIndex, candPIndex;
  int score;
  int nFoundWrong, nFoundMissing, nFoundInserted;
  int growWrongLimit, growGapLimit, growWrongOrGapLimit = 0;

  *totalErrors = lengthToMatch - topScore;
```

```
/* Then we found a Primary that matched the current Candidate. The *
   approach is to try to append regions of lengthToMatch until the end * of
   one sequence or until a shorter segment has to be appended. This * can
   heppen for several reasons: a) the one sequence or the other is * too
   short, b) the hases are -1 in the end of the region most * recently
   appended, or c) the new overlapping region is shorter * than the
   attempted length being appended.  To do this we have to * re-sort the
   hashes according to index instead of by hash value. We * use a couple of
   tmp arrays to hold these indices.  */

if (tmpPHashPrefix == NULL) {
  tmpPHashPrefix = (int *) emalloc(MAXSLEN * sizeof(int));
  tmpCHashPrefix = (int *) emalloc(MAXSLEN * sizeof(int));
  tmpPHashPrefixIndex = (int *) emalloc(MAXSLEN * sizeof(int));
  tmpCHashPrefixIndex = (int *) emalloc(MAXSLEN * sizeof(int));
}

numPHashes =
    hashSeq(pSeq, hashSize, skip, 0, tmpPHashPrefix,
    tmpPHashPrefixIndex);
numCHashes =
    hashSeq(cSeq, hashSize, skip, 0, tmpCHashPrefix,
    tmpCHashPrefixIndex);

growing = TRUE;
*bestLength = topScore;
prevCandPIndex = -1;
prevCandCIndex = -1;
candCIndex = topCIndex + *bestLength;
candPIndex = topPIndex + *bestLength;

/* This is the main loop which continues to add segments as * long as the
   previous segment added was of maximum length. * As soon as the attempted
   length to be added is shorter than * the maximum length (lengthToMatch)
   we set the growing flag * to FALSE. Note, that this is initially true
   before entering * this loop and that there are many ways that growth can
   end. * It is important that in each clause below, if an action is *
   taken or condition detected, that indicates that further * growth will
   not be possible after trying to match the * current segment, we have to
   set the growing flag to FALSE. */

while (growing) {

  /* Now we must bound the new attempted growth region by the * length of
     the shorter of the two sequences */

  growTryLength = lengthToMatch;
  if ((candCIndex + growTryLength) > strlen(cSeq)) {
    growTryLength = strlen(cSeq) - candCIndex;
    growing = FALSE;
  }
  if ((candPIndex + growTryLength) > strlen(pSeq)) {
    growTryLength = strlen(pSeq) - candPIndex;
    growing = FALSE;
  }

  /* Now the maximum length of the attempted growth region is * determined.
     Next we have to determine the starting * position in the currently
     grown region to begin the * extension. This is done by comparing the
     hash values in the * region just before the end of the previous
     segment to have * been appended. HOWEVER, before that, we must make
     sure we * arent comparing FAILED_HASH values. The first segment of *
     code below scans backward over the FAILE_DHASH values, and * the
     second segment looks backward from there for a hash * match. The first
     block may result in setting growing to * FALSE. The second block may
```

```
       not. */

    /* First block: Find the non-failed hashes region */
    minGrowWindowC = 0;
    while ((minGrowWindowC < growTryLength)
   && (tmpCHashPrefix[candCIndex - skip - minGrowWindowC] ==
       FAILEDHASH)) {
      minGrowWindowC++;
    }
    minGrowWindowP = 0;
    while ((minGrowWindowP < growTryLength)
   && (tmpPHashPrefix[candPIndex - skip - minGrowWindowP] ==
       FAILEDHASH)) {
      minGrowWindowP++;
    }

    if ((minGrowWindowC > 0) || (minGrowWindowP > 0)) {
      growing = FALSE;
    }

    /* Now find the first matching non-failed hash */
    /* The next 2 lines are needed in case the first loop is never *
       executed. For instance, if the hashes match right away * -- which
       "should" be common */
    growWindowP = minGrowWindowP;
    growWindowC = minGrowWindowC;
    foundGrowStart = (tmpCHashPrefix[candCIndex - skip - growWindowC]
      == tmpPHashPrefix[candPIndex - skip - growWindowP]);
    for (growWindowC = minGrowWindowC;
 (growWindowC < growTryLength) && (!foundGrowStart); growWindowC++) {
      for (growWindowP = minGrowWindowP;
   (growWindowP < growTryLength) && (!foundGrowStart);
   growWindowP++) {
foundGrowStart = (tmpCHashPrefix[candCIndex - skip - growWindowC]
  == tmpPHashPrefix[candPIndex - skip -
    growWindowP]);

      } /* end for growWindowP */
    } /* end for growWindowC */

    candCIndex -= growWindowC;
    candPIndex -= growWindowP;

    /* Now we safeguard that we don't have a pathological case in which * the
       new candidate site is identical to the previous one with just * enough
       errors in the previous grow region to make them look * different */
    if ((prevCandPIndex == candPIndex) || (prevCandCIndex == candCIndex)) {
      foundGrowStart = FALSE;
    } else {
      prevCandPIndex = candPIndex;
      prevCandCIndex = candCIndex;
    }

    if (foundGrowStart) {
      growWrongLimit =
  ((double) wrongLimit / (double) lengthToMatch) *
  (double) growTryLength;
      growWrongLimit = min2((growWrongLimit + 1), (growTryLength - 1));
      growGapLimit =
  ((double) gapLimit / (double) lengthToMatch) *
  (double) growTryLength;
      growGapLimit = min2((growGapLimit + 1), (growTryLength - 1));
      growWrongOrGapLimit =
  ((double) wrongOrGapLimit / (double) lengthToMatch) *
  (double) growTryLength;
```

```
      growWrongOrGapLimit =
  min2((growWrongOrGapLimit + 1), (growTryLength - 1));

      score = ScoreMatch(pSeq, candPIndex, cSeq, candCIndex, growTryLength,
 growWrongLimit, growGapLimit, growWrongOrGapLimit,
 wrongPenalty, gapPenalty, FALSE,/* recursiveFlag
 */
 &nFoundWrong, &nFoundMissing, &nFoundInserted);
    } else {
      growTryLength = 0;
      growing = FALSE;
      score = 0;
    }

    if ((score > 0) && (score < (growTryLength - growWrongOrGapLimit))) {
      growing = FALSE;
      while (score < (growTryLength - growWrongOrGapLimit)
     && (growTryLength > 1)) {
growTryLength--;
growWrongLimit =
    ((double) wrongLimit / (double) lengthToMatch) *
    (double) growTryLength;
growWrongLimit = min2((growWrongLimit + 1), (growTryLength - 1));
growGapLimit =
    ((double) gapLimit / (double) lengthToMatch) *
    (double) growTryLength;
growGapLimit = min2((growGapLimit + 1), (growTryLength - 1));
growWrongOrGapLimit =
    ((double) wrongOrGapLimit / (double) lengthToMatch) *
    (double) growTryLength;
growWrongOrGapLimit =
    min2((growWrongOrGapLimit + 1), (growTryLength - 1));

score =
    ScoreMatch(pSeq, candPIndex, cSeq, candCIndex, growTryLength,
       growWrongLimit, growGapLimit, growWrongOrGapLimit,
       wrongPenalty, gapPenalty, FALSE,/* recursiveFlag */
       &nFoundWrong, &nFoundMissing, &nFoundInserted);
      }
    }

    *bestLength += (growTryLength - max2(growWindowC, growWindowP));

    // printf("Best Length: %d\n", *bestLength);

    *totalErrors += (growTryLength - score);
    candCIndex += (growTryLength - growWindowC);
    candPIndex += (growTryLength - growWindowP);

    maskedIndex = 0;
    while ((cSeq[candCIndex + maskedIndex] == maskedChar)
   && (pSeq[candPIndex + maskedIndex] == maskedChar)) {
      maskedIndex++;
    }
    if ((maskedIndex > 0)
&& (cSeq[candCIndex + maskedIndex] ==
    pSeq[candPIndex + maskedIndex])) {
      candCIndex += maskedIndex;
      candPIndex += maskedIndex;
      *bestLength += maskedIndex;
    }
    if ((cSeq[candCIndex] == '\0') || (pSeq[candPIndex] == '\0')) {
      growing = FALSE;
    }
  }
```

```
}
```

A.2.4 fasta.c

```
/*************************************************************************
               fasta.c  -  routines for working with fastA files
               -------------------------------------------------
    begin                : Sun Dec 12 1999
    author               : Kevin Pedretti
    email                : pedretti@eng.uiowa.edu
 *************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "uicluster.h"
#include "utils.h"
#include "fasta.h"
#include "options.h"
#include "cluster.h"
#include "memory.h"

static char *inLine; /* buffer to hold current input line */
static char *tag; /* the current tag line, sans > */
static char *inseq; /* buffer for current seq */

int readSeq(FILE * fd_fasta, FASTASeq * seq, FILE * fd_rej, int rc,
    int nRejects)
{
  int seqLen; /* length of current input seq */
  int fs; /* the index of the first space in the tag
   line */
  int i;
  char *status;

  if (inLine == NULL) {
    inLine = (char *) emalloc(MAXLINE * sizeof(char));
    tag = (char *) emalloc(MAXLINE * sizeof(char));
    inseq = (char *) emalloc(MAXSLEN * sizeof(char));
  }

  /* read until first sequence found */
  status = fgets(inLine, MAXLINE, fd_fasta);
  while ((status != NULL) && (inLine[0] != '>')) {
    inLine = fgets(inLine, MAXLINE, fd_fasta);
  }

  /* if the start of a sequence was found, read it into memory */
  if (inLine[0] == '>') {
    chomp(inLine);
    strcpy(tag, inLine + 1);

    /* read the sequence data */
    seqLen = 0;
    inseq[0] = '\0';
    status = fgets(inLine, MAXLINE, fd_fasta);
    while ((status != NULL) && (inLine[0] != '>')) {
      seqLen += strlen(chomp(inLine));
      strcat(inseq, inLine);
      status = fgets(inLine, MAXLINE, fd_fasta);
    }

    /* if the next sequence was found, put back inLine */
    if (inLine[0] == '>') {
      fseek(fd_fasta, -(strlen(inLine)), SEEK_CUR);
```

```
    }

    /* make sure the input sequence isn't too long */
    if (seqLen >= MAXSLEN) {
      eprintf("Sequence %s > MAXSLEN (%d)\n", tag, seqLen);
    }

    /* check to see if input sequence is a reject */
    if (countBases(inseq) <= rc) {
      /* if we have a handle on the reject file, output the reject to it */
      if (fd_rej != NULL) {
fprintf(fd_rej, "%d. %s\n", nRejects + 1, tag);
printSeq(fd_rej, inseq, NBASESONLINE, 1);
      }
      return REJECT_SEQ;
    }

    /* find the first space */
    i = fs = 0;
    while ((i < MAXSNAME) && (tag[i] != ' ')) {
      i++;
    }
    fs = i;
    printf("fs=%d, %s\n", fs, tag);

    /* store the sequence name */
    if (fs == 0) {
      fs = MAXSNAME - 1;
    }
    strncpy(seq->name, tag, fs);
    seq->name[fs] = '\0';
    printf("\n1: %s\n", seq->name);

    /* store the sequence */
    strcpy(seq->seq, inseq);

    return GOOD_SEQ;
  }

  /* if we got here, there are no more seqs */
  return NO_MORE_SEQS;
}

/* readSeqs: reads fasta seqs into linked list rejecting
             those with fewer than rc bases */
void readSeqs(FILE * fd, FILE * fdRej, int rc, FASTASeq ** head,
      int *nSeqs, int *nRej)
{

  char *inLine; /* buffer to hold current input line */
  char *tag; /* the current tag line, sans > */
  char *seq; /* buffer for current seq */
  int seqLen; /* length of current input seq */
  FASTASeq *tail; /* tail seq in list */
  FASTASeq *tmp; /* current input seq */
  int ns = 0; /* number of sequences read */
  int rej = 0; /* number of sequences rejected (shorter than
   rc) */
  int fs; /* the index of the first space in the tag
   line */
  int i;
  char *status;

  inLine = (char *) emalloc(MAXLINE * sizeof(char));
  tag = (char *) emalloc(MAXLINE * sizeof(char));
```

```
  seq = (char *) emalloc(MAXSLEN * sizeof(char));

  *head = tail = NULL;

  while ((inLine = fgets(inLine, MAXLINE, fd)) != NULL) {
    if (inLine[0] == '>') {
      ++ns;
      chomp(inLine);
      strcpy(tag, inLine + 1);

      seqLen = 0;
      seq[0] = '\0';
      status = fgets(inLine, MAXLINE, fd);
      while ((inLine[0] != '>') && (status != NULL)) {
seqLen += strlen(chomp(inLine));
strcat(seq, inLine);
status = fgets(inLine, MAXLINE, fd);
      }
      /* Rewind to the begining of the line */
      fseek(fd, -(strlen(inLine)), SEEK_CUR);

      if (seqLen >= MAXSLEN) {
eprintf("Sequence %s > MAXSLEN (%d)\n", tag, seqLen);
      }

      if (countBases(seq) > rc) {
tmp = getFASTASeq();
fs = 0;

i = 0;
while ((i < MAXSNAME) && (tag[i] != ' ')) {
  i++;
}
fs = i;

if (fs == 0) {
  fs = MAXSNAME - 1;
}
strncpy(tmp->name, tag, fs);
tmp->seq = (char *) emalloc((seqLen + 1) * sizeof(char));
strcpy(tmp->seq, seq);
tmp->next = NULL;

if (*head == NULL) {
  *head = tail = tmp;
} else {
  tail->next = tmp;
  tail = tmp;
}
      } else {
if (fdRej != NULL) {
  ++rej;
  fprintf(fdRej, "%d. %s\n", rej, tag);
  printSeq(fdRej, seq, NBASESONLINE, 1);
}
      }
    }
  }
  *nSeqs = ns;
  *nRej = rej;

  free(inLine);
  free(tag);
  free(seq);
}
```

```
/* printSeq: print a fasta sequence */
void printSeq(FILE * fd, char *seq, int bpl, int nblanks)
{

  int i, j = 0;
  int len = strlen(seq);
  char *ptr;

  /* // original code for(i=0; i<len; i++) { putc (seq[i], fd); ++j; if (j ==
     bpl) { putc ('\n', fd); j = 0; } } */

  ptr = seq;
  for (i = 0; i < len; i++) {
    putc(*ptr, fd);
    ++j;
    ++ptr;
    if (j == bpl) {
      putc('\n', fd);
      j = 0;
    }
  }

  if (j != 0)
    ++nblanks;
  for (i = 0; i < nblanks; i++)
    putc('\n', fd);
}
```

A.2.5 incremental.c

```
/***************************************************************************
             incremental.c  -  routines for incremental clustering
             -------------------------------------------------
    begin                : Wed Jan 12 2000
    author               : Kevin Pedretti
    email                : pedretti@eng.uiowa.edu
 ***************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "uicluster.h"
#include "utils.h"
#include "fasta.h"
#include "options.h"
#include "cluster.h"
#include "memory.h"
#include "incremental.h"

int readClusFiles(char *cfname, Primary ** head, Primary ** tail,
  char *inFilesOut[], int div[])
{

  FILE *fd_cf;
  char *inFiles[MAXPRE];
  int i;
  int nCF;
  int nPrimes;
  int nSecnds;

  /* handle case where no previously cluster files are input */
  if (strcmp(cfname, "none") == 0)
    return 0;
```

```
  for (i = 0; i < MAXPRE; i++) {
    div[i] = -1;
    inFiles[i] = (char *) emalloc((MAXFNAME + 1) * sizeof(char));
  }

  fd_cf = fopen(cfname, "r");
  if (fd_cf == NULL)
    eprintf("can't open %s:", cfname);

  i = 0;
  while ((inFiles[i] = fgets(inFiles[i], MAXFNAME, fd_cf)) != NULL) {
    chomp(inFiles[i]);
    ++i;
    if (i > MAXPRE) {
      eprintf(" too many previously clustered files.");
    }
  }
  nCF = i;
  fclose(fd_cf);

  /* read in the primaries from the pre-clustered files */
  *head = *tail = NULL;
  nPrimes = nSecnds = 0;
  for (i = 0; i < nCF; i++) {
    printf("  reading %s: ", inFiles[i]);
    fd_cf = fopen(inFiles[i], "r");
    if (fd_cf == NULL)
      eprintf("can't open %s:", inFiles[i]);
    nPrimes = nSecnds = 0;
    parseClusFile(fd_cf, head, tail, &nPrimes, &nSecnds);
    printf("%d primaries, %d secondaries\n", nPrimes, nSecnds);
    if (i == 0)
      div[i] = nPrimes;
    else
      div[i] = div[i - 1] + nPrimes;
    fclose(fd_cf);
  }

  for (i = 0; i < nCF; i++) {
    printf("%s\n", inFiles[i]);
    inFilesOut[i] = inFiles[i];
  }
  return (nCF);
}

int parseClusFile(FILE * fd, Primary ** head, Primary ** tail,
  int *nPrimes, int *nSecnds)
{
  char *inLine; /* buffer to hold current input line */
  char *tag; /* the current tag line, sans @P */
  char *seq; /* buffer for current primary seq */
  int seqLen; /* length of current input primary seq */
  Primary *tmp; /* current input primary */
  Secondary *tmpS; /* current input secondary */
  Secondary *s;
  int fs; /* the index of the first space in the tag
   line */
  int i;
  char *status;

  inLine = (char *) emalloc(MAXLINE * sizeof(char));
  tag = (char *) emalloc(MAXLINE * sizeof(char));
  seq = (char *) emalloc(MAXSLEN * sizeof(char));

  *nPrimes = *nSecnds = 0;
```

```
  while ((inLine = fgets(inLine, MAXLINE, fd)) != NULL) {
    if (inLine[0] == '@' && inLine[1] == 'P') {
      ++(*nPrimes);
      chomp(inLine);
      strcpy(tag, inLine + 4);
      seqLen = 0;
      seq[0] = '\0';
      status = fgets(inLine, MAXLINE, fd);
      while ((status != NULL) && (inLine[0] != '@')) {
seqLen += strlen(chomp(inLine));
strcat(seq, inLine);
status = fgets(inLine, MAXLINE, fd);
      }
      /* Rewind to the begining of the line */
      fseek(fd, -(strlen(inLine)), SEEK_CUR);

      if (seqLen >= MAXSLEN) {
eprintf(" Sequence %s > MAXSLEN (%d)\n", tag, seqLen);
      }

      tmp = getPrimary();

      fs = 0;
      i = 0;
      while ((i < MAXSNAME) && (tag[i] != ' ')) {
i++;
      }
      fs = i;

      if (fs == 0) {
fs = MAXSNAME - 1;
      }
      tmp->name = (char *) emalloc((MAXSNAME + 1) * sizeof(char));
      strncpy(tmp->name, tag, fs);
      tmp->seq = (char *) emalloc((seqLen + 1) * sizeof(char));
      strcpy(tmp->seq, seq);
      tmp->next = NULL;
      tmp->nHashes = 0;
      tmp->nTouched = 0;
      tmp->nSecondaries = 0; /* fixme */
      tmp->hashes = NULL;
      tmp->indexes = NULL;
      tmp->headS = NULL;
      tmp->nextCP = NULL;

      if (*head == NULL) {
*head = *tail = tmp;
      } else {
(*tail)->next = tmp;
*tail = tmp;
      }
    } else if (inLine[0] == '@' && inLine[1] == 'S') {
      ++(*nSecnds);
      chomp(inLine);
      strcpy(tag, inLine + 4);
      seqLen = 0;
      seq[0] = '\0';
      status = fgets(inLine, MAXLINE, fd);
      while ((status != NULL) && (inLine[0] != '@')) {
seqLen += strlen(chomp(inLine));
strcat(seq, inLine);
status = fgets(inLine, MAXLINE, fd);
      }
      /* Rewind to the begining of the line */
```

```
      fseek(fd, -(strlen(inLine)), SEEK_CUR);

      if (seqLen >= MAXSLEN) {
eprintf(" Sequence %s > MAXSLEN (%d)\n", tag, seqLen);
      }

      tmpS = getSecondary();

      fs = 0;
      i = 0;
      while ((i < MAXSNAME) && (tag[i] != ' ')) {
++i;
      }
      fs = i;
      if (fs == 0) {
fs = MAXSNAME - 1;
      }

      tmpS->name = (char *) emalloc((MAXSNAME + 1) * sizeof(char));
      strncpy(tmpS->name, tag, fs);
      tmpS->seq = (char *) emalloc((seqLen + 1) * sizeof(char));
      strcpy(tmpS->seq, seq);

      tmpS->next = NULL;

      sscanf(tag + fs, "%d %d %d %lf", &(tmpS->iP), &(tmpS->iS),
     &(tmpS->matchLen), &(tmpS->score));

      if (tag[strlen(tag) - 2] == 'D') {
tmpS->dir = FORWARD;
      } else if (tag[strlen(tag) - 2] == 'P') {
tmpS->dir = REVCOMP;
      }

      /* Add the secondary to the current primary */
      if (tmp->headS == NULL) {
tmp->headS = tmpS;
      } else {
s = tmp->headS;
while (s->next != NULL) {
  s = s->next;
}
s->next = tmpS;
      }
      ++(tmp->nSecondaries);
    } else if (inLine[0] == '@' && inLine[1] == 'V') {
      chomp(inLine);
      strcpy(tag, inLine + 5);

      /* read the sequence */
      seqLen = 0;
      seq[0] = '\0';
      status = fgets(inLine, MAXLINE, fd);
      while ((status != NULL) && (inLine[0] != '@')) {
seqLen += strlen(chomp(inLine));
strcat(seq, inLine);
status = fgets(inLine, MAXLINE, fd);
      }
      /* Rewind to the begining of the line */
      fseek(fd, -(strlen(inLine)), SEEK_CUR);

      if (seqLen >= MAXSLEN) {
eprintf(" Sequence %s > MAXSLEN (%d)\n", tag, seqLen);
      }
```

```
      tmp->vp = (char *) emalloc((seqLen * 2) + 1);
      strcpy(tmp->vp, seq);
      tmp->vpLen = seqLen;
      tmp->maxvpLen = seqLen * 2;

      /* parse the tag line */
      sscanf(tag, "%d %d %d %d %d\n", tmp->nInternal, tmp->nBothExt,
     tmp->nFrontExt, tmp->nTailExt, tmp->nProblems);
    }
  }

  free(inLine);
  free(tag);
  free(seq);

  return (*nPrimes);
}
```

A.2.6 memory.c

```
/***************************************************************************
                  memory.c  -  memory block allocation routines
                  -------------------------------------------
    begin                : Tue Dec 14 1999
    author               : Kevin Pedretti
    email                : pedretti@eng.uiowa.edu
 ***************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "uicluster.h"
#include "utils.h"
#include "fasta.h"
#include "options.h"
#include "cluster.h"
#include "memory.h"

static GHTEntry *ghtMem;
static Primary *pMem;
static Secondary *sMem;
static FASTASeq *fsMem;
/*static char      *seqMem; */
static int ghtIndex = GHTBLOCK;
static int fsIndex = FASTABLOCK;
static int pIndex = PRIMEBLOCK;
static int sIndex = SECNDBLOCK;

GHTEntry *getGHTEntry()
{

  GHTEntry *tmp;

  if (ghtIndex >= GHTBLOCK) {
    /* Array is full, make a new one. */
    ghtMem = (GHTEntry *) emalloc(GHTBLOCK * sizeof(GHTEntry));
    ghtIndex = 0;
  }
  tmp = ghtMem + ghtIndex;
  ghtIndex++;
  return (tmp);
}

Primary *getPrimary()
{
```

```
  Primary *tmp;

  if (pIndex >= PRIMEBLOCK) {
    /* Array is full, make a new one. */
    pMem = (Primary *) emalloc(PRIMEBLOCK * sizeof(Primary));
    pIndex = 0;
  }
  tmp = pMem + pIndex;
  pIndex++;
  return (tmp);
}

Secondary *getSecondary()
{

  Secondary *tmp;

  if (sIndex >= SECNDBLOCK) {
    /* Array is full, make a new one. */
    sMem = (Secondary *) emalloc(SECNDBLOCK * sizeof(Secondary));
    sIndex = 0;
  }
  tmp = sMem + sIndex;
  sIndex++;
  return (tmp);
}


FASTASeq *getFASTASeq()
{

  FASTASeq *tmp;

  if (fsIndex >= FASTABLOCK) {
    /* Array is full, make a new one. */
    fsMem = (FASTASeq *) emalloc(FASTABLOCK * sizeof(FASTASeq));
    fsIndex = 0;
  }
  tmp = fsMem + fsIndex;
  fsIndex++;
  return (tmp);
}
```

A.2.7 options.c

```
/***************************************************************************
                   options.c  - parse options for UIcluster
                   ----------------------------------------
    begin                : Sun Dec 12 1999
    author               : Kevin Pedretti
    email                : pedretti@eng.uiowa.edu
 ***************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "getopt.h"
#include "uicluster.h"
#include "options.h"
#include "utils.h"

/* called by main to get the user-defined parameters. options can
   come from either the command line or the configuration file. the
   command line has priority
*/
int getopts(Options * o, int argc, char **argv)
```

```
{

  int c;
  int option_index = 0;

  static struct option long_opts[] = {
    {"preClus", 1, 0, 0},
    {"rejCrit", 1, 0, 0},
    {"hashSize", 1, 0, 0},
    {"startSkip", 1, 0, 0},
    {"endSkip", 1, 0, 0},
    {"matchLen", 1, 0, 0},
    {"errLimit", 1, 0, 0},
    {"maskChar", 1, 0, 0},
    {"repick", 0, 0, 0},
    {"tryRevC", 0, 0, 0},
    {"hitThresh", 1, 0, 0},
    {"wrongPen", 1, 0, 0},
    {"gapPen", 1, 0, 0},
    {"wrongPen", 1, 0, 0},
    {"keepGoing", 0, 0, 0},
    {"vPrimary", 0, 0, 0},
    {"help", 0, 0, 0},
    {0, 0, 0, 0}
  };
  /* These aren't very good short option names */
  static char *short_opts = "F:R:H:S:s:M:E:C:h:P:p:";

  /* initially set options to hard-coded defaults */
  strcpy((*o).preCFile, "none");
  (*o).rejCrit = 100;
  (*o).hashSize = 8;
  (*o).startSkip = 18;
  (*o).endSkip = 0;
  (*o).lenToMatch = 40;
  (*o).errLimit = 2;
  (*o).maskChar = 'X';
  (*o).repick = 0;
  (*o).tryRevC = 0;
  (*o).hitThresh = 16;
  (*o).wrongPen = 1;
  (*o).gapPen = 1;
  (*o).keepGoing = 0;
  (*o).vPrimary = 0;

  /* parse command line */
  c = getopt_long(argc, argv, short_opts, long_opts, &option_index);
  while (c != -1) {
    switch (c) {
    case 0:
      if (strcmp(long_opts[option_index].name, "preClus") == 0) {
strcpy((*o).preCFile, optarg);
      }
      if (strcmp(long_opts[option_index].name, "rejCrit") == 0) {
(*o).rejCrit = atoi(optarg);
      }
      if (strcmp(long_opts[option_index].name, "hashSize") == 0) {
(*o).hashSize = atoi(optarg);
      }
      if (strcmp(long_opts[option_index].name, "startSkip") == 0) {
(*o).startSkip = atoi(optarg);
      }
      if (strcmp(long_opts[option_index].name, "endSkip") == 0) {
(*o).endSkip = atoi(optarg);
      }
```

```
      if (strcmp(long_opts[option_index].name, "matchLen") == 0) {
(*o).lenToMatch = atoi(optarg);
      }
      if (strcmp(long_opts[option_index].name, "errLimit") == 0) {
(*o).errLimit = atoi(optarg);
      }
      if (strcmp(long_opts[option_index].name, "maskChar") == 0) {
if (strlen(optarg) == 1) {
  (*o).maskChar = optarg[0];
} else {
  wprintf(" mask char can only be one letter");
  return (FAILURE);
}
      }
      if (strcmp(long_opts[option_index].name, "hitThresh") == 0) {
(*o).hitThresh = atoi(optarg);
      }
      if (strcmp(long_opts[option_index].name, "wrongPen") == 0) {
(*o).wrongPen = atoi(optarg);
      }
      if (strcmp(long_opts[option_index].name, "gapPen") == 0) {
(*o).gapPen = atoi(optarg);
      }
      if (strcmp(long_opts[option_index].name, "tryRevC") == 0) {
(*o).tryRevC = 1;
      }
      if (strcmp(long_opts[option_index].name, "repick") == 0) {
(*o).repick = 1;
      }
      if (strcmp(long_opts[option_index].name, "keepGoing") == 0) {
(*o).keepGoing = 1;
      }
      if (strcmp(long_opts[option_index].name, "vPrimary") == 0) {
(*o).vPrimary = 1;
      }

      if (strcmp(long_opts[option_index].name, "help") == 0) {
printUsage();
return FAILURE;
      }
      break;

    case 'F':
      strcpy((*o).preCFile, optarg);
      break;
    case 'R':
      (*o).rejCrit = atoi(optarg);
      break;
    case 'H':
      (*o).hashSize = atoi(optarg);
      break;
    case 'S':
      (*o).startSkip = atoi(optarg);
      break;
    case 's':
      (*o).endSkip = atoi(optarg);
      break;
    case 'M':
      (*o).lenToMatch = atoi(optarg);
      break;
    case 'E':
      (*o).errLimit = atoi(optarg);
      break;
    case 'C':
      if (strlen(optarg) == 1) {
```

```
(*o).maskChar = optarg[0];
      } else {
wprintf(" mask char can only be one letter");
return (FAILURE);
      }
      break;
    case 'h':
      (*o).hitThresh = atoi(optarg);
      break;
    case 'P':
      (*o).wrongPen = atoi(optarg);
      break;
    case 'p':
      (*o).gapPen = atoi(optarg);
      break;
    default:
      wprintf(" unexpected command line option option -- %o", c);
      return (FAILURE);
    }
    c = getopt_long(argc, argv, short_opts, long_opts, &option_index);
  }

  /* get the name of the input fasta file */
  if (optind == (argc - 1)) {
    strcpy((*o).inFile, argv[optind]);
  } else if (optind == (argc)) {
    wprintf(" no input FASTA file specified\n");
    printUsage();
    return FAILURE;
  } else if (optind < (argc - 1)) {
    wprintf
(" too many non-option parameters -- only specify one input FASTA file.\n");
    printUsage();
    return FAILURE;
  }

  return SUCCESS;
}

/* prints usage information */
void printUsage()
{
  printf("%s %s Usage:   uicluster [options] input_fasta_file\n",
 progname(), getversion());
  printf("\n");
  printf("  Valid Options: (defaults are in parenthesis)");
  printf("\n");
  printf
      ("      -F, --preClus     specifies the preClustered infile    (none)\n");
  printf
      ("      -R, --rejCrit     specifies the rejection criteria  (100 bases)\n");
  printf
      ("      -H, --hashSize    specifies the hash size           (  8 bases)\n");
  printf
      ("      -S, --startSkip   specifies the start skip          ( 18 bases)\n");
  printf
      ("      -s, --endSkip     specifies the end skip            (  0 bases)\n");
  printf
      ("      -M, --matchLen    specifies the length to match     ( 40 bases)\n");
  printf
      ("      -E, --errLimit    specifies the error limit         (  2 bases)\n");
  printf
      ("      -C, --maskChar    specifies the mask character         ('X')\n");
  printf
      ("      -h, --hitThresh   specifies the hit threshold           (16)\n");
```

```
  printf
      ("      -P, --wrongPen    specifies the wrong penalty            (1)\n");
  printf
      ("      -p, --gapPen      specifies the gap penalty              (1)\n");
  printf
      ("      --repick          repick cluster primaries             (off)\n");
  printf
      ("      --tryRevC         check reverse compliment             (off)\n");
  printf
      ("      --keepGoing       perform exhaustive search            (off)\n");
  printf
      ("      --vPrimary        generate virtual primary             (off)\n");
  printf("      --help            view this message\n");
  printf("\n\n");
}

/* prints all user-defined parameters to the stream fd */
void printopts(FILE * fd, Options o)
{
  fprintf(fd, "Using Options:\n");
  fprintf(fd, "    Input File   = %s\n", o.inFile);
  fprintf(fd, "    Preclus File = %s\n", o.preCFile);
  fprintf(fd, "    Reject Crit  = %4d      Hash Size  = %4d\n", o.rejCrit,
  o.hashSize);
  fprintf(fd, "    Start Skip   = %4d      End Skip   = %4d\n",
  o.startSkip, o.endSkip);
  fprintf(fd, "    Match Len    = %4d      Error Lim  = %4d\n",
  o.lenToMatch, o.errLimit);
  fprintf(fd, "    Mask Char    = %4c      Repick     = %4d\n", o.maskChar,
  o.repick);
  fprintf(fd, "    Try RevC     = %4d      Hit Thresh = %4d\n", o.tryRevC,
  o.hitThresh);
  fprintf(fd, "    Wrong Pen    = %4d      Gap Pen    = %4d\n", o.wrongPen,
  o.gapPen);
  fprintf(fd, "    Keep Going   = %4d      vPrimary   = %4d\n",
  o.keepGoing, o.vPrimary);
}
```

A.2.8 qsort.c

```
/*************************************************************************
                    qsort.c  -  stand-alone quick sort
                    ----------------------------------
    begin                : Mon Dec 13 1999
    author               : Kevin Pedretti
    email                : pedretti@eng.uiowa.edu
 *************************************************************************/

#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include "qsort.h"

/* Do a quick sort on data[] while maintaining consistancy with index[].
   NOTE: This sort routine sorts data[1..n] NOT data[0..n-1].
         The caller should account for this.
   based on code in Numerical Recipes in C, Second Edition
*/

void qsortWIndx(unsigned long n, int data[], int index[])
{
  unsigned long i, ir = n, j, k, l = 1;
  int *istack, jstack = 0;
  int a, b, temp;
```

```
  istack = ivector(1, NSTACK);
  for (;;) {
    if (ir - l < M) {
      for (j = l + 1; j <= ir; j++) {
a = data[j];
b = index[j];
for (i = j - 1; i >= l; i--) {
  if (data[i] <= a)
    break;
  data[i + 1] = data[i];
  index[i + 1] = index[i];
}
data[i + 1] = a;
index[i + 1] = b;
      }
      if (!jstack) {
free_ivector(istack, 1, NSTACK);
return;
      }
      ir = istack[jstack];
      l = istack[jstack - 1];
      jstack -= 2;
    } else {
      k = (l + ir) >> 1;
      SWAP(data[k], data[l + 1])
  SWAP(index[k], index[l + 1])
  if (data[l] > data[ir]) {
SWAP(data[l], data[ir])
    SWAP(index[l], index[ir])
      }
      if (data[l + 1] > data[ir]) {
SWAP(data[l + 1], data[ir])
    SWAP(index[l + 1], index[ir])
      }
      if (data[l] > data[l + 1]) {
SWAP(data[l], data[l + 1])
    SWAP(index[l], index[l + 1])
      }
      i = l + 1;
      j = ir;
      a = data[l + 1];
      b = index[l + 1];
      for (;;) {
do
  i++;
while (data[i] < a);
do
  j--;
while (data[j] > a);
if (j < i)
  break;
SWAP(data[i], data[j])
    SWAP(index[i], index[j])
      }
      data[l + 1] = data[j];
      data[j] = a;
      index[l + 1] = index[j];
      index[j] = b;
      jstack += 2;
      if (jstack > NSTACK)
nrerror("NSTACK too small in sort2.");
      if (ir - i + 1 >= j - 1) {
istack[jstack] = ir;
istack[jstack - 1] = i;
ir = j - 1;
```

```
      } else {
istack[jstack] = j - 1;
istack[jstack - 1] = l;
l = i;
      }
    }
  }
}

/* display an error and exit */
void nrerror(char error_text[])
{
  fprintf(stderr, "Sorting run-time error...\n");
  fprintf(stderr, "%s\n", error_text);
  fprintf(stderr, "...now exiting to system...\n");
  exit(1);
}

/* allocate memory for an integer array */
int *ivector(long nl, long nh)
{
  int *v;
  v = (int *) malloc((size_t) ((nh - nl + 1 + NR_END) * sizeof(int)));
  if (!v)
    nrerror("allocation failure in ivector()");
  return v - nl + NR_END;
}

/* free an integer aray */
void free_ivector(int *v, long nl, long nh)
{
  free((FREE_ARG) (v + nl - NR_END));
}
```

A.2.9 utils.c

```
/**************************************************************************
                  utils.c  -  some useful utilities for UIcluster
                  -----------------------------------------------
    begin                : Sun Dec 12 1999
    author               : Kevin Pedretti
    email                : pedretti@eng.uiowa.edu
 **************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <unistd.h>
#include <time.h>
#include <string.h>
#include <errno.h>
#include "utils.h"

static char *name = NULL; /* Program name for messages */
static char *version = NULL; /* Program version */
static int mc; /* total num malloc calls made */

/* returns the total num of malloc calls made so far */
int nmallocs()
{
  return mc;
}

/* store name of program */
void setprogname(char *str)
```

```
{
  name = estrdup(str);
}

/* return stored name of program */
char *progname(void)
{
  return name;
}

/* store version number */
void setversion(char *str)
{
  version = estrdup(str);
}

/* return stored version number */
char *getversion(void)
{
  return version;
}

/* print error message and exit program */
void eprintf(char *fmt, ...)
{
  va_list args;

  fflush(stdout);
  if (progname() != NULL) {
    fprintf(stderr, "%s:", progname());
  }

  /* print the error message */
  va_start(args, fmt);
  vfprintf(stderr, fmt, args);
  va_end(args);

  /* print the errno if the put a ':' at end of fmt */
  if (fmt[0] != '\0' && fmt[strlen(fmt) - 1] == ':') {
    fprintf(stderr, " %s", strerror(errno));
  }
  fprintf(stderr, "\n");
  exit(EXIT_FAILURE);
}

/* print warning message */
void wprintf(char *fmt, ...)
{
  va_list args;

  fflush(stdout);
  if (progname() != NULL) {
    fprintf(stderr, "%s:", progname());
  }

  /* print the warning message */
  va_start(args, fmt);
  vfprintf(stderr, fmt, args);
  va_end(args);

  /* print the errno if the put a ':' at end of fmt */
  if (fmt[0] != '\0' && fmt[strlen(fmt) - 1] == ':') {
    fprintf(stderr, " %s", strerror(errno));
  }
  fprintf(stderr, "\n");
```

```
}

/* duplicate a string, terminate program if malloc error occurs */
char *estrdup(char *s)
{
  char *t;

  /* would like to know if we caused the malloc error so call malloc
     directly... not emalloc() */
  ++mc; /* increment malloc calls counter */
  t = (char *) malloc(strlen(s) + 1);
  if (t == NULL) {
    eprintf("estrdup(\"%.20s\") failed:", s);
  }
  strcpy(t, s);
  return t;
}

/* call malloc, terminate program if malloc error occurs */
void *emalloc(size_t n)
{
  void *p;

  ++mc; /* increment malloc calls counter */
  p = malloc(n);
  if (p == NULL) {
    printf("malloc of %u bytes failed:", n);
    eprintf("tot # malloc calls: %d", mc);
  }
  return p;
}

/* call calloc, terminate program if calloc error occurs */
void *ecalloc(size_t n, size_t s)
{
  void *p;

  ++mc; /* increment malloc calls counter */
  p = calloc(n, s);
  if (p == NULL) {
    printf("calloc of %u bytes failed:", n);
    eprintf("tot # malloc calls: %d", mc);
  }
  return p;
}

/* call realloc, terminate program if realloc error occurs */
void *erealloc(void *p, size_t s)
{
  ++mc; /* increment malloc calls counter */
  p = realloc(p, s);
  if (p == NULL) {
    printf("realloc of %u bytes failed:", s);
    eprintf("tot # malloc calls: %d", mc);
  }
  return p;
}

/* remove '\n' from end of string if is there... like perl's chomp */
char *chomp(char *in)
{
  int len = strlen(in);
  if (in[len - 1] == '\n') {
    in[len - 1] = '\0';
  }
```

```
  return (in);
}

/* count the number of bases (A,C,G,T not X or N) in a DNA string */
int countBases(char *in)
{
  int i; /* counter */
  int n = 0; /* number of bases */
  int len = strlen(in);
  for (i = 0; i < len; i++) {
    if ((in[i] == 'A') || (in[i] == 'C') || (in[i] == 'G')
|| (in[i] == 'T')) {
      ++n;
    }
  }
  return n;
}

/* raise base to the exp power */
int ipower(int base, int exp)
{
  int i, ret = 1;
  for (i = 0; i < exp; i++) {
    ret *= base;
  }
  return (ret);
}

/* reverse compliment a DNA string, assume out big enough to store in */
void revComp(char *in, int len)
{
  int j = 0, i;

  /* reverse the string */
  for (i = (len - 1); i > ((len - 1)) / 2; i--) {
    in[j] = in[i];
    j++;
  }

  /* compliment the string */
  for (i = 0; i < len; i++) {
    if (in[i] == 'A')
      in[i] = 'T';
    else if (in[i] == 'T')
      in[i] = 'A';
    else if (in[i] == 'C')
      in[i] = 'G';
    else if (in[i] == 'G')
      in[i] = 'C';
    else if (in[i] == 'a')
      in[i] = 't';
    else if (in[i] == 't')
      in[i] = 'a';
    else if (in[i] == 'c')
      in[i] = 'g';
    else if (in[i] == 'g')
      in[i] = 'c';
  }
}

/* print nicely the difference between two times */
char *printTime(time_t startTime, time_t stopTime)
{
  int elapsedTime;
  int elapsedDays;
```

```
  int elapsedHrs;
  int elapsedMins;
  int elapsedSecs;
  char *out;

  elapsedTime = (int) difftime(stopTime, startTime);
  if (elapsedTime >= 86400) {
    elapsedDays = elapsedTime / 86400;
    elapsedTime = elapsedTime % 86400;
  } else {
    elapsedDays = 0;
  }
  if (elapsedTime >= 3600) {
    elapsedHrs = elapsedTime / 3600;
    elapsedTime = elapsedTime % 3600;
  } else {
    elapsedHrs = 0;
  }
  if (elapsedTime >= 60) {
    elapsedMins = elapsedTime / 60;
    elapsedTime = elapsedTime % 60;
  } else {
    elapsedMins = 0;
  }
  elapsedSecs = elapsedTime;

  /* change this so 1000 not hard coded - opps */
  out = (char *) emalloc(1000 * sizeof(char));
  sprintf(out, "%d days, %d hours, %d mins, %d secs",
  elapsedDays, elapsedHrs, elapsedMins, elapsedSecs);

  return (out);
}
```

A.2.10 bl2seq.c

```
/**************************************************************************
                  bl2seq.c  -  runs bl2seq on two sequences
                  -----------------------------------------
    begin                : Sun Mar 18 2001
    author               : Kevin Pedretti
    email                : pedretti@eng.uiowa.edu
 **************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include "utils.h"
#include "bl2seq.h"

void bl2seq(char *seq1, int seq1Len, char *seq2, int seq2Len,
    bl2seq_hit * hits, int *nHits)
{
  FILE *SEQ1;
  FILE *SEQ2;
  FILE *OUT;
  int i;
  int n = 0;
  char *p;

  /* open temporary files to store sequences to bl2seq */
  SEQ1 = fopen("/tmp/seq1.fasta", "w+");
  SEQ2 = fopen("/tmp/seq2.fasta", "w+");
```

```
  if (SEQ1 == NULL || SEQ2 == NULL) {
    eprintf("can't open files for bl2seq\n");
  }

  /* bl2seq discards X's so we must be sure to change them */
  /* print seq1 to file, changing X's to N's */
  fprintf(SEQ1, ">seq1\n");
  for (i = 0; i < seq1Len; i++) {
    if (seq1[i] != 'X') {
      fputc(seq1[i], SEQ1);
    } else {
      fputc('N', SEQ1);
    }
  }
  fprintf(SEQ1, "\n");
  fclose(SEQ1);

  /* print seq2 to file, changing X's to N's */
  fprintf(SEQ2, ">seq2\n");
  for (i = 0; i < seq2Len; i++) {
    if (seq2[i] != 'X')
      fputc(seq2[i], SEQ2);
    else
      fputc('N', SEQ2);
  }
  fprintf(SEQ2, "\n");
  fclose(SEQ2);

  /* call bl2seq on the two sequences */
  call_bl2seq("/tmp/seq1.fasta", "/tmp/seq2.fasta", "/tmp/bl2seq.out");

  /* open the bl2seq output file */
  OUT = fopen("/tmp/bl2seq.out", "r");
  if (OUT == NULL) {
    eprintf("can't open bl2seq output file");
  }

  /* parse the hits from the file. */
  *nHits = parse_hits(OUT, hits);
  fclose(OUT);
}

int parse_hits(FILE * fd, bl2seq_hit * hits, int maxHits)
{
  char inLine[BL2SEQ_MAXLINE];
  int state = STATE_INIT;
  int curHit = 0;
  int in;
  int isEnd;
  int nStarts, nLens, nStrands;
  int sb1, sb2, lastsb1, lastsb2;
  int lastlen;
  int laststrand1, laststrand2;

  while (fgets(inLine, BL2SEQ_MAXLINE, fd) != NULL) {
    if (state == STATE_INIT) {
      if (strstr(inLine, "starts {") != NULL) {
state = STATE_STARTS;
nStarts = 0;
isEnd = 0;
// printf("START HIT %d\n", curHit);
// printf("STARTS = \n");
      }
      if (strstr(inLine, "lens {") != NULL) {
state = STATE_LENS;
```

```
nLens = 0;
isEnd = 0;
// printf("LENS = \n");
      }
      if (strstr(inLine, "strands {") != NULL) {
state = STATE_STRANDS;
nStrands = 0;
isEnd = 0;
// printf("STRANDS = \n");
      }
    } else {
      if (state != STATE_STRANDS) {
/* parse a number from the input line */
in = atoi(inLine);
      } else {
/* parse the direction from the input line */
if (strstr(inLine, "plus")) {
  in = PLUS;
} else {
  in = MINUS;
}
      }
      /* determine if this is the last entry at the current state */
      if (strstr(inLine, "}") != NULL) {
isEnd = 1;
      }

      if (state == STATE_STARTS) {
if (nStarts == 0) {
  sb1 = in;
} else if (nStarts == 1) {
  sb2 = in;
}

/* always store the last start pair */
if ((nStarts % 2) == 0) {
  lastsb1 = in;
} else {
  lastsb2 = in;
}
++nStarts;
      } else if (state == STATE_LENS) {
lastlen = in;
++nLens;
      } else if (state == STATE_STRANDS) {
if ((nStrands % 2) == 0) {
  laststrand1 = in;
} else {
  laststrand2 = in;
}
++nStrands;
      }

      if (isEnd == 1) {
/* if STATE_STRANDS, we're at the end of a hit */
if (state == STATE_STRANDS) {
  /* store the current hit */
  hits[curHit].sb1 = sb1;
  hits[curHit].eb1 = lastsb1 + lastlen - 1;
  hits[curHit].sb2 = sb2;
  hits[curHit].eb2 = lastsb2 + lastlen - 1;

  /* figure out the dir value to store */
  if ((laststrand1 == PLUS) && (laststrand2 == PLUS)) {
    hits[curHit].dir = PP;
```

```
  } else if ((laststrand1 == PLUS) && (laststrand2 == MINUS)) {
    hits[curHit].dir = PM;
  } else if ((laststrand1 == MINUS) && (laststrand2 == PLUS)) {
    hits[curHit].dir = MP;
  } else if ((laststrand1 == MINUS) && (laststrand2 == MINUS)) {
    hits[curHit].dir = MM;
  }
  ++curHit;
}
state = STATE_INIT;
      }
    }
  }
  return curHit;
}

/* environ defined in unistd.h */
extern char **environ;

/* simple wrapper function to call bl2seq.
 * seq1 and seq2 are the two input file names.
 * out is the output filename to use.
 */
int call_bl2seq(char *seq1, char *seq2, char *out)
{
  int pid, status;

  if (seq1 == NULL || seq2 == NULL) {
    return 1;
  }
  pid = fork();
  if (pid == -1) {
    return -1;
  }

  /* setup the arguments to pass to bl2seq */
  if (pid == 0) {
    char *argv[13];
    argv[0] = "bl2seq";
    argv[1] = "-i";
    argv[2] = seq1;
    argv[3] = "-j";
    argv[4] = seq2;
    argv[5] = "-p";
    argv[6] = "blastn";
    argv[7] = "-o";
    argv[8] = "/dev/null";
    argv[9] = "-a";
    argv[10] = out;
    argv[11] = "-FF";
    argv[12] = 0;
    execve("/mnt/r0-blastdb/blast-bin/bl2seq", argv, environ);
    exit(127);
  }
  /* try until for child, retrying if we're interrupted */
  do {
    if (waitpid(pid, &status, 0) == -1) {
      if (errno != EINTR) {
return -1;
      }
    } else {
      return status;
    }
  } while (1);
}
```

REFERENCES

[1] M.D. Adams, J.M. Kelley, J.D. Gocayne, M. Dubnick, M.H. Polymeropoulos, H. Xiao, C.R. Merril, A. Wu, B. Olde, R.F. Moreno, et al., *Complementary DNA sequencing: Expressed sequence tags and the human genome projectr,.* Science 252 (1991) 1651-1656.

[2] M.D. Adams, A.R. Kerlavage, R.D. Fleishmann, R.A. Fuldner, C.J. Bult, N.H. Lee, E.F. Kirkness, K.G. Weinstock, J.D. Gocayne, O. White et al., *Initial assessment of human gene diversity and expression patterns based upon 83 million nucleotides of cDNA sequence,* Nature 377 (1995) 3-17.

[3] S.F. Altshul, W. Gish, W. Miller, E.W. Meyers, D.J. Lipman, *Basic local alignment search tool,* Journal of Molecular Biology 215 (1990) 403-410.

[4] D.J. Becker, T. Sterling, D. Savarese, J.E. Dorband, U.A. Ranawak, C.V. Packer, *Beowulf: A Parallel Workstation for Scientific Computation,* International Conference on Parallel Processing Proceedings (1995).

[5] M.F. Bonaldo, G. Lennon, M.B. Soares, *Normalization and subtraction: two approaches to facilitate gene discovery,* Genome Research 6 (1996) 791-806.

[6] T.A. Braun, *Personal Communication.*

[7] C. Burge, S. Karlin, *Prediction of complete gene structures in human genomic DNA,* Journal of Molecular Biology 268 (1997) 78-94.

[8] J. Burke, D. Davison, W. Hide, *d2_cluster: A Validated Method for Clustering EST and Full-Length cDNA Sequences,* Genome Research 9 (1999) 1135-1142.

[9] J.P. Burke, H. Wang, W. Hide, D. Davison, *Alternative gene form discovery and candidate gene selection from gene indexing projects,* Genome Research 8 (1998) 276-290.

[10] T.L. Casavant, *Coordinated Laboratory for Computational Genomics http://genome.uiowa.edu.*

[11] T.L. Casavant, *Manuscript in preparation.*

[12] FastA format description, *http://www.ncbi.nlm.nih.gov/BLAST/fasta.html.*

[13] P. Green, *Phrap,* unpublished data, *http://www.phrap.org.*

[14] X. Guan, R.J. Mural, R.R. Einstein, R.C. Mann, E.C. Uberbacher, *GRAIL: An Integrated Artificial Intelligence System for Gene Recognition and Interpretation,* Proceedings of the Eighth IEEE Converence on AI Applications (1992) 9-13.

[15] D. Gusfield, *Algorithms on Strings, Trees, and Sequences,* Cambridge University Press, New York, 1997.

[16] C. Horstmann and G. Cornell *Core Java 2, Volume 1: Fundamentals* Prentice Hall PTR, United States, 2000.

[17] International Human Genome Sequencing Consortium, *Initial sequencing and analysis of the human genome,* Nature 409 (2001) 860-921.

[18] B. Kernighan and D. Ritchie, *The C Programming Language, 2nd Edition,* Prentice Hall, Upper Saddle River, New Jersey, 1989.

[19] Message Passing Interface Form, *MPI: A message-passing interface standard,* University of Tennessee Technical Report CS-94-230 (1994).

[20] R.T. Miller, A.G. Christoffels, C. Gopalakrishnan, J.A. Burke, A.A. Ptitsyn, T.R. Broveak, W.A. Hide, *A comprehensive approach to clustering of expressed human gene sequence: The Sequence Tag Alignment and Consensus Knowledge-base* Genome Research 9 (1999) 1143-1155.

[21] J.D. Parsons, S. Brenner, M.J. Bishop, *Clustering cDNA Sequences* Computational Applications in Bioscience 8 (1992) 461-466.

[22] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, *Numerical Recipes in C, 2nd Edition,* Cambridge University Press, New York, 1992.

[23] S.L. Salzberg, D.B. Searls, S. Kasif, *Computational Methods in Molecular Biology,* Elsevier Science B.V., Amsterdam, 1998.

[24] F. Sanger, S. Nicklen, A.R. Coulson, *DNA sequencing with chain-terminating*

inhibitors, Proceedings of the National Academey of Science of the United States of America 74 (1997) 5463-5467.

[25] T.E. Scheetz, *Development of genomic resources for the rat: an EST map and analysis of alternative polyadenylation,* PhD Thesis, University of Iowa, May 2001.

[26] T.E. Scheetz, C.L. Birkett, C.A. Roberts, J.M. Gardiner, D.Y. Nishimura, V.C. Sheffield, M.B. Soares, T.L. Casavant, *Efficient, Exact Clustering Analysis of ESTs to Support Serial Subtraction of Pooled cDNA Libraries,* Proceedings of the 1999 Human Genome and Sequencing Meeting, Cold Spring Harbor Laboratory, Long Island, NY.

[27] G.D. Schuler, *Pieces of the puzzle: expressed sequence tags and the catalog of human genes.* Journal of Molecular Medicine 75 (1997) 694-698.

[28] A.F. Smit and P. Green, *RepeatMasker,* unpublished data, *http://repeatmasker.genome.washington.edu.*

[29] T.F. Smith and M.S. Waterman, *Identification of common molecular subsequences,* Journal of Molecular Biology 147 (1981) 195-197.

[30] W.R. Stevens, *Advanced Programming in the UNIX Environment,* Addison Wesley, United States, 1997.

[31] T.A. Tatusova and T.L. Madden, *Blast 2 sequences – a new tool for comparing protein and nucleotide sequences",* FEMS Microbiology Letter 174 (1999) 247-250.

[32] K. Thompson, *UNIX Implementation,* Bell Systems Technical Journal 57 (1978) 1931-1946.

[33] United States Congress, Office of Technology Assessment, *Mapping Our Genes– The Genome Projects: How Big? How Fast?,* OTA-BA-373, U.S. Government Printing Office, Washington, DC, 1998.

[34] J.C. Venter, *Identification of new human receptor and transporter genes by high throughput cDNA (EST) sequencing,* Journal of Pharmacy and Pharmacology 45 (1993) 355-360.

[35] L. Wall, T. Christiansen, R.L. Schwartz, *Programming Perl, 2nd Edition,* O'Reilly & Associates, United States, 1996.