INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Query Output Merge Tool for DNA Sequences

— BlastMerge

Xiaoming Wang

A Major Report
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

September 2001

Abstract

Query Output Merge Tool for DNA Sequence
—— BlastMerge

Xiaoming Wang

The standalone BLAST program is a popular tool to search a DNA sequence against the NCBI database. Since NCBI updates its database every day, the user has to download the latest database before he can run his query. Since the NCBI database is huge (by now more than 100M), the download of the whole database is a time-consuming work. With the continue increase of the NCBI database, this issue will become more significant.

The best solution is to re-download the whole database periodically. For each day before the user runs the BLAST program, he has to download the update database, such as month.aa. This update database is smaller than the whole database. He then searches the query on both the whole local database and the update database. The merge of the two query results should be equivalent to the query results on the latest whole database. The BlastMerge program is designed for this purpose.

BlastMerge program is the tool that can merge the query results of blastn or blastp program. Originally the idea is from Dr. Clement Lam and Dr. Gregory Butler. The key point of the algorithm used in BlastMerge is to use some data structure to store query results.

In this major report we designed the BlastMerge tool with the Object Oriented Methodology, developed BlastMerge tool with C++ language on a UNIX platform, and verified BlastMerge tool by merging the query output of yeast.nt and its update.

Acknowledgments

The fulfillment of this major report has been done under the Dr. Clement Lam's supervision. I would like to thank Dr. Clement Lam for his deliberate and careful supervision.

Contents

List of Figures

Chapter 1 Introduction

1.1 NCBI database

DNA (deoxyribonucleic acid) is the genetic material of nearly all-living organisms. It controls heredity and is located in the cell nucleus. The genetic information of the DNA is contained in the sequence of bases along the molecule. This sequence is called the DNA sequence. The research of the genetic information of DNA sequence is very important. Understanding nature's mute but elegant language of living cells is the quest of modern molecular biology. It will be an aid in gene discovery, in the design of molecular modeling, in the planning of site-directed mutagenesis, and in experiments that can potentially reveal previously unknown relationships with respect to the structure and function of genes and proteins.

With the explosion of sequence and structural information available to researchers, the field of bioinformatics, or more properly, computational biology, is playing an increasingly larger role in the study of fundamental biomedical problems. It is necessary to develop gene database and analyze it.

The National Center for Biotechnology Information (NCBI) built the NCBI database(GenBank). It is very convenient for researchers to query and to compare DNA information in the database. GenBank is the genetic sequence database, an annotated collection of all publicly available nucleotide and protein sequences. The records represent single, contiguous stretches of DNA and RNA with annotations. The files are grouped in divisions: some are phylogenetically derived, while others are based on the technical approach that was used to generate the sequence information. Presently all records in GenBank are generated from direct submissions to the DNA sequence databases from the original authors, who volunteer their records as part of the publication process or to make the data publicly available. Genbank, which is built by the National

Center for Biotechnology Information at NIH in Bethesda, Maryland, is part of the International Nucleotide database Collaboration, along with its two partners, the DNA Database of Japan (DDBJ, Mishima, Japan) and the European Molecular Biology Laboratory (EMBL) nucleotide database from the European Mole Bioinformatics Institute (EBI, Hinxton, England). All three centers are separate points of data submission, but all are exchanging their information daily, and are making the same database available to the community at large. NCBI and its partners maintain the following databases.

1.1.1 Protein Database

1.1.1.1 PDB: Protein Data Bank at Brookhaven National Laboratories

Overview

Protein Data Bank contains the core public collection of three-dimensional structures of proteins as well as holding 3-D structures of nuleic acid, carbohydrates, and a variety of complexes. They are experimentally determined by X-ray crystallographers and NMR spectroscopists.

PDB-ID Codes

The structure record accessioning scheme of the Protein Data Bank is a unique four-character alphanumeric code called a PDB-ID or PDB code. This scheme uses the digits 0 to 9, and the uppercase letters A to Z. Thus there are over 1.3 million possible combinations. PDB-IDs are not assigned in any particular order. Rather, indexers at Protein Data Bank try to devise mnemonics that makes the structures easier to remember, such as 3INS.

Sequences from PDB Structure Records

PDB-file-encoded sequences are notorious. Since completeness of a structure is not always guaranteed, PDB file contains two copies of the sequence information, an explicit

sequence and an implicit sequence. Both are required to construct the chemical graph of a biopolymer.

1.1.1.2 SWISS-PROT

SWISS-PROT is an annotated protein sequence database established in 1986 and maintained collaboratively, since 1987, by the Department of Medical Biochemistry of the University of Geneva and the EMBL Data Library (now the EMBL Outstation - The European Bioinformatics Institute). The SWISS-PROT protein sequence data bank consists of sequence entries.

The SWISS-PROT database distinguishes itself from other protein sequence databases by three distinct criteria:

Annotation

Minimal redundancy

Integration with other databases

1.1.1.3 PIR

The PIR (Protein Information Resource) database was initiated at the NBRF in the early 1960's by the late Margaret O. Dayhoff as a collection of sequences for the study of evolutionary relationships among proteins. The database is now an international collaboration of three data centers: the NBRF, the Munich Information Center for Protein Sequences (MIPS), and the Japan International Protein Information Database (JIPID). The three centers cooperate to produce and distribute a single database of 'wild-type' protein sequences.

1.1.1.4 PRF

The PRF (Protein Research Fundation) of Japan database contains protein sequences abstracted from scientific publications.

1.1.1.5 BLAST protein database

nr.Z

All non-redundant GenBank CDS translations+PDB+SwissProt+PIR+PRF .

Month.aa.Z

All new or revised GenBank CDS translation+PDB+SwissProt+PIR released in the last 30 days.

Swissprot.Z

The last major release of the SWISS-PROT protein sequence database (no updates). These are uploaded to the system when they are received from EMBL.

yeast.aa.Z

Yeast (Saccharomyces cerevisiae) protein sequences. This database is not to be confused with a listing of all Yeast protein sequences. It is a database of the protein translations of the Yeast complete genome.

ecoli.aa.Z

E. coli (Escherichia coli) genomic CDS translations.

pdbaa.Z

Sequences derived from the 3-dimensional structure in the Brookhaven Protein Data Bank.

pataa.Z

Protein sequences derived from the Patent division of GenBank.

kabat [kabatpro]

Kabat's database of sequences of immunological interest.

alu

Translations of select Alu repeats from REPBASE, suitable for masking Alu repeats from query sequences.

1.1.2 Nucleotides Database

1.1.2.1 MMDB

NCBI's Molecular Modeling Database (MMDB: Hogue et al., 1990), is part of NCBI's Entrez (Schuler et al., 1997) 3-D structures of biomolecules from crystallographic and NMR studies. MMDB is a database of ASN.1-formatted records (Rose, 1990), not PDB-formatted records. Structures in MMDB have value-added information compared to the

original PDB structures. These include the addition of the explicit chemical graph information following an extensive suite of validation procedures, the addition of uniformly derived secondary structure definitions, citation matching to MEDLINE, and the molecule based assignment of taxonomy to each biologically derived protein or nucleic acid chain.

1.1.2.2 EMBL

The EMBL Nucleotide Sequence Database is a comprehensive database of DNA and RNA sequences collected from the scientific literature and patent applications and directly submitted from researchers and sequencing groups. Data collection is done in collaboration with GenBank (USA) and the DNA Databank of Japan (DDBJ).

1.1.2.3 DDBJ

Entries from the DNA Databank of Japan (DDBJ) are wholly incorporated into GenBank.

1.1.2.4 dbEST

dbEST is the division of GenBank that contains "single-pass" cDNA sequences, or Expressed Sequence Tags, from a number of organisms.

1.1.2.5 BLAST Nucleotides database

nt.Z

All Non-redundant GenBank+EMBL+DDBJ+PDB sequences (but no EST, STS, GSS, or HTGS sequences).

month.na.Z

All new or revised GenBank+EMBL+DDBJ+PDB sequences released in the last 30. days.

est.Z

Non-redundant Database of GenBank+EMBL+DDBJ EST Divisions.

est_human.Z

Non-redundant Database of Human GenBank+EMBL+DDBJ EST sequences.

est_mouse.Z

Non-redundant Database of Mouse GenBank+EMBL+DDBJ EST sequences.

est_others.Z

Non-redundant Database of all other organisms GenBank+EMBL+DDBJ EST sequences.

sts.Z

Non-redundant Database of GenBank+EMBL+DDBJ STS Divisions.

htg.Z

High Throughput Genomic Sequences.

yeast.nt.Z

Yeast (Saccharomyces cerevisiae) genomic nucleotide sequences.

ecoli.nt.Z

E. coli genomic nucleotide sequences.

pdbnt.Z

Sequences derived from the 3-dimensional structure Brookhaven Data Bank

vector.Z

Vector subset of GenBank, NCBI.

mito.Z

Database of mitochondrial sequences (Rel. 1.0, July 1995).

gss.Z

Genome Survey Sequence, includes single-pass genomic data, exon-trapped sequences, and Alu PCR sequences.

patnt.Z

Nucleotide sequences derived from the Patent division of GenBank.

1.1.3 FASTA format description

When you search the alignments on a DNA sequence database, you need to specify your DNA sequence. For the BLAST search tool, the common format of DNA sequence is the FASTA sequence format.

A sequence in FASTA format begins with a single-line description followed by lines of sequence data. The description line is distinguished from the sequence data by a greater-

than (">") symbol in the first column. It is recommended that all lines of text be shorter than 80 characters in length. An example sequence in FASTA format is:

>gi|532319|pir|TVFV2E|TVFV2E envelope protein
ELRLRYCAPAGFALLKCNDADYDGFKTNCSNVSVVHCTNLMNTTVTTGLLLNG
SYSENRTQIWQKHRTSNDSALILLNKHYNLTVTCKRPGNKTVLPVTIMAGLVFHS
QKYNLRLRQAWCHFPSNWKGAWKEVKEEIVNLPKERYRGTNDPKRIFFQRQWG
DPETANLWFNCHGEFFYCKMDWFLNYLNNLTVDADHNECKNTSGTKSGNKRA
PGPCVQRTYVACHIRSVIIWLETISKKTYAPPREGHLECTSTVTGMTVELNYIPKN
RTNVTLSPQIESIWAAELDRYKLVEITPIGFAPTEVRRYTGGHERQKRVPFVXXXX
XXXXXXXXXXXXXXXXXXVQSQHLLAGILQQQKNLLAAVEAQQQMLKLTIWG
VK

Sequences are expected to be represented in the standard IUB/IUPAC amino acid and nucleic acid codes, with these exceptions: lower-case letters are accepted and are mapped into upper-case; a single hyphen or dash can be used to represent a gap of indeterminate length; and in amino acid sequences, U and * are acceptable letters (see below).

Before submitting a request, any numerical digits in the query sequence should either be removed or replaced by appropriate letter codes. (e.g., N for unknown nucleic acid residue or X for unknown amino acid residue).

The nucleic acid codes supported are:

A --> adenosine	M --> A C (amino)
C --> cytidine	S --> G C (strong)
G --> guanine	W --> A T (weak)
T --> thymidine	B --> G T C
U --> uridine	D --> G A T
R --> G A (purine)	H --> A C T
Y --> T C (pyrimidine)	V --> G C A
K --> G T (keto)	N --> A G C T (any)
- gap of indeterminate length	

For those programs that use amino acid query sequences (BLASTP and TBLASTN), the accepted amino acid codes are:

A	alanine	P	proline

B	aspartate or asparagine	Q	glutamine
C	cystine	R	arginine
D	aspartate	S	serine
E	glutamate	T	threonine
F	phenylalanine	U	selenocysteine
G	glycine	V	valine
H	histidine	W	tryptophan
I	isoleucine	Y	tyrosine
K	lysine	Z	glutamate or glutamine
L	leucine	X	any
M	methionine	*	translation stop
N	asparagine	-	gap of indeterminate length

1.2 Database search tool—BLAST

1.2.1 Sequence Alignment and database searching

Biology has a long tradition of comparative analysis leading to discovery. For instance, Darwin's comparison of morphological features of the Galapagos finches and other species ultimately led him to postulate the theory of natural selection. In essence, we are performing the same type of analysis today, but in much greater detail, when we compare the sequence of genes and proteins. In this activity, we analyze the similarities and differences--at the level of individual bases or amino acids—with the aim of inferring structural, functional, and evolutionary relationships among the sequence under study. The most common comparative method is sequence alignment, which provides an explicit mapping between the residues of two or more sequences. For simplification we discuss only pairwise alignments, which involves comparing two sequences.

1.2.1.1 The evolutionary basis of sequence alignment

One goal of sequence alignment is to enable the researcher to determine whether two sequences display sufficient similarity to justify the inference of homology. Although these two terms are often interchanged in popular usage, let us distinguish them to avoid

confusion. Similarity is an observable quantity that might be expressed as, say, percent of identity or some other suitable measure. Homology on the other hand, refers to a conclusion drawn from these data that two genes share a common evolutionary history. Genes either are or are not homologous—there is no degrees of homology as there are of similarity.

Bearing in mind the goal of inferring evolutionary relationships, it is fitting that most alignment methods try, at least to some extent, to model the molecular mechanisms by which sequences evolve. While it is presumed that homologous sequences have diverged from common ancestral sequence through iterative molecular changes, we do not actually know what the ancestral sequence was (barring the possibility that DNA could be recovered from a fossil); all we have to observe are the sequence from extant organisms. The changes that occur during divergence from the common ancestor can be categorized as substitutions, insertions, and deletions. In the ideal case that a sequence alignment genuinely reflects the evolutionary history of two genes or protein, residues that have been aligned, yet are not identical, would represent substitutions. Regions in which the residues of one sequence correspond to nothing in the other would be interpreted as either an insertion into one sequence or a deletion from the other.

In a residue-by-residue alignment, it is often apparent that certain regions of a protein, or perhaps specific amino acids, are more highly conserved than others. This information may be suggestive of which residues are most crucial for maintaining a protein's structure or function. On the other hand, there may be other positions that do not play a significant functional role yet happen to be identical for historical reasons. There is reason for particular caution when the sequences are taken from very closely related species, since similarities may be more reflective of history than of function. Nevertheless, sequence alignments provide a useful way to gain new insights by leveraging existing knowledge, for example, by deducing structural and functional properties of a novel protein from comparison to those that have been well studied. It must be emphasized, however, that these inferences should not be assumed to be correct based on computational analysis alone: they must always be tested experimentally.

Upon observing a surprisingly high degree of sequence similarity between two genes or proteins, we might infer that they share a common evolutionary history, and from this we might anticipate that they would have similar biological functions. But again, this prediction should be treated as hypothetical until tested experimentally.

The earliest sequence alignment methods were applicable to a simple type of relationship in which the sequence shows easily detectable similarity along their entire lengths. An alignment that spans essentially the full extents of the input sequences is called a global alignment. Protein consisting of a single globular domain can often be aligned by means of a global strategy, as can any homologous sequences that have not diverged substantially.

1.2.1.2 The Modular Nature of Proteins

Many proteins do not display global patterns of similarity but instead appear to be mosaics of modular domains. Patterns of modularity often arise by in-frame exchange of whole exons. Global alignment methods do not take this phenomenon into account, which is understandable, considering that they were developed before the exon/intron structure of gens had been discovered. In most cases, it is advisable to instead use a sequence comparison method that can produce a local alignment. Such an alignment consists of paired subsequences, which may be surrounded by residues that are completely unrelated. Consequently, users should bear in mind that some local similarities might be missed if a global alignment strategy is applied inappropriately. Another obvious case in which local alignments are desired is the alignment of the nucleotide sequence of spliced mRNA to its genomic sequence, where each exon would be in a distinct local alignment.

Dot matrix representations have enjoyed widespread popularity, in part because of their ability to reveal complex relationship involving multiple region of local similarity. The basic idea is to use the sequences as the coordinates of a two-dimensional graph and then plot points of correspondence within its interior. Each dot usually indicates that within

some small window, the sequence similarity is above some cutoff. When two sequences are consistently matching over an extended region, the dots will merge to form a diagonal line segment.

In a dot matrix representation, certain patterns of dots may appear to sketch out a "path", but it is up to the viewer to deduce the alignment from this information. Another graphical representation known as a path graph provides an explicit representation of an alignment.

To understand a path graph, and image a two-dimensional lattice in which the vertices represent points between the sequence residues. An edge that connects two vertices along a diagonal corresponds to the pairing of one residue from each sequence. Horizontal and vertical edges pair a residue from one sequence with nothing in the other. In other words, these edges constitute a gap in the alignment. The entire graph corresponds to the search space that must be examined for potential alignments. Each possible path through this space corresponds to exactly one alignment.

1.2.1.3 Optimal Alignment Methods

For any but the most trivial problems, the total number of distinct alignments is extraordinarily large, so it is usually of interest to identify the "best" one among them (or the several best ones). This is where the concept of representing an alignment as a path pays off. Many problems in computer science can be reduced to the task of finding the optimal path through a graph; efficient algorithms have been developed for this purpose. One requirement is a means of assigning a quality score to each possible path (alignment). Normally this is accomplished by summing the incremental contributions of each step along its route. For now let us assume that some positive incremental scores will be used for aligning identical residues, with negative scores used for substitutions and gaps. According to this definition of alignment quality, finding the path whose total score is maximal will give us the best sequence alignment.

What is today known as the Needleman-Wunsch algorithm is an application of a best path strategy called dynamic programming to the problem of finding optimal sequence

alignments (Needleman and Wunsch, 1970). The idea behind dynamic programming comes from the observation that any partial subpaths that ends at a point along the true optimal sequence alignment must itself be the optimal path leading up to that point. Thus, the optimal path can be found by an incremental extension of the optimal path. In the basic Needleman-Wunsch formulation, the optimal alignment must extend from beginning to end in both sequences, that is, from the top-left corner in the search space to the bottom right. In other words, it seeks global alignment.

But a simple modification to the basic strategy allows the optimal local alignment to be found (Smith and Waternman, 1981). The path for this alignment need not reach the edges of the search graph, but may begin and end internally. Such an alignment would be locally optimal if its score could not be improved by either increasing or decreasing the extent of the alignment. This procedure relies on a property of the scoring system, namely, that the cumulative score for a path will decrease in regions of poorly matching sequence (the scoring systems described below satisfy this criterion). When the score drops to zero, extension of path is terminal and new one can begin. There can be many individual paths bounded by regions of poorly matching sequence instead of by the ends of the sequences, as is the case for global alignments. Of other paths, the single one with the highest score is reported as the optimal local alignment.

1.2.1.4 Substitution Scores and Gap Penalties

The score system just described made use of a simple match/mismatch scheme, but in comparisons of proteins, we can increase sensitivity to weak alignments through the use of a substitution matrix. It is well known that certain amino acids can substitute easily for one another in related proteins, presumably owing to their similar physicochemical properties. When calculating alignment scores, identical amino acids should be given higher values than substitutions, but conservative substitutions should also be given greater values than non-conservative changes. In other words, a range value is desired. Furthermore, different sets of values may be desired for comparing very similar sequence as opposed to highly divergent sequences. These considerations can be dealt with in a

flexible manner through the use of a substitution matrix, in which the score for any pair of amino acids can be obtained, with a simple lookup.

The first substitution matrices to gain widespread usage were those based on point-accepted-mutation (PAM) model of evolution(Dayhoff et al., 1978). One PAM is a unit of evolutionary divergence in which 1% of the amino acids have been changed. This does not imply that after 100PAMs every amino acids will be different; some positions may change several times, perhaps even reverting to the original amino acid, while others may not change at all. If these changes were purely random, the frequencies of each possible substitution would be determined simply by the overall frequencies of the different amino acids (called the background frequencies). However, in related proteins, the observed substitution frequencies (called the target frequencies) are biased toward those that do not seriously disrupt the protein's function. In other words, these are point mutations that have been "accepted" during evaluation.

The BLOSUM substitution matrices have been constructed in a similar fashion, but making use of a different strategy for estimating the target frequencies. The underlying set of data was derived from the BLOCKS database, which contains local multiple alignments ("block") involving distantly related sequences. As with the PAM model, there is a numbered series of BLOSUM matrices, but the number in this case refers to the maximum level of identity, which sequences may have and still contribute independently to model.

To compensate for insertions and deletions, it is desirable to allow some gaps to be introduced into an alignment but not so many that the alignment asserts an implausible series of alterations. This is accomplished by deducting some amount from the alignment score for each gap introduced. Although a number of strategies have been proposed for penalizing gaps, the most common formulation involves a fixed deduction for introducing a gap plus an additional deduction proportional to the length of the gap. This is governed by two parameters: G (sometimes called the gap opening penalty) and L (the gap extension penalty).

1.2.1.5 Statistical Significance of Alignments

For any given alignment we can calculate a score, but it is important to determine whether this score is high enough to provide evidence of homology. In addressing this matter, it is helpful to have some notion of the highest score that can be expected due to chance alone. Unfortunately, there is no mathematical theory to describe the expected distribution of scores for global alignments. One of the few methods available for assessing their significance is comparison of the observed alignment score to those of many alignment made from random sequences of same length and comparison as those under study (Fitch, 1983).

However, for local alignments the situation is much better. As an initial simplification of the problem, attention has been focused on local alignments that do not contain any gaps (Karlin and Altschul, 1990). This type of alignment is called a high-scoring segment pair (HSP). HSP may be found using a modification of the Smith-Waterman algorithm or by simply imposing huge penalties for gaps. Karlin-Altschul statistics provides a mathematical theory to describe the expected distribution of random HSP scores. The form of the probability density function is known as the extreme value distribution. This is worth noting because application of the more familiar normal distribution can result in greatly exaggerated claims of significance. By relating an observed alignment score S to the expected distribution, it is possible to calculate a p value, which gives the probability that an alignment with that score or better could be due simply to chance. Highly significant scores would therefore be those with p values very close to zero.

A related quantity is E, the expected number of chance alignments with scores at least equal to S. The extreme value distribution is characterized by two parameters, K and λ, which can be calculated analytically and are constant for any particular scoring system and set of background frequencies. The significance of an alignment also depends on the size of the search space that was examined (just as one's chances of finding a needle in a haystack depend on the size of the haystack). The size of search space has typically been calculated as the product of the sequence lengths; but for correct statistics, these lengths must be reduced by the expected length of a local alignment to avoid an "edge effect"

(Altschul and Gish, 1996). This reduction is required because an alignment that begins near the edge of the search space will run out of sequence before it can achieve a significant score.

The restriction that alignments cannot contain gap is a useful simplification but represents a departure from biological reality, since in fact gaps are needed to accurately model insertions and deletions. However, provided the gaps are relatively few in number, it may still be possible to find high-scoring ungapped regions between them. Typically, several HSPs occurring in close proximity will be observed. In this case it is desirable to evaluate their significance as an ensemble rather than individually. Perhaps, for example, no segment is significant in its own right, but the appearance of several together is very unlikely to have occurred by chance. Karlin-Altschul sum statistics allows the significance of the N best HSPs to be calculated (Karlin and Altschul, 1993). The essence of this method is to sum the score of the N best segments and then attempt to determine the probability that this value could be due to chance. Typically, some additional heuristics are applied to ensure that scores are summed only if the segments are compatible with an alignment (i.e., ascending coordinates in both sequences with little or no overlap). Although the distribution of summed scores will differ from that of maximal HSP scores, it may be calculated analytically.

Finally, it is desirable to have the ability to evaluate the significance of local alignments in which gaps are explicitly modeled. Such as the traditional Smith-Waterman alignments.

Although no formal proof exists, it is believed that scores for these alignments will also follow the extreme value distribution (Smith et al., 1985; Waterman and Vingron, 1994;Altschul and Gish, 1996). However, the value of K and λ needed to parameterize this distribution cannot be directly calculated. Instead, methods for estimating these values through simulations have been developed (Waterman and Vingron, 1994; Altschul and Gish, 1996).

1.2.2 Database Similarity Searching

The discussion above focuses on the alignment of specific pairs of sequences, but for a newly determined sequence we generally have no way of knowing the appropriate sequences to use in such a comparison. Database similarity searching allows us to determine which of the hundreds of thousands of sequences present in the database are potentially related to a particular sequence of interest. This process sometimes leads to unexpected discoveries. The first "eureka moment" with this strategy came when the viral oncogene v-sis was found to be a modified form of cellular gene that encodes platelet-derived growth factor (Doolittle et al., 1983). At the time of this discovery, sequences database were small enough that such a finding might have been considered surprising. But today it would be much more surprising to perform a database search and not get a hit. The genome of the yeast Saccharomyces cerevisae has been completely sequenced, as have several smaller genomes. Among the vertebrates, large numbers of partial sequences representing novel human and mouse genes have been deposited in GenBank as the result of a number of expressed sequence tag (EST) projects. The chief utility of observing EST matches in a database search is that the cDNA clones from which they were derived are freely available and can provide the critical reagents for isolating genes of interest, including their homologs in other model organisms.

In database searching, the basic operation is to sequentially align a query sequence to each subject sequence in the database. The results are reported as ranked hit list followed by a series of individual sequence alignments, plus various scores and statistics. Choices of search program sequence database, and various optional parameters can have impact on the effectiveness of a search. Furthermore, there are various interfaces to these facilities such as console-style commands, and WWW forms.

Current sequence databases are already immense and have continued to increase at an exponential rate, making straightforward application of dynamic programming methods impractical for database searching. One solution is to use massively parallel computers and other specialized hardware. When optimal methods are impractical, it is necessary to

resort to heuristic methods, which make use of approximations to significantly speed up sequence comparisons, but with a small risk of missing true alignments.

One heuristic method is based on the strategy of breaking a sequence up into short runs of consecutive letters called words. Words-based methods, introduced in the early 1980s by Wilbur and Lipman, are used by virtually all of today's popular search programs. The basic idea is that an alignment representing a true sequence relationship will contain at least one word that is common to both sequences. These word hits can be identified extremely rapidly by pre-indexing all words from the query and then consulting the index as the database is scanned.

1.2.2.1 FASTA

The FASTA program was the first widely used program for database similarity searching (LipMan and Pearsin, 1985; Pearson and Lipman, 1988). To achieve a high degree of sensitivity, this program performs optimized searches for local alignments using a substitution matrix. As noted, however, it would take a substantial amount of time to apply this strategy exhaustively. To improve speed, the program uses the observed pattern of word hits to identify potential matches before the more time-consuming optimized search is attempted. The trade-off between speed and sensitivity is controlled by the ktup parameter, which specifies the size of a word. Increasing the value of ktup decreases the number of background word hits (i.e.. those that do not mark the position of an optimal alignment). This in turn decreases the amount of optimized searching required and improves overall search speed. The default *ktup* value for comparing protein is 2, but for finding very distant relationships, reduction to 1 is recommended.

The FASTA program does not investigate every word hit encountered, but instead looks initially for segments containing several nearby hits. Using a heuristic method, these segments are assigned scores, and the score of the best segment found appears in the output as the initl score. Several segments may then be combined and a new initn score is calculated from the ensemble. Most potential matches are then further evaluated by performing a search for gapped local alignment that is constrained to a diagonal band

centrered around the best initial segment. The score of this optimized alignment is shown in the output as the opt score. For alignments finally reported (a user-specified number from the top of hit list), a full Smith-Waterman alignment search (i.e., without the constraining band) is performed. Only the single optimal alignment is produced for each database sequence. Since, however, meaningful alignments can be missed by this approach if the proteins contain multiple modules, matching sequences should be further analyzed with the LLIGN program.

Beginning with version 2.0, FASTA provides an estimate of the statistical significance of each alignment found. The program assumes an extreme value distribution for random scores, but uses a rewritten form of the probability density function in which the expected score is a linear function of the natural log of the length of the database sequence. Simple linear regression can then be used to calculate a normalized z score for each alignment. Finally, an expectation E is calculated, which gives the expected number of random alignment with z scores greater than or equal to the value observed.

1.2.2.2 BLAST

1.2.2.2.1 The statistics of global sequence comparison

To assess whether a given alignment constitutes evidence for homology, it helps to know how strong an alignment can be expected from chance alone. In this context, "chance" can mean the comparison of (i) real but non-homologous sequences; (ii) real sequences that are shuffled to preserve compositional properties; or (iii) sequences that are generated randomly based upon a DNA or protein sequence model.
Analytic statistical results invariably use the last of these definitions of chance, while empirical results, based on simulation and curve-fitting may use any of the definitions.

Unfortunately, under even the simplest random models and scoring systems, very little is known about the random distribution of optimal global alignment scores. Monte Carlo experiments can provide rough distributional results for some specific scoring systems and sequence compositions, but these can not be generalized easily. Therefore, one of the

few methods available for assessing the statistical significance of a particular global alignment is to generate many random sequence pairs of the appropriate length and composition, and calculate the optimal alignment score for each. While it is then possible to express the score of interest in terms of standard deviations from the mean, it is a mistake to assume that the relevant distribution is normal and convert this *Z*-value into a *P*-value; the tail behavior of global alignment scores is unknown. The most one can say reliably is that if 100 random alignments have score inferior to the alignment of interest, the *P*-value in question is likely less than 0.01. One further pitfall to avoid is exaggerating the significance of a result found among multiple tests. When many alignments have been generated, e.g. in a database search, the significance of the best must be discounted accordingly. If it was selected as the best among 1000 independent trials an alignment with *P*-value 0.0001 in the context of a single trial may be assigned a *P*-value of only 0.1

1.2.2.2.2 The statistics of local sequence comparison

Fortunately statistics for the scores of local alignments, unlike those of global alignments, are well understood. This is particularly true for local alignments lacking gaps, which we will consider first. Such alignments were precisely those sought by the original BLAST database search programs.

A local alignment without gaps consists simply of a pair of equal length segments, one from each of the two sequences being compared. A modification of the Smith-Waterman or Sellers algorithms will find all segment pairs whose scores can not be improved by extension or trimming. These are called high-scoring segment pairs or HSPs.

To analyze how high a score is likely to arise by chance, a model of random sequences is needed. For proteins, the simplest model chooses the amino acid residues in a sequence independently, with specific background probabilities for the various residues. Additionally, the expected score for aligning a random pair of amino acid is required to be negative. Were this not the case, long alignments would tend to have high score

independently of whether the segments aligned were related, and the statistical theory would break down.

Just as the sum of a large number of independent identically distributed (i.i.d) random variables tends to a normal distribution, the maximum of a large number of i.i.d. random variables tends to an extreme value distribution. (We will elide the many technical points required to make this statement rigorous.) In studying optimal local sequence alignments, we are essentially dealing with the latter case. In the limit of sufficiently large sequence lengths m and n, the statistics of HSP scores are characterized by two parameters, K and λ. Most simply, the expected number of HSPs with score at least S is given by the formula.

$$E = kmn\, e^{-\lambda S} \qquad (1)$$

We call this the E-value for the score S.

This formula makes eminently intuitive sense. Doubling the length of either sequence should double the number of HSPs attaining a given score. Also, for an HSP to attain the score $2x$ it must attain the score x twice in a row, so one expects E to decrease exponentially with score. The parameters K and λ can be thought of simply as natural scales for the search space size and the scoring system respectively.

1.2.2.2.3 Bit scores

Raw scores have little meaning without detailed knowledge of the scoring system used, or more simply its statistical parameters K and λ. Unless the scoring system is understood, citing a raw score alone is like citing a distance without specifying feet, meters, or light years. By normalizing a raw score using the formula

$$S' = (\lambda S - \ln K)/(\ln 2) \qquad (2)$$

One attains a "bit score" S', which has a standard set of units. The E-value corresponding to a given bit score is simply

$$E = mn\, 2^{-S'} \qquad (3)$$

Bit scores subsume the statistical essence of the scoring system employed, so that to calculate significance one needs to know in addition only the size of the search space.

1.2.2.2.4 P-values

The number of random HSPs with score >= *S* is described by a Poisson distribution. This means that the probability of finding exactly *a* HSPs with score >=*S* is given by

$$e^{-E} E^{a} / a! \qquad (4)$$

where *E* is the *E*-value of *S* given by equation (1) above.
Specifically, the chance of finding zero HSPs with score >=*S* is e^{-E}, so the probability of finding at least one such HSP is

$$P = 1 - e^{-E} \qquad (5)$$

This is the *P*-value associated with the score *S*. For example, if one expects to find three HSPs with score >= *S*, the probability of finding at least one is 0.95. The BLAST programs report *E*-value rather than *P*-values because it is easier to understand the difference between, for example, *E*-value of 5 and 10 than *P*-values of 0.993 and 0.99995. However, when $E < 0.01$, *P*-values and *E*-value are nearly identical.

1.2.2.2.5 Database searches

The *E*-value of equation (1) applies to the comparison of two proteins of lengths *m* and *n*. How does one assess the significance of an alignment that arises from the comparison of a protein of length *m* to a database containing many different proteins, of varying lengths? One view is that all proteins in the database are *a priori* equally likely to be related to the query. This implies that a low *E*-value for an alignment involving a short database sequence should carry the same weight as a low *E*-value for an alignment involving a long database sequence.

To calculate a "database search" *E*-value, one simply multiplies the pairwise-comparison *E*-value by the number of sequences in the database. Recent versions of the FASTA protein comparison programs take this approach.

An alternative view is that a query is *a priori* more likely to be related to a long than to a short sequence, because long sequences are often composed of multiple distinct domains. If we assume the *a priori* chance of relatedness is proportional to sequence length, then the pairwise *E*-value involving a database sequence of length *n* should be multiplied by *N/n*, where *N* is the total length of the database in residues. Examining equation (1), this can be accomplished simply by treating the database as a single long sequence of length *N*. The BLAST programs take this approach to calculating database *E*-value.

Notice that for DNA sequence comparisons, the length of database records is largely arbitrary, and therefore this is the only really tenable method for estimating statistical significance.

1.2.2.2.6 The statistics of gapped alignments

The statistics developed above have a solid theoretical foundation only for local alignments that are not permitted to have gaps. However, many computational experiments and some analytic results strongly suggest that the same theory apply as well to gap alignments. For ungapped alignments, the statistical parameters can be calculated, using analytic formulas, from the substitution scores and the background residue frequencies of the sequences being compared. For gapped alignments, these parameters must be estimated from a large-scale comparison of "random" sequences.

Some database search programs, such as FASTA or various implementation of the Smith-Waterman algorithm, produce optimal local alignment scores for the comparison of the query sequence to every sequence in the database. Most of these scores involve unrelated sequences, and therefore can be used to estimate λ and *K*. This approach avoids the artificiality of a random sequence model by employing real sequences, with their

attendant internal structure and correlations, but it must face the problem of excluding from the estimation scores from pairs of related sequences.

The BLAST programs achieve much of their speed by avoiding the calculation of optimal alignment scores for all but a handful of unrelated sequences. The must therefore rely upon a pre-estimation of the parameters λ and K, for a selected set of substitution matrices and gap costs. This estimation could be done using real sequences, but has instead relied upon a random sequence model, which appears to yield fairly accurate results.

1.2.2.2.7 Edge effects

The statistics described above tend to be somewhat conservative for short sequences. The theory supporting these statistics is an asymptotic one, which assumes an optimal local alignment can begin with any aligned pair of residues. However, a high-scoring alignment must have some length, and therefore can not begin near the end of either of two sequences being compared.

This "edge effect" may be corrected by calculating an "effective length" for sequences. the BLAST programs implement such a correction.

For sequences longer than about 200 residues the edge effect correction is usually negligible.

1.2.2.2.8 The choice of substitution scores

The results a local alignment program produces depend strongly upon the scores it uses. No single scoring scheme is best for all purposes, and an understanding of the basic theory of local alignment scores can improve the sensitivity of one's sequence analyses. As before, the theory is fully developed only for scores used to find ungapped local alignments. So we start with that case.

A large number of different amino acid substitution scores, based upon a variety of rationales, have been described.

However, the scores of any substitution matrix with negative expected scores can be written uniquely in the form

$$S_{ij} = (\ln(q_{ij}/(p_i p_j))/\lambda \qquad (6)$$

where the q_{ij}, called target frequencies, are positive numbers that sum to 1, the p_i are background frequencies for the various residues, and λ is a positive constant. The λ here is identical to the λ of equation (1).

Multiplying all the scores in a substitution matrix by a positive constant does not change their essence: an alignment that was optimal using the original scores remains optimal. Such multiplication alters the parameter λ but not the target frequencies q_{ij}. Thus, up to a constant scaling factor, every substitution matrix is uniquely determined by its target frequencies. These frequencies have a special significance.

A given class of alignments is best distinguished from chance by the substitution matrix whose target frequencies characterize the class.

To elaborate, one may characterize a set of alignments representing homologous protein regions by the frequency with which each possible pair of residues is aligned. If valine in the first sequence and leucine in the second appear in 1% of all alignment positions, the target frequency for (valine, leucine) is 0.01.

The most direct way to construct appropriate substitution matrices for local sequence comparison is to estimate target and background frequencies, and calculate the corresponding log-odds scores of formula (6). These frequencies in general can not be derived from first principles, and their estimation requires empirical input.

1.2.2.2.9 The PAM and BLOSUM amino acid substitution matrices

While all substitution matrices are implicitly of log-odds form, the first explicit construction using formula (6) was by Dayhoff and coworkers. From a study of observed residue replacements in closely related proteins, they constructed the PAM (for "point accepted mutation") model of molecular evolution. One "PAM" corresponds to an average change in 1% of all amino acid positions.

After 100 PAMs of evolution, not every residue will have changed: some will have mutated several times, perhaps returning to their original state, and others not at all. Thus it is possible to recognize as homologous proteins separated by much more than 100 PAMs. Note that there is no general correspondence between PAM distance and evolutionary time, as different protein families evolve at different rates. Using the PAM model, the target frequencies and the corresponding substitution matrix may be calculated for any given evolutionary distance. When two sequences are compared, it is not generally known a priori what evolutionary distance will best characterize any similarity they may share. Closely related sequences, however, are relatively easy to find even will non-optimal matrices, so the tendency has been to use matrices tailored for fairly distant similarities. For many years, the most widely used matrix was PAM-250, because it was the only one originally published by Dayhoff. Dayhoff's formalism for calculating target frequencies has been criticized, and there have been several efforts to update her numbers using the vast quantities of derived protein sequence data generated since her work. These newer PAM matrices do not differ greatly from the original ones.

An alternative approach to estimating target frequencies, and the corresponding log-odds matrices, has been advanced by Henikoff & Henikoff. They examine multiple alignments of distantly related protein regions directly, rather than extrapolate from closely related sequences. An advantage of this approach is that it cleaves closer to observation; a disadvantage is that it yields no evolutionary model. A number of tests suggest that the "BLOSUM" matrices produced by this method generally are superior to the PAM matrices for detecting biological relationships.

1.2.2.2.10 DNA substitution matrices

Our theoretical development concerning the optimality of matrices constructed using equation (6) unfortunately is invalid as soon as gaps and associated gap scores are introduced, and no more general theory is available to take its place. However, if the gap scores employed are sufficiently large, one can expect that the optimal substitution scores for a given application will not change substantially. In practice, the same substitution scores have been applied fruitfully to local alignments both with and without gaps. Appropriate gap scores have been selected over the years by trial and error, and most alignment programs will have a default set of gap scores to go with a default set of substitution scores. If the user wishes to employ a different set of substitution scores, there is no guarantee that the same gap scores will remain appropriate.

No clear theoretical guidance can be given, but "affine gap scores", with a large penalty for opening a gap and a much smaller one for extending it, have generally proved among the most effective.

1.2.2.2.11 Low complexity sequence regions

There is one frequent case where the random models and therefore the statistics discussed here break down. As many as one fourth of all residues in protein sequences occur within regions with highly biased amino acid composition. Alignments of two regions with similarly biased composition may achieve very high scores that owe virtually nothing to residue order but are due instead to segment composition. Alignments of such "low complexity" regions have little meaning in any case: since these regions most likely arise by gene slippage, the one-to-one residue correspondence imposed by alignment is not valid. While it is worth noting that two proteins contain similar low complexity regions, they are best excluded when constructing alignments.

The BLAST programs employ the SEG algorithm to filter low complexity regions from proteins before executing a database search.

1.2.2.12 BLAST Tool Outline

BLAST® (Basic Local Alignment Search Tool) is a set of similarity search programs designed to explore all of the available sequence databases regardless of whether the query is protein or DNA. The BLAST programs have been designed for speed, with a minimal sacrifice of sensitivity to distant sequence relationships. The scores assigned in a BLAST search have a well-defined statistical interpretation, making real matches easier to distinguish from random background hits. BLAST uses a heuristic algorithm which seeks local as opposed to global alignments and is therefore able to detect relationships among sequences which share only isolated regions of similarity.

The BLAST algorithm was written balancing speed and increased sensitivity for distant sequence relationships. Instead of relying on global alignments (commonly seen in multiple sequence alignment programs) BLAST emphasizes regions of local alignment to detect relationships among sequences which share only isolated regions of similarity (Altschul et al., 1990). Therefore, BLAST is more than a tool to view sequences aligned with each other or to find homology, but a program to locate regions of sequence similarity with a view to comparing structure and function.

BLAST can be run in a standalone or in a remote machine. The main advantage of a standalone BLAST is to be able to create your own BLAST database. This make BLAST query more efficient, fast, and independent of the remote server.

BLAST tool mainly consists of four programs: blastn, blastp, blastx, tblastn, and tblastx.

blastn: Compares a nucleotide query sequence against a nucleotide sequence database.

blastp: Compares an amino acid query sequence against a protein sequence database.

blastx: Compares a nucleotide query sequence translated in all reading frames against a protein sequence database. You could use this option to find potential translation products of an unknown nucleotide sequence.

tblastn: Compares a protein query sequence against a nucleotide sequence database dynamically translated in all reading frames.

tblastx: Compares the six-frame translations of a nucleotide query sequence against the six-frame translations of a nucleotide sequence database.

Chapter 2 BlastMerge Tool

With an enormous amount of data stored in databases and data warehouses, it is increasingly important to develop powerful tools for analysis of such data and mining interesting knowledge from it. In DNA sequence search field we have similar situation.

If you use the standalone BLAST search tool to search protein and nucleic acid sequences and compare them against a selection of local NCBI databases, you have to maintain your local database. Since NCBI update its database every day, if you want to search your query on the latest database, you have to download the whole latest database before you can run queries. Since the NCBI database is huge (by now more than 100M), downloading the whole database is a time-consuming work. With the continue increase of the site of the NCBI database, this issue will become more significant.

The best solution is to re-download the whole database only periodically. Each day before you run the BLAST program, you download the update database, such as month.aa. This update database is smaller than the whole database. You search your query on both the whole local database and the update database, the merge of these two query results is equivalent to the query results on the latest whole database. The blastMerge program is designed for this purpose.

2.1 Object-oriented Design in BlastMerge

The BlastMerge program is the tool that can merge the query results of blastn and blastp program. Originally the idea is from Dr. Clement Lam and Dr. Gregory Butler. The key point of the BlastMerge design is to use object-oriented methodology to model query results. First let us analyze the format of query results. The query results generated by blastn and blabtp are different.

We use Object Oriented Methodology and Data Mining theory to design the BlastMerge. In object-oriented modeling, classes, objects, and their relationships are the primary modeling elements. Classes and objects model what it is in the system we are trying to describe, and the relationships between them reveals how they are structured in terms of each other. Classification has been used for thousands of simplify descriptions of complex systems, so that we can more easily understand them. When using object-oriented programming to build software systems, classes and relationships become the actual code.

Data mining is a process of inferring knowledge from such huge data. Data Mining has three major components *Clustering* or *Classification*, *Association Rules* and *Sequence Analysis*. By simple definition, in classification/clustering we analyze a set of data and generate a set of grouping rules, which can be used to classify future data. An association rule is a rule that implies certain association relationships among a set of objects in a database. In this process we discover a set of association rules at multiple levels of abstraction from the relevant set(s) of data in a database. In *sequential Analysis*, we seek to discover patterns that occur in sequence.

The analysis to combine object-oriented modeling and data mining method is specialized for huge data processing. Using object-oriented methodology we can easily model real world system. With data mining theory we can infer knowledge from such huge data. Data mining involves knowledge base and *machine-learning*. It contains two aspects: heuristic and some inferring algorithms. Therefore using dada mining theory to analyze DNA sequence database and its query output is an interesting field.

2.2 The query result of blast program

2.2.1 The query result for blastn program

To design BlastMerge tool, first we have to model DNA sequence query output, that is, understand what it is in the query outputs of the blast tool. We classified them as follows.

The query result of the blastn program can be classified into the following parts (for a sample of the detail output, refer to appendix I).

1) Program name and version.

2) Reference. It defines the developer of the blast tool.

3) Query. It defines DNA sequence that is to be used to compare with database sequence.

4) Database name, which database was searched.

5) Sequence producing significant alignments. This part consists of sequence record title, Score, and E value.

6) A list of match records.

For blastn each record contains following parts.

a) Full sequence record title.

b) Sequence length.

A List of High Scoring pairs (HSPs)[parts (c) to (g)].

They represent "pairwise alignment" among High Scoring pairs (HSPs). A "pairwise alignment" is one in which the aligned positions of the query part and the database match part (the subject) are arranged with one vertical space between them. In protein alignments, identical residues are listed in the middle. Conserved residues are represented by plus signs. In DNA alignments, vertical lines connect identical residues. Gaps are represented as dashes within the query or subject sequence.

Due to filtering, an amino acid query sequence may contain X's in place of low complexity sequences. (N's in a nucleic acid query). This accounts for a decrease in identity and increase in E value than would otherwise be seen in a match of a query against the identical or other highly related sequences in the database.

More than one alignment per database entry may be listed among the HSPs. The parts (c) to (g) may be repeated several times.

c) Score and E value.

The score of an alignment, S, calculated as the sum of substitution and gap scores. Substitution scores are given by a look-up table (see PAM, BLOSUM). Gap scores are typically calculated as the sum of G, the gap opening penalty and L, the gap extension penalty. For a gap of length n, the gap cost would be G+Ln. The choice of gap costs, G and L is empirical.

Expectation value, E. The number of different alignments with scores equivalent to or better than S that are expected to occur in a database search by chance. The lower the E value, the more significant the score.

d) Identities. The extent to which two (nucleotide or amino acid) sequences are invariant.

e) Strand.

f) Query.

g) Sbjct.

Parts (7) to (9) specify the Values for lambda, K, and H calculated from the results of the search (ungapped, gapped).

7) λ

A statistical parameter used in calculating BLAST scores that can be thought of as a natural scale for scoring system. The value lambda is used in converting a raw score (S) to a bit score (S').

8) K

A statistical parameter used in calculating BLAST scores that can be thought of as a natural scale for search space size. The value K is used in converting a raw score (S) to a bit score (S').

9) H

H is the relative entropy of the target and background residue frequencies. H can be thought of as a measure of the average information (in bits) available per position that distinguishes an alignment from chance. At high values of H, short alignments

can be distinguished from chance, whereas at lower H values, a longer alignment may be necessary.

10) Matrix.

Specify which matrix was used during the search.

A key element in evaluating the quality of a pairwise sequence alignment is the "substitution matrix", which assigns a score for aligning any possible pair of residues. The matrix used in a BLAST search can be changed depending on the type of sequences you are searching with.

11) Gap Penalty.

Gap creation and gap extension costs.

A space introduced into an alignment to compensate for insertions and deletions in one sequence relative to another. To prevent the accumulation of too many gaps in an alignment, introduction of a gap causes the deduction of a fixed amount (the gap score) from the alignment score. Extension of the gap to encompass additional nucleotides or amino acid is also penalized in the scoring of an alignment.

Parts 12) to 22) specify BLAST statistics.

12) Number of Hits to DB.

13) Number of sequence.

14) Number of extensions.

15) Number of successful extensions.

16) Number of sequence better than.

17) Length of query.

18) Length of database.

19) Effective HSP length.

High-scoring segment pair. Local alignments with no gaps that achieve one of the top alignment scores in a given search.

20) Effective length of database.

21) Effective search space.

22) Effective search space used.

23) T.

24) A.

25) X1.

26) X2.

27) S1

28) S2.

2.2.2 Query Result for blastp

The query results between blastn and blastp have minor differences. In the query results of blastn, HSPs contain a list of Score, Expect, Identities, Strand, Query and Sbjct corresponding to each hit record. In the query results of blastp HSPs contain Score, Expect, Identities, Positives, gap and a list of Query and Sbjct corresponding to each hit record.

2.3 Modeling Query Output

With object-oriented modeling methodology we consider the whole query output as an object. We use class AlignmentRecord to represent the data of a query result. It is a container class, which contains the information one instance per query. That is item 1) to 5), item 7) to 28), and a list of match records. We use a sorted linked list to represent this part of data. Class AlignmentRecord contains a pointer to the sorted linked list. Each node of the sorted linked list is an AligmentDetail. Its data members are item a) to g). In the real world, we cannot predict the number of the matched records. So we use a sorted linked list to store the number of matched records. Thus, there is no size limitation to the records. With the use of sorted linked list BlastMerge tool it is easily to output merge query result in sequence. For very huge query output processing it will be more efficient. With the use of a pointer to linked list, when AlignmentRecord construct it only allocates a pointer to linked list, the real linked list is constructed during parsing the query output file. The objects diagram is as follows.

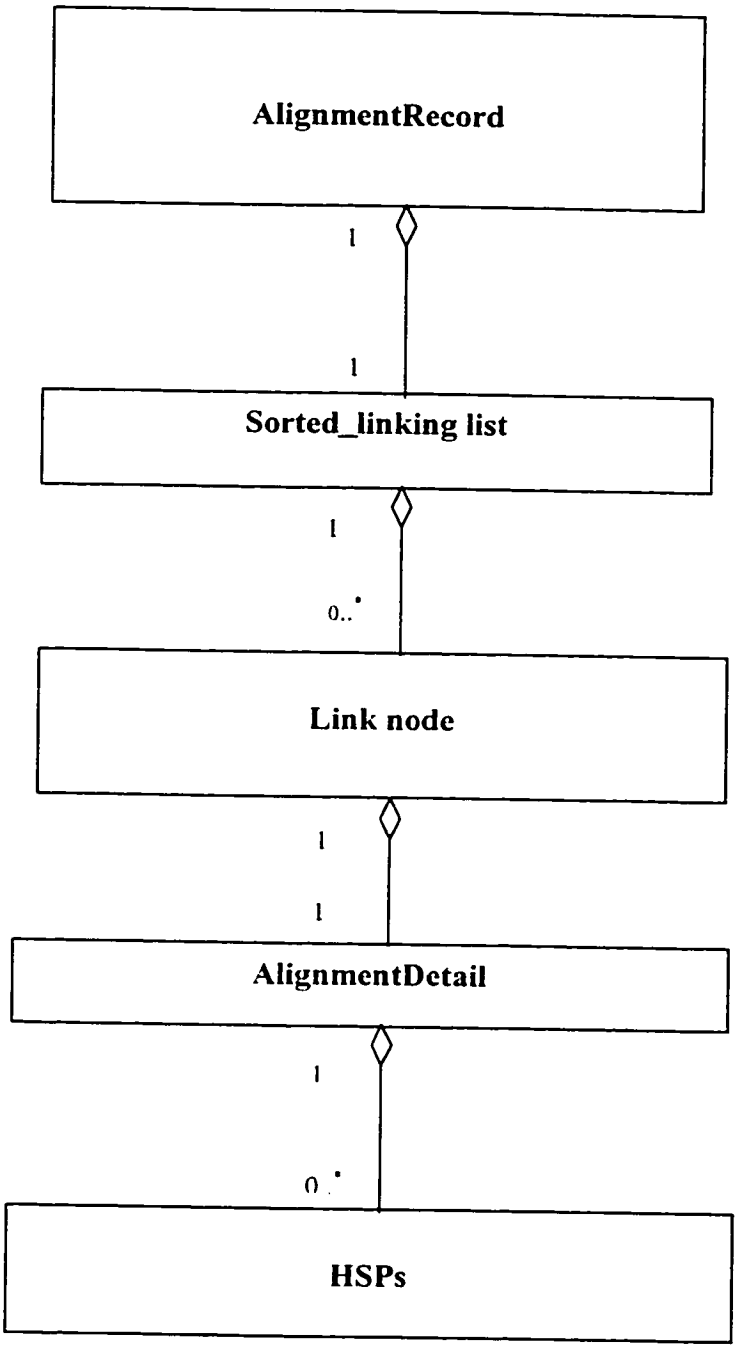

Figure 1 Objects diagram in BlastMerge

2.3.1 Class AlignmentRecord

Class AlignmentRecord represents the output file per query. It contains the information one instance per query and a sorted linked list. Its class diagram is

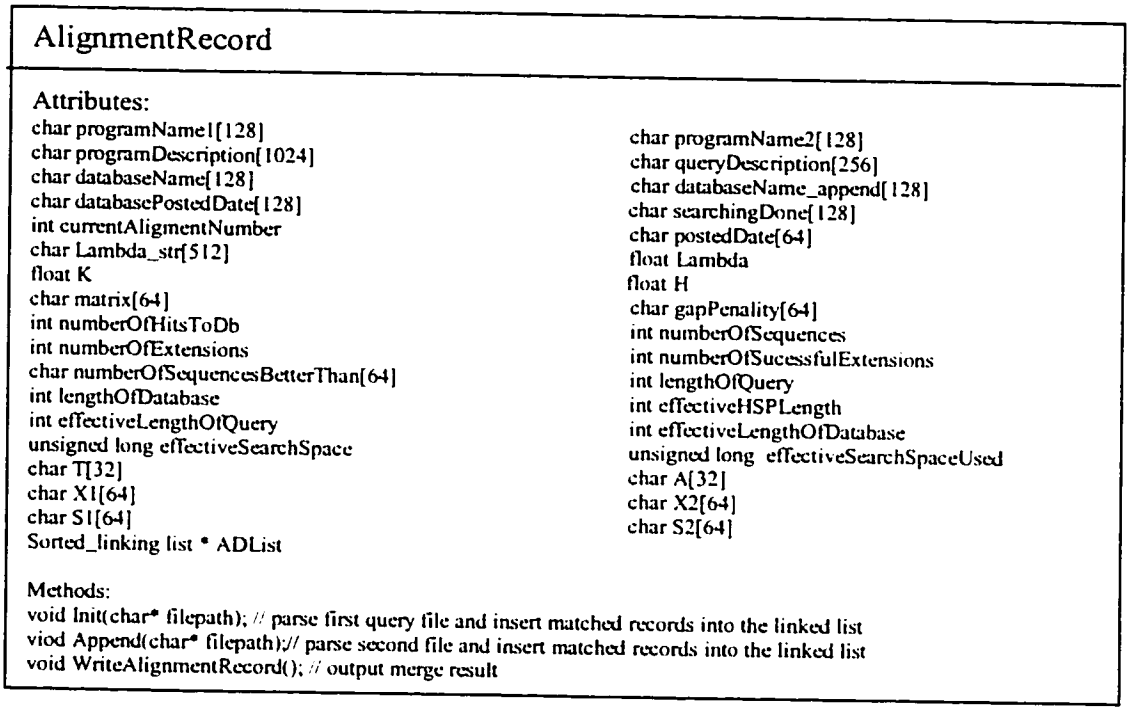

Figure 2. Class diagram AlignmentRecord

2.3.2 Class AlignmentDetail

Class AlignmentDetail contains a set of HSPs and the current score number. Class diagram is

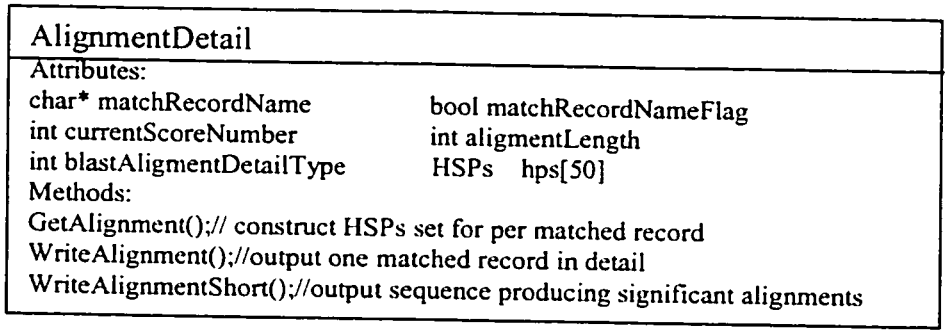

Figure 3 class diagram of AlignmentDetail.

2.3.3 Class HSPs

Class HSPs represents a matched record behavior.

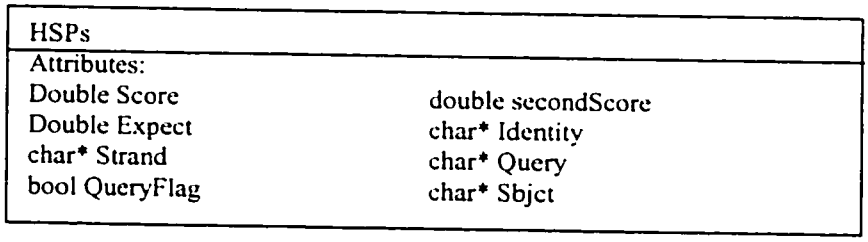

Figure 4 Class diagram of HSPs

For sorted linked list it is standard class. We omit its class description.

2.3.4 Sequence diagram of parsing query output files in BlastMerge tool

Sequence diagrams illustrate how objects interact with each other. They focus on message sequences, that is, how messages are sent and received between a number of objects. Sequence diagram has two axes: the vertical axis shows time and horizontal axis shows a set of objects. A sequence diagram also reveals the interaction for a specific scenario— a specific interaction between the objects that happens at some point in time during the system's execution. Parse query file is a major scenario when BlastMerge tool merges query files. We give the sequence diagram as follows.

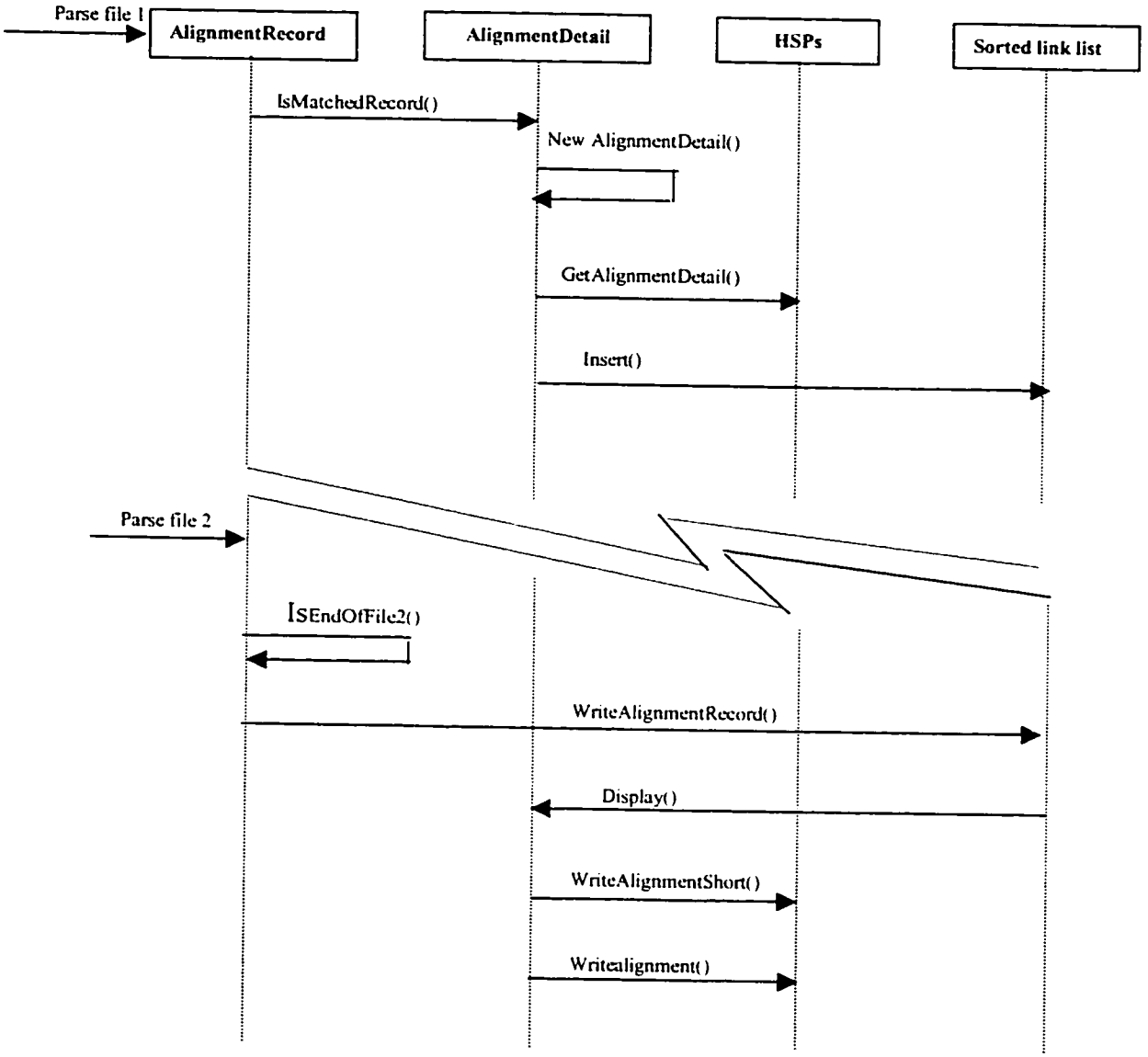

Figure 5 Sequence diagram in parsing file

When the BlastMerge tool merges two query results, it reads first the query result and inserts them into a sorted linked list, then reads the second query result and inserts them into the sorted linked list.

2.4 Some sample output

The BlastMerge program can merge the outputs of two queries, one against a database and the other against the update to the database. It can handle ouputs from blastn and blastp. For example, we downloaded the DNA sequence database yeast.aa and its update month.aa. As a test, we use the query DNA sequence.

>gi|416278|dbj|D11125|VIBVCI384 V. cholerae rfbT3 gene for Ogawa specific antigen in the case of Inaba serotype, complete cds
TCTAATAGAACTTTTGATGAGATTTTTAACATAATAAACTCAAAATTCGGAAG
TAAGGCATATTTTATTCATCCATTATCATCCGCTGAACATCCTGAGTTTAATA
AAGCAACGCAGGATATTAATGGGAATATCTGTTTTAAATATGTATCATAAAA
TAATTTAATATATTCCGTATGTCATTGCAAGTTCAACAGACATTTCCGAAGA

With blastp we obtain two query output yeast.xout and month.xout. We run blastMerge. Then we obtain the merged output out.data.

We have also downloaded the DNA sequence database yeast.nt and pdbnt. With blastn, we get two query output yeast.nt.out and pdbnt.out. We run BlasetMerge to get merge output out.data. Since the query output of blastp is too big, so we give only the query outputs from the DNA database yeast and pdbnt. In order to limit the number of pages, we take only the first four highest scoring hit records. Then we give the merge output of BlastMerge. The detail is in Appendix I.

From BlastMerge output in Appendix I we find that, for sequence producing significant alignments and hit records, the merge output of original database and its update is exactly same as the query output of whole database. We take it from original database query output since statistics part is hard to merge. However, since its daily update is smaller compared to the latest whole database the statistics part of original is dominant. With the

increase of DNA sequence database, the search space difference between the whole database and its update will be decreased.

2.5 A brief user guide

The BlastMerge is a program to merge query outputs of blastn or blastp. It is easy of use.

Step 1. You download the databases that you would like to search from the NCBI web site. For example, blast database ftp site is (ftp://ftp.ncbi.nlm.nih.gov/blast/db/).

Step 2. Format the downloaded databases by running the formatdb tool.

Before running the blast program blastall, the formatdb should be used to format the FASTA databasefor both protein and DNA database. The advantage is that ambiguity information is now retrieved from the file produced by formatdb, rather than from the original FASTA file. The original FASTA file is no longer needed for the BLAST runs. This saves both disk-space and improves performance, as the large FASTA file is no longer needed by BLAST.

A simple example of using formatdb is:

formatdb –i ecoli.nt –p F –o T

where the arguments have following meaning:

-i — input file for formatting (this parameter must be set) [File In]

ecoli.nt — the database you want to format.

-p — type of file.

T — protein.

F — nucleotide [T/F] Optional.

-o — Parse options

T —True: Parse SeqId and create indexes.

F — False: Do not parse SeqId. Do not create indexes.

Step 3. Run blastn or blastp to get the query outputs.

The blastall may be used to perform all five flavors of blast comparison (balstn, blastp, blastx, blasttn, or tblastx). One may obtain the help on blastall options by executing "blastall –" (note the dash). A typical use of blastall to perform a blastn search (nucl. Vs. nucl.) of a file called QUERY would be

blastall –p blastn -d nr -i QUERY -o out.QUERY

-p — Prgram name [String]

Input should be one of "balstn", "blastp", "blastx", "blasttn", or "tblastx".

-d — Databse [String]

default = nr

The database specified must first be formatted with formatdb. Multiple database names (braketed by quotations) will be accepted. An example would be

-d "nr est"

-i — Query File [File In]

default = stdin

The query should be in FAST format. If multiple FAST entries are in the input file.

-o — BLAST report Output File [File In] Optional

Step 4. Copy the two query outputs to the directory where the BlastMerge executable exists.

Step 5. Run the BalstMerge, when it prompts you, input two query output file names. After you type the names of two query outputs it will generate the merge output (out.data) in the same directory as the BlastMerge executable exists.

Chapter 3 Conclusion

3.1 Conclusion

With the explosive expansion of DNA sequence database any tool that directly processes data in the database will become expensive. Find the way to indirectly process the output result of the tool is the best solution. BlastMerge is designed for this purpose. BlastMerge is an efficient tool to process the query output. Currently it can merge the query outputs of blaptn and blastp. We merge two query outputs, one against a database and the other against the update to the database. For sequence producing significant alignment and hit records the output of BlastMerge is exactly the same as the query output against the latest database. For the statistics part, BlastMerge takes it from the original database, since the search space of the original database is predominant. With the continue increase of DNA sequence database the search space of the original database will become more significant compared to that of its update.

3.2 Future work

By now BlastMerge tool can only process outputs from blastn and blastp. Improving BlastMerge to process the query outputs of other blast tool is one of possible the future works.

Current BlastMerge version can only process a single query output. In future we should improve BlastMerge tool to handle multiple queries.

Applying Data mining theory to the BlastMerge tool should be considered in the future since machine-learning and knowledge base theory is inevitable in processing huge database.

4. Reference

[1] Andreas D.Baxevanis and B.F.Francis Ouellette (1998) BIOINFORMATICS A Practical Guide to the Analysis of Genes and Proteins. John Wiley & Sons.Inc.

[2] Stephen F. Altschul, Warren Gish,..etc. Basical Local Search Tool. J. Mol. Boil. (1990) 215, 408-410.

[3] NCBI http://www.ncbi.nlm.nih.gov/

[4] William R.Pearson. Effective Protein Sequence Comparison. Methods in Enzymology, 266:227-258,1996.

[5] Steven Henikoff and Jorja G. Henikoff. Amino acid substitution matrices fromprotein block. In Proceedings of the National Academy of science, volume 89, page 10915-10919,10919, Number 1992.

[6] Altschul SF, Gish W, Miller W, Myers EW, and Lipman DJ. Basic local alignment search tool. J Mol Biol,215(3):403-410, Oct 1990.

[7] Bairoch A and Apweiler R. The SWISS-PROT protein sequence sata bank and its supplement TrEMBL in 1998. Nucleic Acids Research,26(1):38-42,1998

[8] Barker WC, Garavelli JS, Haft DH, Hunt LT, Marzec CR, Orcutt BC, Srinivasarao GY, Yeh LSL,Ledley RS, Mewes HW, Pfeiffer F, and Tsugita A. The PIR-International Protein Sequence Database. Nucleic Acids Research,26(1):27-32,1998.

[9] Stephen F. Altschul, Thomas L,Madden, Alejandro A.Scaeffer, Jinghui zhang. Zheng Zhang, Webb Miller, and David J.Lipman. Gapped BLAST and PSIBLAST: a new generation of protein database search programs. Nucleic Acids Research, 25(17), 1997.

[10] David J.Lipman and William R.Pearson. Rapid and Sensitive Protein Similarity Searches. Science, 227:1435-1441, 1985.

[11] James Rumbaugh, Michael Blaha, William Premerlance, Frederick Eddy, and William Loreenson. Object-Oriented Modeling and Design. Prentice Hall,1991.

[12] Benson DA, Boguski MS, Lipman DJ, Ostell J, and Oullette BF. Genbank, Nucleic Acids Research,26(1)1-7,1998

5. Appendix I

5.1 Quey output (yeast.nt.out) for yeast.nt

Quey output (yeast.nt.out) for yeast.nt is as follow.

```
BLASTN 2.0.14 [Jun-29-2000]

Reference: Altschul, Stephen F., Thomas L. Madden, Alejandro A. Schaffer,
Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman (1997),
"Gapped BLAST and PSI-BLAST: a new generation of protein database search
programs", Nucleic Acids Res. 25:3389-3402.

Query= gi|416278|dbj|D11125|VIBVCI384 V. cholerae rfbT3 gene for Ogawa
specific antige
         (231 letters)

Database: E:\major_report\db_download\yeast.nt\yeast.nt
           17 sequences; 12,155,026 total letters


                                                                   Score     E
Sequences producing significant alignments:                       (bits)  Value

ref|NC_001146.1| Saccharomyces cerevisiae chromosome XIV, comple...    43  3e-004
ref|NC_001144.1| Saccharomyces cerevisiae chromosome XII, comple...    35  0.073
ref|NC_001147.1| Saccharomyces cerevisiae chromosome XV, complet...    33  0.30
ref|NC_001142.1| Saccharomyces cerevisiae chromosome X, complete...    33  0.30

>ref|NC_001146.1| Saccharomyces cerevisiae chromosome XIV, complete chromosome sequence
          Length = 784328

 Score = 43.2 bits (21), Expect = 3e-004
 Identities = 21/21 (100%)
 Strand = Plus / Plus


Query: 96    attatcatccgctgaacatcc 116
              |||||||||||||||||||||
Sbjct: 44808 attatcatccgctgaacatcc 44828


 Score = 33.0 bits (16), Expect = 0.30
 Identities = 16/16 (100%)
 Strand = Plus / Minus


Query: 33     ttttgatgagattttt 48
               ||||||||||||||||
Sbjct: 429548 ttttgatgagattttt 429533


 Score = 31.0 bits (15), Expect = 1.2
 Identities = 15/15 (100%)
 Strand = Plus / Plus


Query: 145   aatgggaatatctgt 159
              |||||||||||||||
```

```
Sbjct: 21710 aatgggaatatctgt 21724


 Score = 31.0 bits (15), Expect = 1.2
 Identities = 15/15 (100%)
 Strand = Plus / Plus


Query: 24     taatagaacttttga 38
              ||||||||||||||
Sbjct: 557276 taatagaacttttga 557290


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 14/14 (100%)
 Strand = Plus / Minus


Query: 164    aatatgtatcataa 177
              ||||||||||||||
Sbjct: 420911 aatatgtatcataa 420898


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 14/14 (100%)
 Strand = Plus / Minus


Query: 123    taataaagcaacgc 136
              ||||||||||||||
Sbjct: 309783 taataaagcaacgc 309770


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 14/14 (100%)
 Strand = Plus / Plus


Query: 152    atatctgttttaaa 165
              ||||||||||||||
Sbjct: 738969 atatctgttttaaa 738982


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 14/14 (100%)
 Strand = Plus / Minus


Query: 83     attttattcatcca 96
              ||||||||||||||
Sbjct: 440715 attttattcatcca 440702


>ref|NC_001144.1| Saccharomyces cerevisiae chromosome XII, complete chromosome sequence
          Length = 1078172

 Score = 35.1 bits (17), Expect = 0.073
 Identities = 17/17 (100%)
 Strand = Plus / Minus


Query: 154    atctgttttaaatatgt 170
              |||||||||||||||||
Sbjct: 155341 atctgttttaaatatgt 155325


 Score = 31.0 bits (15), Expect = 1.2
 Identities = 15/15 (100%)
 Strand = Plus / Minus
```

```
Query: 27       tagaacttttgatga 41
                ||||||||||||||||
Sbjct: 1041926 tagaacttttgatga 1041912


>ref|NC_001147.1| Saccharomyces cerevisiae chromosome XV, complete chromosome sequence
          Length = 1091283

 Score = 33.0 bits (16), Expect = 0.30
 Identities = 16/16 (100%)
 Strand = Plus / Minus


Query: 34      tttgatgagattttta 49
                ||||||||||||||||
Sbjct: 636999 tttgatgagattttta 636984


 Score = 31.0 bits (15), Expect = 1.2
 Identities = 15/15 (100%)
 Strand = Plus / Minus


Query: 212      tcaacagacatttcc 226
                ||||||||||||||||
Sbjct: 1041336 tcaacagacatttcc 1041322


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 14/14 (100%)
 Strand = Plus / Plus


Query: 54     aataaactcaaaat 67
               ||||||||||||||
Sbjct: 22740 aataaactcaaaat 22753


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 14/14 (100%)
 Strand = Plus / Minus


Query: 173     cataaaataattta 186
                ||||||||||||||
Sbjct: 766574 cataaaataattta 766561


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 14/14 (100%)
 Strand = Plus / Minus


Query: 159    ttttaaatatgtat 172
               ||||||||||||||
Sbjct: 72761 ttttaaatatgtat 72748


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 14/14 (100%)
 Strand = Plus / Minus


Query: 176     aaaataatttaata 189
                ||||||||||||||
Sbjct: 736136 aaaataatttaata 736123


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 14/14 (100%)
 Strand = Plus / Minus
```

```
Query: 34      tttgatgagatttt 47
               ||||||||||||||
Sbjct: 942314 tttgatgagatttt 942301


>ref|NC_001142.1| Saccharomyces cerevisiae chromosome X, complete chromosome sequence
          Length = 745440

 Score = 33.0 bits (16), Expect = 0.30
 Identities = 16/16 (100%)
 Strand = Plus / Minus


Query: 150    gaatatctgttttaaa 165
               ||||||||||||||||
Sbjct: 180402 gaatatctgttttaaa 180387


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 14/14 (100%)
 Strand = Plus / Minus


Query: 211    ttcaacagacattt 224
               ||||||||||||||
Sbjct: 582789 ttcaacagacattt 582776


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 17/18 (94%)
 Strand = Plus / Minus


Query: 158    gttttaaatatgtatcat 175
               |||||||||||| ||||||
Sbjct: 396997 gttttaaatatttatcat 396980


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 14/14 (100%)
 Strand = Plus / Minus


Query: 119    agtttaataaagca 132
               ||||||||||||||
Sbjct: 669781 agtttaataaagca 669768


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 14/14 (100%)
 Strand = Plus / Plus


Query: 54     aataaactcaaaat 67
               ||||||||||||||
Sbjct: 16983 aataaactcaaaat 16996


  Database: E:\major_report\db_download\yeast.nt\yeast.nt
    Posted date:  Mar 14, 2001  8:54 PM
  Number of letters in database: 12,155,026
  Number of sequences in database:  17

Lambda     K      H
    1.41    0.715     1.33

Gapped
Lambda     K      H
    1.41    0.715     1.33
```

```
Matrix: blastn matrix:1 -3
Gap Penalties: Existence: 5, Extension: 2
Number of Hits to DB: 2504
Number of Sequences: 17
Number of extensions: 2504
Number of successful extensions: 739
Number of sequences better than 10.0: 15
length of query: 231
length of database: 12,155,026
effective HSP length: 16
effective length of query: 215
effective length of database: 12,154,754
effective search space: 2613272110
effective search space used: 2613272110
T: 0
A: 0
X1: 5 (10.2 bits)
X2: 9 (18.3 bits)
S1: 12 (24.9 bits)
S2: 14 (29.0 bits)
```

5.2 Query output (pdbnt.out) for pdbnt

Query output(pdbnt.out) for pdbnt is as follow.

```
BLASTN 2.0.14 [Jun-29-2000]

Reference: Altschul, Stephen F., Thomas L. Madden, Alejandro A. Schaffer,
Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman (1997),
"Gapped BLAST and PSI-BLAST: a new generation of protein database search
programs", Nucleic Acids Res. 25:3389-3402.

Query= gi|416278|dbj|D11125|VIBVCI384 V. cholerae rfbT3 gene for Ogawa
specific antige
         (231 letters)

Database: E:\major_report\blast_download\pdbnt\pdbnt
           2787 sequences; 53,920 total letters

Searching.....................................................done

                                                                  Score     E
Sequences producing significant alignments:                       (bits)  Value

pdb|1F66|J Chain J, 2.6 A Crystal Structure Of A Nucleosome Core...    25  0.16
pdb|1F66|I Chain I, 2.6 A Crystal Structure Of A Nucleosome Core...    25  0.16
pdb|1EQZ|J Chain J, X-Ray Structure Of The Nucleosome Core Parti...    25  0.16
pdb|1EQZ|I Chain I, X-Ray Structure Of The Nucleosome Core Parti...    25  0.16

>pdb|1F66|J Chain J, 2.6 A Crystal Structure Of A Nucleosome Core Particle
           Containing The Variant Histone H2a.Z
          Length = 146

 Score = 24.9 bits (12), Expect = 0.16
 Identities = 12/12 (100%)
 Strand = Plus / Minus


Query: 103 tccgctgaacat 114
           ||||||||||||
Sbjct: 72  tccgctgaacat 61


>pdb|1F66|I Chain I, 2.6 A Crystal Structure Of A Nucleosome Core Particle
           Containing The Variant Histone H2a.Z
```

```
          Length = 146

 Score = 24.9 bits (12), Expect = 0.16
 Identities = 12/12 (100%)
 Strand = Plus / Minus


Query: 103 tccgctgaacat 114
           ||||||||||||
Sbjct: 72  tccgctgaacat 61


>pdb|1EQZ|J Chain J, X-Ray Structure Of The Nucleosome Core Particle At 2.5 A
           Resolution
          Length = 146

 Score = 24.9 bits (12), Expect = 0.16
 Identities = 12/12 (100%)
 Strand = Plus / Minus


Query: 103 tccgctgaacat 114
           ||||||||||||
Sbjct: 72  tccgctgaacat 61


>pdb|1EQZ|I Chain I, X-Ray Structure Of The Nucleosome Core Particle At 2.5 A
           Resolution
          Length = 146

 Score = 24.9 bits (12), Expect = 0.16
 Identities = 12/12 (100%)
 Strand = Plus / Minus


Query: 103 tccgctgaacat 114
           ||||||||||||
Sbjct: 72  tccgctgaacat 61


  Database: E:\major_report\blast_download\pdbnt\pdbnt
    Posted date:  Feb 8, 2001  9:45 PM
  Number of letters in database: 53,920
  Number of sequences in database:  2787

Lambda     K      H
    1.41    0.715     1.33

Gapped
Lambda     K      H
    1.41    0.715     1.33

Matrix: blastn matrix:1 -3
Gap Penalties: Existence: 5, Extension: 2
Number of Hits to DB: 9
Number of Sequences: 2787
Number of extensions: 9
Number of successful extensions: 9
Number of sequences better than 10.0: 5
length of query: 231
length of database: 53,920
effective HSP length: 11
effective length of query: 220
effective length of database: 23,263
effective search space:  5117860
effective search space used:  5117860
T: 0
A: 0
X1: 5 (10.2 bits)
X2: 9 (18.3 bits)
```

```
S1: 12 (24.9 bits)
S2: 10 (20.8 bits)
```

5.3 Outpt (out.data) for BlastMerge

Outpt(out.data) for BlastMerge as follow.

```
BLASTN 2.0.14 [Jun-29-2000]

BLASTN 2.0.14 [Jun-29-2000]

Reference: Altschul, Stephen F., Thomas L. Madden, Alejandro A. Schaffer,
Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman (1997),
"Gapped BLAST and PSI-BLAST: a new generation of protein database search
programs", Nucleic Acids Res. 25:3389-3402.


Query= gi|416278|dbj|D11125|VIBVCI384 V. cholerae rfbT3 gene for Ogawa
specific antige
         (231 letters)


Database: E:\major_report\db_download\yeast.nt\yeast.nt
           17 sequences; 12,155,026 total letters




Database: E:\major_report\blast_download\pdbnt\pdbnt
           2787 sequences; 53,920 total letters



                                                                    Score     E
Sequences producing significant alignments:                         (bits)  Value

ref|NC_001146.1| Saccharomyces cerevisiae chromosome XIV, comple...   43  3e-004
ref|NC_001144.1| Saccharomyces cerevisiae chromosome XII, comple...   35  0.073
pdb|1F66|J Chain J, 2.6 A Crystal Structure Of A Nucleosome Core...   25  0.160
pdb|1F66|I Chain I, 2.6 A Crystal Structure Of A Nucleosome Core...   25  0.160
pdb|1EQZ|J Chain J, X-Ray Structure Of The Nucleosome Core Parti...   25  0.160
pdb|1EQZ|I Chain I, X-Ray Structure Of The Nucleosome Core Parti...   25  0.160
ref|NC_001147.1| Saccharomyces cerevisiae chromosome XV, complet...   33  0.300
ref|NC_001142.1| Saccharomyces cerevisiae chromosome X, complete...   33  0.300

>ref|NC_001146.1| Saccharomyces cerevisiae chromosome XIV, complete chromosome sequence
          Length = 784328

 Score = 43.2 bits (21), Expect = 3e-004
 Identities = 21/21 (100%)
 Strand = Plus / Plus


Query: 96    attatcatccgctgaacatcc 116
              |||||||||||||||||||||
Sbjct: 44808 attatcatccgctgaacatcc 44828


 Score = 33.0 bits (16), Expect = 0.30
 Identities = 16/16 (100%)
 Strand = Plus / Minus


Query: 33     ttttgatgagattttt 48
```

```
                ||||||||||||||||
Sbjct: 429548 ttttgatgagattttt 429533


 Score = 31.0 bits (15), Expect = 1.2
 Identities = 15/15 (100%)
 Strand = Plus / Plus


Query: 145   aatgggaatatctgt 159
              ||||||||||||||
Sbjct: 21710 aatgggaatatctgt 21724


 Score = 31.0 bits (15), Expect = 1.2
 Identities = 15/15 (100%)
 Strand = Plus / Plus


Query: 24      taatagaacttttga 38
                ||||||||||||||
Sbjct: 557276 taatagaacttttga 557290


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 14/14 (100%)
 Strand = Plus / Minus


Query: 164     aatatgtatcataa 177
                ||||||||||||||
Sbjct: 420911 aatatgtatcataa 420898


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 14/14 (100%)
 Strand = Plus / Minus


Query: 123     taataaagcaacgc 136
                ||||||||||||||
Sbjct: 309783 taataaagcaacgc 309770


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 14/14 (100%)
 Strand = Plus / Plus


Query: 152     atatctgttttaaa 165
                ||||||||||||||
Sbjct: 738969 atatctgttttaaa 738982


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 14/14 (100%)
 Strand = Plus / Minus


Query: 83      attttattcatcca 96
                ||||||||||||||
Sbjct: 440715 attttattcatcca 440702


>ref|NC_001144.1| Saccharomyces cerevisiae chromosome XII, complete chromosome sequence
          Length = 1078172

 Score = 35.1 bits (17), Expect = 0.073
 Identities = 17/17 (100%)
 Strand = Plus / Minus
```

```
Query: 154    atctgttttaaatatgt 170
              |||||||||||||||||
Sbjct: 155341 atctgttttaaatatgt 155325


 Score = 31.0 bits (15), Expect = 1.2
 Identities = 15/15 (100%)
 Strand = Plus / Minus


Query: 27      tagaacttttgatga 41
              |||||||||||||||
Sbjct: 1041926 tagaacttttgatga 1041912


>pdb|1F66|J Chain J, 2.6 A Crystal Structure Of A Nucleosome Core Particle
           Containing The Variant Histone H2a.Z
          Length = 146

 Score = 24.9 bits (12), Expect = 0.16
 Identities = 12/12 (100%)
 Strand = Plus / Minus


Query: 103 tccgctgaacat 114
           ||||||||||||
Sbjct: 72  tccgctgaacat 61


>pdb|1F66|I Chain I, 2.6 A Crystal Structure Of A Nucleosome Core Particle
           Containing The Variant Histone H2a.Z
          Length = 146

 Score = 24.9 bits (12), Expect = 0.16
 Identities = 12/12 (100%)
 Strand = Plus / Minus


Query: 103 tccgctgaacat 114
           ||||||||||||
Sbjct: 72  tccgctgaacat 61


>pdb|1EQZ|J Chain J, X-Ray Structure Of The Nucleosome Core Particle At 2.5 A
           Resolution
          Length = 146

 Score = 24.9 bits (12), Expect = 0.16
 Identities = 12/12 (100%)
 Strand = Plus / Minus


Query: 103 tccgctgaacat 114
           ||||||||||||
Sbjct: 72  tccgctgaacat 61


>pdb|1EQZ|I Chain I, X-Ray Structure Of The Nucleosome Core Particle At 2.5 A
           Resolution
          Length = 146

 Score = 24.9 bits (12), Expect = 0.16
 Identities = 12/12 (100%)
 Strand = Plus / Minus


Query: 103 tccgctgaacat 114
           ||||||||||||
```

```
Sbjct: 72  tccgctgaacat 61
>ref|NC_001147.1| Saccharomyces cerevisiae chromosome XV, complete chromosome sequence
          Length = 1091283

 Score = 33.0 bits (16), Expect = 0.30
 Identities = 16/16 (100%)
 Strand = Plus / Minus


Query: 34      tttgatgagattttta 49
                ||||||||||||||||
Sbjct: 636999 tttgatgagattttta 636984


 Score = 31.0 bits (15), Expect = 1.2
 Identities = 15/15 (100%)
 Strand = Plus / Minus


Query: 212      tcaacagacatttcc 226
                |||||||||||||||
Sbjct: 1041336 tcaacagacatttcc 1041322


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 14/14 (100%)
 Strand = Plus / Plus


Query: 54     aataaactcaaaat 67
               ||||||||||||||
Sbjct: 22740 aataaactcaaaat 22753


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 14/14 (100%)
 Strand = Plus / Minus


Query: 173     cataaaataattta 186
                ||||||||||||||
Sbjct: 766574 cataaaataattta 766561


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 14/14 (100%)
 Strand = Plus / Minus


Query: 159   ttttaaatatgtat 172
              ||||||||||||||
Sbjct: 72761 ttttaaatatgtat 72748


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 14/14 (100%)
 Strand = Plus / Minus


Query: 176     aaaataatttaata 189
                ||||||||||||||
Sbjct: 736136 aaaataatttaata 736123


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 14/14 (100%)
 Strand = Plus / Minus


Query: 34      tttgatgagatttt 47
                ||||||||||||||
```

```
Sbjct: 942314 tttgatgagatttt 942301


>ref|NC_001142.1| Saccharomyces cerevisiae chromosome X, complete chromosome sequence
          Length = 745440

 Score = 33.0 bits (16), Expect = 0.30
 Identities = 16/16 (100%)
 Strand = Plus / Minus


Query: 150    gaatatctgttttaaa 165
              ||||||||||||||||
Sbjct: 180402 gaatatctgttttaaa 180387


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 14/14 (100%)
 Strand = Plus / Minus


Query: 211    ttcaacagacattt 224
              ||||||||||||||
Sbjct: 582789 ttcaacagacattt 582776


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 17/18 (94%)
 Strand = Plus / Minus


Query: 158    gttttaaatatgtatcat 175
              |||||||||| ||||||
Sbjct: 396997 gttttaaatatttatcat 396980


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 14/14 (100%)
 Strand = Plus / Minus


Query: 119    agtttaataaagca 132
              ||||||||||||||
Sbjct: 669781 agtttaataaagca 669768


 Score = 29.0 bits (14), Expect = 5.0
 Identities = 14/14 (100%)
 Strand = Plus / Plus


Query: 54    aataaactcaaaat 67
              ||||||||||||||
Sbjct: 16983 aataaactcaaaat 16996


Database: E:\major_report\db_download\yeast.nt\yeast.nt
           17 sequences; 12,155,026 total letters


Database: E:\major_report\blast_download\pdbnt\pdbnt
           2787 sequences; 53,920 total letters

           Posted date:  Mar 14, 2001  8:54 PM

Lambda     K      H
    1.41    0.715     1.33

Gapped
Lambda     K      H
    1.41    0.715     1.33
```

```
Matrix: blastn matrix:1 -3

Gap Penalties: Existence: 5, Extension: 2

Number of Hits to DB: 2504
Number of Sequences: 17
Number of extensions: 2504
Number of successful extensions: 739
Number of sequences better than 10.0: 15

Length of query: 231
Length of database: 12155026
effective HSP length: 16
effective length of query: 215
effective length of database: 12154754
effective search space: -1681695186
effective search space used: 2613272110
T: 0

A: 0

X1: 5 (10.2 bits)

X2: 9 (18.3 bits)

S1: 12 (24.9 bits)

S2: 14 (29.0 bits)
```

6. Appendix II The list of the programs in an appendix.

Following is soure code of BlastMerge.

```
/*********************************************************/
/*                                                       */
/* File Name: blastApp.h                                 */
/* Class Name: AligmentRecord                            */
/*                                                       */
/* Author: Xiaoming Wang                                 */
/*                                                       */
/*********************************************************/

#include <string.h>
#include "Llist_1.h"


#define MAX_RECORDS 50

bool findStringFirst( const char*src,const char* substr);


class AligmentRecord
{
public:
```

```
        AligmentRecord();//constructor
        void Init(char* filePath);
        void Append(char* FilePath);
        void WriteAligmentRecord();
//private:
        char programName1[128];
        char programName2[128];
        char programDescription[1024];
      char queryDescription[256];
      char databaseName[128];
        char databaseName_append[128];
        char databasePostedDate[128];
        char searchingDone[128];
        // count on how many local aligments in database
        int currentAligmnetNumber;
        List* ADList;
        char postedDate[64];
        char Lambda_str[512];
      float Lambda;
        float K;
        float H;
        char matrix[64];
        char gapPenality[64];
        int numberOfHitsToDb;
        int numberOfSequences;
        int numberOfExtensions;
        int numberOfSucessfulExtensions;
        char numberOfSequencesBetterThan[64];
        int lengthOfQuery;
        int lengthOfDatabase;
        int effectiveHSPLength;
        int effectiveLengthOfQuery;
        int effectiveLengthOfDatabase;
        unsigned long effectiveSearchSpace;
        unsigned long  effectiveSearchSpaceUsed;
        char T[32];
        char A[32];
        char X1[64];
        char X2[64];
        char S1[64];
        char S2[64];


};


/*******************************************************/
/*                                                     */
/* File Name: blastApp.cpp                             */
/* Class Name: AligmentRecord                          */
/*                                                     */
/* Author: Xiaoming Wang                               */
/*                                                     */
/*******************************************************/
```

```
#include "blastApp.h"
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include "blast.h"

// global variable
bool programNameFlag = false;
bool programDescriptionFlag = false;
bool queryDescriptionFlag = false;
bool databaseNameFlag = false;
bool databaseName_appendFlag = true;
bool searchingDoneFlag = false;
int blastType = 0;


AligmentRecord::AligmentRecord()
{

        strcpy(programName1,"");
        strcpy(programName2,"");
        strcpy(programDescription,"");
        strcpy(queryDescription,"");
        strcpy(databaseName,"");
        strcpy(databaseName_append,"");
        strcpy(databasePostedDate,"");
        strcpy(searchingDone,"");
      currentAligmnetNumber = 0;

        ADList = new List(); // create empty list
        strcpy(postedDate,"");
        strcpy(Lambda_str,"");
      Lambda = 0.0;
      K = 0.0;
        H = 0.0;
        strcpy(matrix,"");
        strcpy(gapPenality,"");
        numberOfHitsToDb = 0;
        numberOfSequences = 0;
        numberOfExtensions = 0;
        numberOfSucessfulExtensions = 0;
        strcpy(numberOfSequencesBetterThan,"");
        lengthOfQuery = 0;
        lengthOfDatabase = 0;
        effectiveHSPLength = 0;
        effectiveLengthOfQuery = 0;
        effectiveLengthOfDatabase = 0;
        effectiveSearchSpace =0;
        effectiveSearchSpaceUsed = 0;
        strcpy(T,"");
        strcpy(A,"");
        strcpy(X1,"");
        strcpy(X2,"");
        strcpy(S1,"");
        strcpy(S2,"");
```

```
}

void AligmentRecord::Append(char* FilePath)
{
        char* buffer;
        FILE* fp;
        fp = fopen(FilePath,"r");
        if(!fp)
        {
                printf("can not open file:%s\n",FilePath);
                return;
        }
        buffer =  (char*)malloc(256);
        while(fgets(buffer,256,fp))
        {
                if (findStringFirst(buffer,"BLASTN"))
                {
                        blastType = BLASTN;
                        strcpy(programName2,buffer);
                }
                else if (findStringFirst(buffer,"BLASTP"))
                {
                        blastType = BLASTP;
                        strcpy(programName2,buffer);
                }
                else if(findStringFirst(buffer,"Database:"))
                {
                        strcpy(databaseName_append,buffer);
                        databaseName_appendFlag = false;
                }
                else if ( databaseName_appendFlag == false)
                {
                        strcat(databaseName_append, buffer);
                        databaseName_appendFlag = true;
                }
                else if(!findStringFirst(buffer,">"))
                        continue;
                else
                {
                                while(1)
                        {
                                ELEM elem = new AligmentDetail();
                                buffer = elem->getAligment(fp,buffer, blastType);
                                if(!ADList->search(elem))
                                        ADList->sort_insert(elem);
                                else
                                        delete elem;
                                if (strstr(buffer, "Database:") != NULL)
                                        break;
                        }
                }
        }
        fclose(fp);

}
```

```
void AligmentRecord::Init(char* FilePath)
{
        char* buffer;
        FILE* fp;
        fp = fopen(FilePath,"r");
        if(!fp)
        {
                printf("can not open file:%s\n",FilePath);
                return;
        }
        buffer =  (char*)malloc(256);
        while(fgets(buffer,256,fp) != NULL)
        {
                if (strcmp(buffer, "") == 0) continue;
                if (findStringFirst(buffer,"BLASTN"))
                {
                        blastType = BLASTN;
                        strcpy(programName1,buffer);
                }
                else if (findStringFirst(buffer,"BLASTP"))
                {
                        blastType = BLASTP;
                        strcpy(programName1,buffer);
                }
                else if (findStringFirst(buffer,"Reference:"))
                {
                        programNameFlag = true;
                    strcpy(programDescription,buffer);
                }
                else if (findStringFirst(buffer,"Query="))
                {
                        programDescriptionFlag = true;
                        strcpy(queryDescription,buffer);
                }
                else if(findStringFirst(buffer,"Database:"))
                {
                        queryDescriptionFlag = true;
                        strcpy(databaseName,buffer);
                }
                else if(findStringFirst(buffer,"Searching..."))
                {
                        databaseNameFlag = true;
                        strcpy(searchingDone,buffer);
                        strcat(searchingDone,"\n");
                }
                else if (strstr(buffer, "Score     E") != NULL)
                {
                    searchingDoneFlag =true;
                        continue;
                }
                else if(findStringFirst(buffer,"Sequences producing significant alignments:"))
                {
                        searchingDoneFlag =true;
                        continue;
                }
```

```
else if( strstr(buffer,"...") != NULL) // skip the line which contains pattern "..."
        continue;
else if (programNameFlag == false) // start copy
{

        strcat(programName1,buffer);
}
else if(findStringFirst(buffer,">"))
{

        while(1)
        {
                ELEM elem = new AligmentDetail();
                buffer = elem->getAligment(fp,buffer, blastType);
                if(!ADList->search(elem))
                        ADList->sort_insert(elem);
                else
                        delete elem;
                if (strstr(buffer, "Database:") != NULL)
                        break;
        }
}
else if (strstr(buffer,"Posted date:") != NULL)
{
        strcpy(postedDate,buffer);
}
else if ( (strstr(buffer,"Number of letters in database:") != NULL )||
    (strstr(buffer,"Number of sequences in database:") != NULL))
                continue;
else if (strstr(buffer,"Lambda    K      H") != NULL)
{
        strcpy(Lambda_str,buffer);
        fgets(buffer,256,fp);
        strcat(Lambda_str,buffer);
        fgets(buffer,256,fp);
        strcat(Lambda_str,buffer);
        fgets(buffer,256,fp);
        strcat(Lambda_str,buffer);
        fgets(buffer,256,fp);
        strcat(Lambda_str,buffer);
        fgets(buffer,256,fp);
        strcat(Lambda_str,buffer);
}
else if (strstr(buffer,"Matrix:") != NULL)
{
   strcpy(matrix,buffer);
}
else if (strstr(buffer,"Gap Penalties:") != NULL)
{
   strcpy(gapPenality,buffer);
}
else if (strstr(buffer,"Number of Hits to DB:") != NULL)
{
        for( unsigned int i = 0; i < strlen(buffer) && buffer[i] != ':'; i++);
        strcpy(buffer, buffer + i + 1);
```

```
        numberOfHitsToDb = atoi(buffer);
    }
    else if (strstr(buffer,"Number of Sequences:") != NULL)
    {
            for( unsigned int i = 0; i < strlen(buffer) && buffer[i] != ':'; i++);
            strcpy(buffer, buffer + i + 1);
        numberOfSequences = atoi(buffer);
    }
    else if (strstr(buffer,"Number of extensions:") != NULL)
    {
            for( unsigned int i = 0; i < strlen(buffer) && buffer[i] != ':'; i++);
            strcpy(buffer, buffer + i + 1);
        numberOfExtensions = atoi(buffer);
    }
    else if (strstr(buffer,"Number of successful extensions:") != NULL)
    {
            for( unsigned int i = 0; i < strlen(buffer) && buffer[i] != ':'; i++);
            strcpy(buffer, buffer + i + 1);
        numberOfSucessfulExtensions = atoi(buffer);
    }
    else if (strstr(buffer,"Number of sequences better than") != NULL)
    {
           strcpy(numberOfSequencesBetterThan,buffer);
    }
    else if (findStringFirst(buffer,"length of query:"))
    {
            for( unsigned int i = 0; i < strlen(buffer) && buffer[i] != ':'; i++);
            strcpy(buffer, buffer + i + 1);
        lengthOfQuery = atoi(buffer);
    }
    else if (findStringFirst(buffer,"length of database:"))
    {
            for( unsigned int i = 0; i < strlen(buffer) && buffer[i] != ':'; i++);
            strcpy(buffer, buffer + i + 1);
            char* temp = (char*)malloc(strlen(buffer));
            char *tempPtr;
            tempPtr = temp;
        while(*buffer !='\0')
            {
                    if(*buffer !=',')
                    {
                            *tempPtr = *buffer;
                            tempPtr++;
                    }
                    buffer++;
            }
            *tempPtr = '\0';
        lengthOfDatabase = atoi(temp);
            free(temp);
    }
    else if (strstr(buffer,"effective HSP length:") != NULL)
    {
            for( unsigned int i = 0; i < strlen(buffer) && buffer[i] != ':'; i++);
            strcpy(buffer, buffer + i + 1);
        effectiveHSPLength = atoi(buffer);
    }
```

```
else if (strstr(buffer,"effective length of query:") != NULL)
{
        for( unsigned int i = 0; i < strlen(buffer) && buffer[i] != ':'; i++);
        strcpy(buffer, buffer + i + 1);
    effectiveLengthOfQuery = atoi(buffer);
}
else if (strstr(buffer,"effective length of database:") != NULL)
{
        for( unsigned int i = 0; i < strlen(buffer) && buffer[i] != ':'; i++);
        strcpy(buffer, buffer + i + 1);
        char* temp = (char*)malloc(strlen(buffer));
        char* tempPtr;
        tempPtr = temp;
    while(*buffer != '\0')
        {
                if(*buffer !=',')
                {
                        *tempPtr = *buffer;
                        tempPtr++;
                }
                buffer++;
        }
        *tempPtr = '\0';
    effectiveLengthOfDatabase = atoi(temp);
        free(temp);
}
else if (strstr(buffer,"effective search space:") != NULL)
{
        for( unsigned int i = 0; i < strlen(buffer) && buffer[i] != ':'; i++);
        strcpy(buffer, buffer + i + 1);
    effectiveSearchSpace = atol(buffer);
}
else if (strstr(buffer,"effective search space used:") != NULL)
{
        for( unsigned int i = 0; i < strlen(buffer) && buffer[i] != ':'; i++);
        strcpy(buffer, buffer + i + 1);
    effectiveSearchSpaceUsed = atol(buffer);
}
else if (findStringFirst(buffer,"T:"))
{
        strcpy(T,buffer);
        fgets(buffer,256,fp);
        strcpy(A,buffer);
        fgets(buffer,256,fp);
        strcpy(X1,buffer);
        fgets(buffer,256,fp);
        strcpy(X2,buffer);
        fgets(buffer,256,fp);
        strcpy(S1,buffer);
        fgets(buffer,256,fp);
        strcpy(S2,buffer);
}
else if (programDescriptionFlag == false)
{
    strcat(programDescription,buffer);
}
```

```
                else if (queryDescriptionFlag == false)
                {
                        strcat(queryDescription,buffer);
                }
                else if (databaseNameFlag == false)
                {
                        strcat(databaseName,buffer);
                }

                else if (searchingDoneFlag == false)
                {
                        strcat(searchingDone,buffer);
                }
        }
        fclose(fp);

}

void AligmentRecord::WriteAligmentRecord()
{
        FILE* fpOut;
        fpOut = fopen("out.data","w");
       printf("%s\n",programName1);
        printf("%s\n",programName2);
        printf("%s\n",programDescription);
        printf("%s\n",queryDescription);
        printf("%s\n",databaseName);
        printf("%s\n",searchingDone);
        fprintf(fpOut,"%s\n",programName1);
        fprintf(fpOut,"%s\n",programName2);
        fprintf(fpOut,"%s\n",programDescription);
       fprintf(fpOut,"%s\n",queryDescription);
        fprintf(fpOut,"%s\n",databaseName);
        fprintf(fpOut,"%s\n",databaseName_append);
        fprintf(fpOut,"%s\n",searchingDone);
        ADList->display(fpOut);
        printf("\n%s",databaseName);
        printf("      %s\n",postedDate);
        printf("%s\n",Lambda_str);

      printf("%s",matrix);
        printf("%s",gapPenality);
        printf("Number of Hits to DB: %d\n", numberOfHitsToDb);
        printf("Number of Sequences: %d\n", numberOfSequences);
        printf("Number of extensions: %d\n",numberOfExtensions);
        printf("Number of successful extensions: %d\n",numberOfSucessfulExtensions);
        printf("%s",numberOfSequencesBetterThan);
        printf("Length of query: %d\n", lengthOfQuery);
        printf("Length of database: %d\n", lengthOfDatabase);
        printf("effective HSP length: %d\n", effectiveHSPLength);
        printf("effective length of query: %d\n", effectiveLengthOfQuery);
        printf("effective length of database: %d\n", effectiveLengthOfDatabase);
        printf("effective search space: %u\n", effectiveSearchSpace);
        printf("effective search space used: %u\n", effectiveSearchSpaceUsed);
        printf("%s", T);
        printf("%s",A);
```

```
            printf("%s",X1);
            printf("%s",X2);
            printf("%s",S1);
            printf("%s",S2);
            fprintf(fpOut,"%s\n",databaseName);
            fprintf(fpOut,"%s\n",databaseName_append);
            fprintf(fpOut,"      %s\n",postedDate);
            fprintf(fpOut,"%s\n",Lambda_str);
            fprintf(fpOut,"%s\n",matrix);
            fprintf(fpOut,"%s\n",gapPenality);
            fprintf(fpOut,"Number of Hits to DB: %d\n",numberOfHitsToDb);
            fprintf(fpOut,"Number of Sequences: %d\n",numberOfSequences);
            fprintf(fpOut,"Number of extensions: %d\n",numberOfExtensions);
            fprintf(fpOut,"Number of successful extensions: %d\n",numberOfSucessfulExtensions);
            fprintf(fpOut,"%s\n",numberOfSequencesBetterThan);
            fprintf(fpOut,"Length of query: %d\n",lengthOfQuery);
            fprintf(fpOut,"Length of database: %d\n",lengthOfDatabase);
            fprintf(fpOut,"effective HSP length: %d\n",effectiveHSPLength);
            fprintf(fpOut,"effective length of query: %d\n",effectiveLengthOfQuery);
            fprintf(fpOut,"effective length of database: %d\n",effectiveLengthOfDatabase);
            fprintf(fpOut,"effective search space: %d\n",effectiveSearchSpace);
            fprintf(fpOut,"effective search space used: %u\n",effectiveSearchSpaceUsed);
            fprintf(fpOut,"%s\n", T);
            fprintf(fpOut,"%s\n",A);
            fprintf(fpOut,"%s\n",X1);
            fprintf(fpOut,"%s\n",X2);
            fprintf(fpOut,"%s\n",S1);
            fprintf(fpOut,"%s\n",S2);
            fclose(fpOut);
            ADList->clear();
}

void main()
{
            char *filePath,*filePath1;
            filePath = (char*)malloc(40);
           filePath1 = (char*)malloc(40);
            char *name1, *name2;
            name1 = (char*)malloc(20);
            name2 = (char*)malloc(20);
            printf(" Please enter first file name:\n");
            scanf("%s",name1);
            printf("Please enter second file name:\n");
            scanf("%s",name2);
            sprintf(filePath,name1);
            sprintf(filePath1,name2);
          AligmentRecord* ald = new AligmentRecord();
            ald->Init(filePath);
            ald->Append(filePath1);
            ald->WriteAligmentRecord();
            free(filePath);
            free(filePath1);

}

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

```
/*                                                                    */
/* File name: Llist_1.h                                               */
/* Class Name: List                                                   */
/* Author:    Xiaoming Wang                                           */
/*                                                                    */
/**********************************************************************/

/**********************************************************************/
/*          Interface of class List                                   */
/**********************************************************************/

//------------------List.h--------------------

#ifndef _List_1_h
#define _List_1_h


#include"link.h"


class List {                //A singly_linked list class
private:
        link* head;             //Pointer to lidt header
        link* tail;
        link* curr;             //position of "current" ELEM
public:
        bool search(const ELEM& elem);
        void sort_insert(const ELEM& item);
        List(); //Constructor
        ~List();                //Destructor
        void clear();           //Remove all ELEMs from list
        void insert(const ELEM&);  //Insert ELEM at current position
        void append(const ELEM&);  //Insert ELEM AT tail of list
        ELEM remove();          //Remove and return current ELEM
       void first();            //Set curr to first position
        void next();            //Move curr to next position
       void prev();             //Move curr to previous position
        int length() const;       //Return current lenth fo list
        void setPos(const int);   //Set current to specified position
        void setValue(const ELEM&); //Set current ELEM's value
        ELEM currValue() const;   //Return current ELEM's value
        bool isEmpty() const;     //Return TRUE if list is empty
        bool isInList() const;    //TRUE if curr is within list
        bool find(const ELEM&);   //Find value (from current position)
        void display(FILE* fpOut) const;     //Dislpay all the element in list
        void add( const ELEM&);   //Add a new node to circular linked_list
};

#endif // End of class List.h


/**********************************************************************/
/*                                                                    */
/* File name: Llist_1.cpp                                             */
/* Class Name: List                                                   */
/* Author:    Xiaoming Wang                                           */
```

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



/**********************************************************************/
/*                Definition of Class List          */
/**********************************************************************/

//-------------------------Llist.cpp---------------------
//#include "stdafx.h"
#include <stdio.h>
#include <assert.h>
#include"Llist_1.h"
#include <string.h>
#include "blast.h"

// Definitions for List class -- doubly-linked List implementation.

List::List()      // Constructor -- ignore sz
  { tail = head = curr = new link; }

List::~List() {              // Destructor
  while(head != NULL) {       // Return link nodes to free store
    curr = head;
    head = head->next;
    delete curr;
  }
}

void List::clear() {          // Remove all ELEMs from List
  while (head->next != NULL) { // Return link nodes to free store
    curr = head->next;
    head->next = curr->next;
    delete curr;
  }
  curr = tail = head;
}

// Insert ELEM at current position
void List::insert(const ELEM& item) {
  assert(curr != NULL);
  curr->next = new link(item, curr->next, curr);
  if (curr->next->next != NULL)
    curr->next->next->prev = curr->next;
  if (tail == curr)  tail = curr->next;
}

void List::sort_insert(const ELEM& item)
{
                curr = head->next;
        if (curr)
        {
                while(curr->next && (curr->element->Expect[0] <= item->Expect[0]))
                                curr = curr->next;
```

```
                if (curr->element->Expect[0] > item->Expect[0])
                        curr = curr->prev;
                if(curr->next)
                        curr->next = new link(item,curr->next,curr);
                else
                        curr->next = new link(item,NULL,curr);
                if(curr->next->next != NULL)
                        curr->next->next->prev = curr->next;
        }
        else
        {
                 tail->next = new link(item, NULL, tail);
                tail = tail->next;
        }

}

void List::append(const ELEM& item) { // Append ELEM at tail of List
 tail->next = new link(item, NULL, tail);
 tail = tail->next;
}

ELEM List::remove() {           // Remove ELEM at current position
 assert(isInList());          // Must be valid position in List
 ELEM temp = curr->next->element;
 link* ltemp = curr->next;
 if (ltemp->next != NULL) ltemp->next->prev = curr;
 else tail = curr;            // Removed tail ELEM - change tail
 curr->next = ltemp->next;
 delete ltemp;
 return temp;
}

void List::prev()             // Move curr to previous position
 { if (curr != NULL)  curr = curr->prev; }

void List:: first()           // Set curr to first position
 { curr = head; }

void List::next()              // Move curr to next position
 { if (curr != NULL)  curr = curr->next; }

int List::length() const {        // Return current length of List
 int cnt = 0;
 for (link* temp = head->next; temp != NULL; temp = temp->next)
   cnt++;                     // Count the number of elements
 return cnt;
}

void List::setPos(const int pos) {  // Set curr to specified position
 curr = head;
 for(int i=0; (curr!=NULL) && (i<pos); i++)
  curr = curr->next;
}

void List::setValue(const ELEM& val) // Set current ELEM's value
```

```
  { assert(isInList()); curr->next->element = val; }

ELEM List::currValue() const        // Return value of current ELEM
  { assert(isInList()); return curr->next->element; }

bool List::isEmpty() const          // Return TRUE if List is empty
  { if (head->next == NULL)
      return false;
    else
    return  true;
  }

bool List::isInList() const         // TRUE if curr is within List
  { if( (curr != NULL) && (curr->next != NULL))
      return true;
      else
                              return false;
    }

bool List::find(const ELEM& val) {  // Find value (from current position)
  while (isInList())
    if (currValue() == val) return true;
    else next();
  return false;                     // Not found
}

void List::display(FILE* fpOut) const
{
          printf("                                                    Score    E  \n");
          printf("Sequences producing significant alignments:                (bits)  Value  \n");
         fprintf(fpOut,"                                                Score    E  \n");
          fprintf(fpOut,"Sequences producing significant alignments:                (bits)  Value  \n\n");

          link* temp;
          temp = head;
          while(temp->next != NULL)
          {
                    temp->next->element->writeAligmentShort(fpOut);
                    temp = temp->next;
          }

          temp = head;
          while(temp->next != NULL)
          {
                    temp->next->element->writeAligment(fpOut);
                    temp = temp->next;
          }
}

bool List::search(const ELEM &elem)
{
  link* temp;
          temp = head->next;
          while( temp)
          {
                    if(strcmp(temp->element->matchRecordName,elem->matchRecordName) == 0)
```

```
                    return true;
                temp = temp->next;
        }
        return false;
}

/*********************************************************/
/*                                                       */
/* File Name: link.h                                     */
/* Class Name: AlignmentDetail link                      */
/*                                                       */
/* Author: Xiaoming Wang                                 */
/*                                                       */
/*********************************************************/

#ifndef _link_h
#define _link_h
# include <stdlib.h>
# include <stdio.h>
# include <iostream.h>
# include <fstream.h>

#include<assert.h>
#include<iostream.h>

#define MAX_SCORE_NUM 20
bool findStringFirst(char* src, char* substr);


/*********************************************************************/
/*          Interface File of Class AligmentDetail       */
/*********************************************************************/


class AligmentDetail
{
public:
  AligmentDetail();
        char* getAligment(FILE* fp,char* buffer, int blastType);
        void writeAligment(FILE* fpOut);
        void writeAligmentShort(FILE* fpOut);
        char* matchRecordName;
        bool matchRecordNameFlag;
        int currentScoreNumber;
        int aligmentLength;
        int blastAligmentDetailType;
        double Score[MAX_SCORE_NUM];
        double secondScore[MAX_SCORE_NUM];
        double Expect[MAX_SCORE_NUM];
        char* Identity[MAX_SCORE_NUM];
        char* Strand[MAX_SCORE_NUM];
        char* Query[MAX_SCORE_NUM];
        bool QueryFlag;

        char* Sbjct[MAX_SCORE_NUM];
};
```

```
typedef AligmentDetail* ELEM;


const int LIST_SIZE=12;

/*****************************************************************/
/*                Interface File of Class link        */
/*****************************************************************/


class link {                  // A doubly-linked list node
public:                      //  with freelist
  ELEM element;                  // ELEM value for this node
  link* next;                 // Pointer to next node in list
  link* prev;                 // Pointer to previous node
  static link* freeList;          // Link class freelist
  link(const ELEM& elemval, link* nextp =NULL, link* prevp =NULL)
    { element = elemval;  next = nextp;  prev = prevp;}
  link(link* nextp =NULL, link* prevp = NULL)
    { next = nextp;  prev = prevp; }
  ~link() { }                   // Destructor: take no action
  void* operator new(size_t);      // Overloaded new operator
  void operator delete(void*);     // Overloaded delete operator
};

#endif //end of class link


/*******************************************************/
/*                                                     */
/* File Name: link.cpp                                 */
/* Class Name: AlignmentDetail link                    */
/*                                                     */
/* Author: Xiaoming Wang                               */
/*                                                     */
/*******************************************************/



/*****************************************************************/
/*                Definition of Class link          */
/*****************************************************************/

//#include "stdafx.h"
#include <string.h>
#include "link.h"
#include "blast.h"
bool findStringFirst(char* src, char* substr)
{
   char *p1,*p2;
         p1 = src;
         p2 = substr;
         int n = strlen(p2);
         while ( n != 0)
         {
```

```
                if ( *p1 != *p2 )
                        return false;
                p1++;
                p2++;
                n--;
        }
        return true;
}

//This creates space for the freelist variable
link* link::freeList = NULL;

void* link::operator new(size_t) { // Overload new operator
 if (freeList == NULL) return ::new link;  // Create new space
 link* temp = freeList;          // Otherwise, get from freeList
 freeList = freeList->next;
 return temp;                    // Return the link node
}

void link::operator delete(void* ptr) { // Overload delete operator
 ((link*)ptr)->next = freeList;       // Put on freeList
 freeList = (link*)ptr;
}

/*****************************************************************/
/*       Definition of Class  AligmentDetail          */
/*****************************************************************/


AligmentDetail::AligmentDetail()
{
   matchRecordName = (char*)malloc(512);
        strcpy(matchRecordName,"");
        matchRecordNameFlag = false;
        blastAligmentDetailType = 0;
   aligmentLength = 0;
        currentScoreNumber = 0;
        for (int i = 0; i < MAX_SCORE_NUM; i++)
        {
                Score[i] = 0.0;
                secondScore[i] = 0.0;
                Expect[i] =0.0;
                Identity[i] = (char*)malloc(32);
                strcpy(Identity[i],"");
                Strand[i] = (char*)malloc(32);
                strcpy(Strand[i],"");
                Query[i] = (char*)malloc(128);
                strcpy(Query[i],"");
                Sbjct[i] = (char*)malloc(128);
        strcpy(Sbjct[i],"");
        }
        QueryFlag = false;

}

char* AligmentDetail::getAligment(FILE* fp, char* buffer, int blastType)
```

```
{
        char *tok, *stopString, *tempStr;
        strcpy(matchRecordName,buffer);
        if(blastType == BLASTN)
                blastAligmentDetailType = BLASTN;
        else if(blastType == BLASTP)
                blastAligmentDetailType = BLASTP;
        while(fgets(buffer,256,fp))
        {
                if(strcmp(buffer,"") == 0)
                        continue;
                else if( findStringFirst(buffer,">"))
                {
                        return buffer;
                }
    else if (strstr(buffer,"Database:") !=NULL)
                {
                        return buffer;
                }
                else if (strstr(buffer,"Length =") != NULL)
                {
                        matchRecordNameFlag = true;
                        for( unsigned int i = 0; i < strlen(buffer) && buffer[i] != '='; i++);
                        strcpy(buffer, buffer + i + 1);
                        aligmentLength = atoi(buffer);
                }
                else if (strstr(buffer,"Score =") != NULL)
                {

                        tok = strtok(buffer,",");
                    for( unsigned int i = 0; i < strlen(tok) && tok[i] != '='; i++);
                        strcpy(tok, tok + i + 1);
                        Score[currentScoreNumber] = strtod(tok,&stopString);
                        tempStr = stopString;
                        for(i = 0; i < strlen(tempStr) && tempStr[i] != '('; i++);
                    strcpy(tempStr, tempStr + i + 1);
                        secondScore[currentScoreNumber] = strtod(tempStr, &stopString);
                        tok = strtok(NULL,",");
                    for( i = 0; i < strlen(tok) && tok[i] != '='; i++);
                        strcpy(tok, tok + i + 1);
                        Expect[currentScoreNumber] = atof(tok);
                }
                else if (strstr(buffer,"Identities =") != NULL)
                {
                        for( unsigned int i = 0; i < strlen(buffer) && buffer[i] != '='; i++);
                        strcpy(buffer, buffer + i + 1);
                        strcpy(Identity[currentScoreNumber],buffer);
                }
                else if (strstr(buffer,"Strand =") != NULL)
                {
                        for( unsigned int i = 0; i < strlen(buffer) && buffer[i] != '='; i++);
                        strcpy(buffer, buffer + i + 1);
                        strcpy(Strand[currentScoreNumber],buffer);
                }
                else if (findStringFirst(buffer,"Query:"))
                {
```

```
                        QueryFlag = false;
                        sprintf(Query[currentScoreNumber],"\n%s",buffer);
                }
                else if (strstr(buffer,"Sbjct:"))
                {
                        QueryFlag = true;
                        strcpy(Sbjct[currentScoreNumber],buffer);
                        currentScoreNumber++;
                }
                else if ( matchRecordNameFlag == false)
                {
                        strcat(matchRecordName,buffer);
                }
                else if ( QueryFlag == false)
                {
                        strcat(Query[currentScoreNumber],buffer);
                }
        }
        return buffer;

}


void AligmentDetail::writeAligment(FILE* fpOut)
{
        printf("    \n\n");
        printf("\n\n%s\n", matchRecordName);
        printf("          Length = %d\n", aligmentLength);
        fprintf(fpOut,"\n%s",matchRecordName);
        fprintf(fpOut,"          Length = %d\n\n",aligmentLength);
        for(int i = 0; i < currentScoreNumber; i++)
        {
                if ( (blastAligmentDetailType == BLASTP && i == 0) || (blastAligmentDetailType ==
BLASTN) )
                {
                        printf("Score = %f bits (%d), Expect = %2e\n",
                        Score[i],(int)secondScore[i],Expect[i]);
                        printf("Identities = %s\n", Identity[i]);
                        if(blastAligmentDetailType == BLASTN)
                        printf("Strand = %s\n", Strand[i]);
                }
                printf("\n%s",  Query[i]);
                printf("%s\n", Sbjct[i]);


        if ( (blastAligmentDetailType == BLASTP && i == 0) || (blastAligmentDetailType == BLASTN)
)
        {
                if((int)(Expect[i]*1000))
                        fprintf(fpOut,"\nScore = %.1f bits (%d) Expect =
%.3f\n",Score[i],(int)secondScore[i],Expect[i]);
                else
                        fprintf(fpOut,"\nScore = %.1f bits (%d) Expect =
%.0e\n",Score[i],(int)secondScore[i],Expect[i]);
                fprintf(fpOut,"Identities = %s",Identity[i]);
                if(blastAligmentDetailType == BLASTN)
```

```
                        fprintf(fpOut,"Strand = %s\n",Strand[i]);
        }
        fprintf(fpOut,"\n%s",Query[i]);
        fprintf(fpOut,"%s",Sbjct[i]);
        }

}


void AligmentDetail::writeAligmentShort(FILE* fpOut)
{
  char * temp, *temp2;
        int len = strlen(matchRecordName);
        temp = matchRecordName + 1;
        temp2 = (char*)malloc(65);
  if(len > 64)
        {
                temp2 = strncpy(temp2,temp,64);
                temp2[64] = '\0';
        }
        else
        {
                strcpy(temp2,temp);
                temp2[len - 1]='\0';
                temp2[len-2]= '\0';

        }

        printf("%s...  %d  %1e\n",temp2,(Score[0] + 0.5),Expect[0]);
  if((int)(Expect[0]*1000))
        {
                fprintf(fpOut,"%s",temp2);
                if(strlen(temp2) < 64)
                {
                        for ( unsigned int k = 0; k < 64 - strlen(temp2); k++)
                                fprintf(fpOut," ");
                }
                fprintf(fpOut,"...  %d  %.3f\n",(int)(Score[0] + 0.5),Expect[0]);
        }
        else
        {
                fprintf(fpOut,"%s",temp2);
                if(strlen(temp2) < 64)
                {
                        for (unsigned int k = 0; k < 64 - strlen(temp2); k++)
                                fprintf(fpOut," ");
                }
                fprintf(fpOut,"...  %d  %.0e\n",(int)(Score[0] + 0.5),Expect[0]);
        }
  free(temp2);

}
```