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. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

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.

UMI®

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

NOTE TO USERS

This reproduction is the best copy available

UMI

REVERSE ENGINEERING AND OPTIMISATION OF THE BLASTP PROGRAM

Shawn Delaney

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

September 1998

© Shawn Delaney, 1998

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-39484-0

Canada

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Shawn Delaney**

Entitled: **Reverse Engineering and Optimisation of the BLASTP Program**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

_____ Chair

_____ Examiner

_____ Examiner

_____ Supervisor

Approved _____

Chair of Department or Graduate Program Director

_____ 19 _____ _____

Dr. Nabil Esmail, Dean

Faculty of Engineering and Computer Science

Abstract

Reverse Engineering and Optimisation of the BLASTP Program

Shawn Delaney

The BLASTP (Basic Local Alignment Search Tool for Proteins) is a popular protein database search program. The BLASTP algorithm consists of three steps: (1) Constructing the set of neighbourhood words, (2) Scanning a database sequence for word hits, and (3) Extending a word hit. By using a reverse engineering approach, the architecture of the program that implements the algorithm is obtained and described. It was found that approximately 90% of the program's execution time is spent doing the third step of the algorithm, extending word hits. This work describes three optimisations to the algorithm that significantly decrease the program's execution time. The optimisations are of two types: (1) Two new sequence representations that facilitate the extension step and are only used in the extension procedure and (2) A scanning procedure that restricts the number of invocations of the extension procedure. The first optimisation represents the query as a sequence of memory addresses. These addresses are those of the rows of the residue pair score matrix which correspond to a particular residue. This representation reduces the number of instructions executed per matrix access. The second optimisation represents the query and database sequence as a sequence of residue-doublets. A doublet consists of two adjacent residues. Extensions are done in steps of aligned residue-doublet pairs. The third optimisation counts the number of word hits per aligned segments of the query and database sequence. The extension procedure is invoked only if the number of hits per alignment meets a threshold criteria. Each optimised algorithm is implemented in the BLASTP version 1.4 program. A comparison of performance data between each of the optimised programs and the unmodified program shows a performance increase of 15%, 48% and 63% respectively.

Acknowledgments

No one person has participated more in my professional growth than my friend, mentor and colleague, Greg Butler. I thank him not only for his contribution to this work, but also for our many conversations covering all aspects of computer science and software engineering. His tutelage and support have been overwhelming and his influence will be with me always.

I would like to thank Clement Lam and Larry Thiel. Their knowledge and expertise aided me greatly in the development of this work. I thank Rajjan Shinghal for his suggestions and his participation on my thesis committee.

Much of this work was influenced by the feedback obtained from Reg Storms and Adrian Tsang from the Department of Biology as well as from Paul Joyce from the Department of Chemistry and Biochemistry. I thank them for their contribution.

Finally, I would like to thank the analyst pool in the Department of Computer Science, in particular Stan Swiercz, Paul Gill and Michael Assels, whose excellent support too often goes unrecognised.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation For Thesis

The BLAST (**B**asic **L**ocal **A**lignment **S**earch **T**ool) suite of programs [1] is arguably the best tool currently available for searching molecular sequence databases. BLASTP is the particular program that looks for similarities between a query protein sequence and those in a protein sequence database. Search speed is a critical issue in scanning sequence databases whose sizes continue to grow as the sequence data from large-scale genome sequencing efforts is produced [2]. As the size of the search space increases, the performance of search programs such as BLAST decreases. There are two solutions to this problem: (1) Develop faster algorithms for scanning sequence databases and (2) Reduce the search space by grouping similar database sequences. This thesis addresses the problem of similarity search performance using the former approach. In this work, the more specific problem of protein database search performance is addressed by optimising the existing BLASTP version 1.4 program with an acceptable compromise in search sensitivity.

A faster BLASTP program would benefit scientists in several ways - two examples are provided: (1) Searching with a protein database scanning tool such as BLASTP is one of several steps performed in the analysis of genome sequence data (Figure 1). A faster program reduces the likelihood that analysis with BLASTP is a rate-limiting step in the work-flow and (2) BLAST servers, such as the one at NCBI [3], would require less time to perform BLASTP searches providing better service to clients.

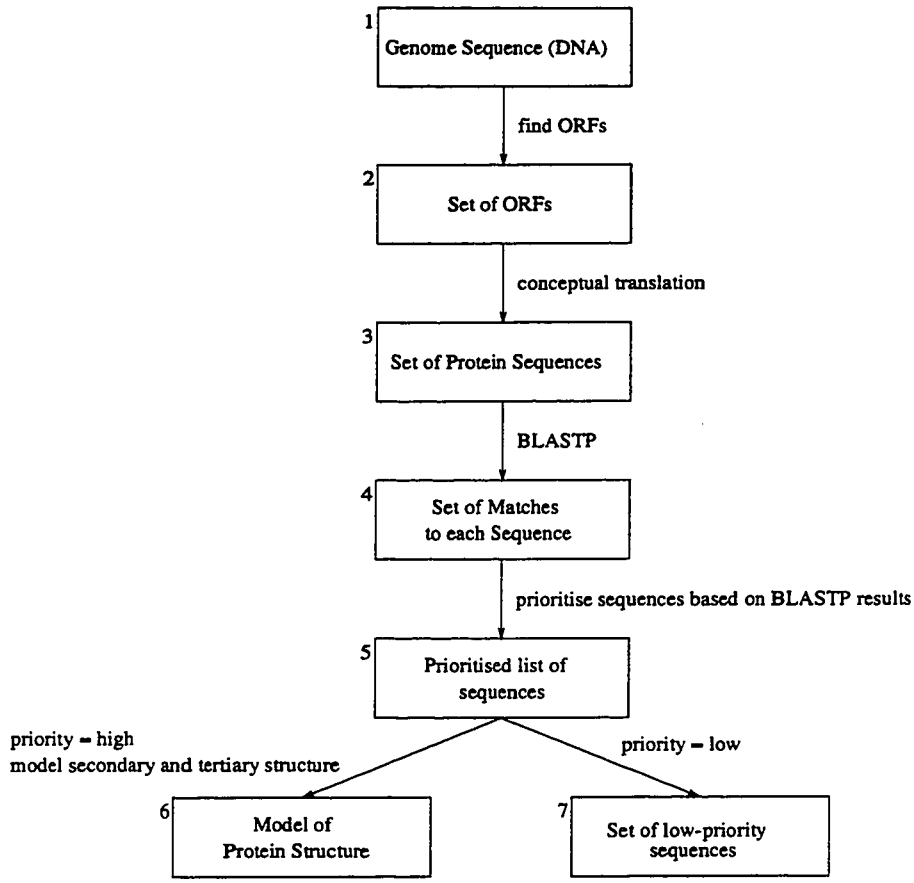

Figure 1: *Role of BLASTP in Genome Analysis Work-flow* A protein sequence comparison program such as BLASTP can be used as a filter to mark sequences for further investigation. In this example work-flow, the genome sequence (*1*) is scanned to identify protein coding regions or ORFs (open reading frames) (*2*). Each ORF is conceptually translated into three protein sequences each of which is used as a query in a BLASTP search (*3*). The results of the searches (*4*) are a criteria for prioritising the sequences (*5*). Here, the definition of "high and low priority sequence" is subjective to the investigator but may, for example, be a sequence that has no database matches (i.e a novel sequence) or one that matches a database sequence that is important to the investigator, for example, a match to a known drug target. Prioritising the sequences is necessary since further analysis may involve building secondary or tertiary structure models of the sequence using resource-intensive programs (*6*). Low priority sequences are set aside for later investigation (*7*).

1.2 The BLASTP Program

BLASTP is an appropriate program for protein database scanning for several reasons:

- The BLASTP algorithm is designed for fast database scanning. The algorithm is heuristic in nature, allowing the user to modify the search sensitivity by assigning values to various parameters; mainly W, T and X. These are fully described in section 2.3.
- The program produces local alignments. Proteins are more similar in sub-regions rather than over their entire lengths. Searching for local alignments increases the likelihood of finding similarities between an unknown protein and one that is distantly related in an evolutionary context.
- The program contains a method for assessing similarity scores. This reduces the likelihood of interpreting a match as significant when it it is not.
- Given a particular scoring scheme, in most cases, the algorithm is sensitive enough to find all similarities that a highly sensitive, but more time expensive, dynamic programming algorithm would detect.

The program performs two tasks: (1) It scans the protein sequence database with an input query sequence and compiles a list of *HSPs* (high scoring segment pairs and (2) analyses the *HSP* list in order to assign statistical significance to those matches. The minimum program input is a query and a database. The BLASTP algorithm generates the *HSP* list which is post-processed to generate an output list in which each *HSP* is assigned a measure of statistical significance.

1.2.0.1 Program Input

The program requires two basic inputs; the name of the file which contains the protein query sequence in FASTA format (Box 1) and the name of the protein sequence database. These are specified on the command line:

blastp `[database_file] [query_file]`

The FASTA format consists of a sequence descriptor and a sequence of characters

that represent the protein. The descriptor is prefixed by a '>' character. It contains the sequence identifier(s) - the sequence may be in several databases - and a description which includes the name of the sequence and possibly a short description of its biological function. The sequence database, also in FASTA format, must be processed by the `setdb` program prior to searching with BLASTP. This is described in Figure 12.

```
Box 1
gi|129937| sp|P27644|PGLR_AGRTU POLYGALACTURONASE (PECTINASE) (PGL)
gi|95113| pir||A40364 picA protein - Agrobacterium tumefaciens gi|142256 (M62814) PGL
ORF [Agrobacterium tumefaciens]
>MALATRATGGAGRRKPVRARCARGLHLVSCHKTQLLGFTIRNAASWTIHPQGCEDL
TAAASTIIAPHDSPNTDGFNPESCRNVMISGVRFSVGDDCIAVKAGKRGPDGEDDH
LAETRGITVRHCLMQPGHGGLVIGSEMSGGVHDVTVEDCDMIGTDRGLRLKTGARS
GGGMVGNITMRRVLLDGVQTALSANAHYHCDADGHDDWVQSRNPAPVNDGTPFVDG
ITVEDVEIRNLAHAAGVFLGLPDVPSATSLSATSPIVSHDPSAVATPPIMADRVRP
MRMRLVFEQADVVCDDPALLNDAPVSISSYFD
```

The program has several command line options which are listed in [4]. Those options that a typical user may specify are: (1) `-matrix`, which specifies the residue pair score matrix, and (2) W, T or X, which may be adjusted to control the sensitivity of the search. The default matrix used is `BLOSUM62`. With the `-matrix` option, the user may specify the name of a file containing an alternative or user-defined matrix:

```
blastp [database_file] [query_file] -matrix [score-matrix_file]
```

Controlling Search Sensitivity

The program parameters, W, T and X may be adjusted to control the sensitivity of the search:

```
blastp [database_file] [query_file] W=[ ] T=[ ] X=[ ]
```

The sensitivity can be increased, i.e more *HSPS* can be detected, by (1) lowering the neighborhood word score threshold, T, while keeping the word size, W, constant, (2) by lowering both W and T appropriately and/or (3) by raising the word hit extension falloff score X. These parameters are fully explained in section 2.3. For a more complete discussion of all possible input parameters see [4].

1.2.0.2 Program Output

The output of the program consists of five parts: (1) Program introduction, (2) Histogram of expectations if one is requested, (3) List of one-line summaries for each matching database sequence, (4) List of alignments or *HSPs* (high scoring segment pairs) and (5) Parameters used and search statistics. Parts three and four are of general interest to users and are further described here. For a description of the remaining parts see [4].

The one-line summary list (Figure 2) facilitates the comparison of the scores and statistical significance of individual matches to that of the set. The first column, `Sequences Producing High Scoring Segment Pairs` is the sequence descriptor from the FASTA format (see Box 1), and contains the sequence identifier and name. The second column `High Score` contains the score of the highest scoring *HSP* - the *MSP* or maximal segment pair. The query may have more than one *HSP* with a subject, but only the highest scoring one is reported in the one-line summary. The third column `Smallest Sum Probability P(N)` contains the lowest `P-value` ascribed to any set of *HSPs*. The fourth column `Smallest Sum Probability (N)` displays the number of *HSPs* in the set which was ascribed the lowest `P-value`. Essentially, the `P-value` is the probability that this *HSP* could occur by chance alone. The greater the number of *HSPs* between two sequences, the lower is this probability. If not otherwise specified, the list of one-line summaries is sorted by increasing `P-value`.

Sequences producing High-scoring Segment Pairs	High Score	Smallest Sum Probability P(N)	N
sp|P27644|pglr|agrtu polygacturonase (pectinase) (pgl)...	1649	6.1e-226	1
gi|1575707 (U70481) abscission polygalacturonas...	117	1.1e-15	2
gi|1575705 (U70480) abscission polygalacturonas...	112	1.6e-14	2
pir||S57806 polygalacturonase precursor - tomato...	112	2.2e-14	2
gi|479088 (X77231) polygalacturonase [Prunus p...	142	5.5e-12	3

Figure 2: *Example BLASTP Output - One Line Summary*

The set of *HSPs* is listed for each matching database sequence. (Figure 3) shows

an example of the information listed for each *HSP*. The sequence descriptor for the matching sequence is given followed by each *HSP* that was found between this sequence and the query. A description of an *HSP* consists of a statistical summary and the alignment.

The statistical summary contains: (1) the alignment `Score`, (2) the number of times one might `Expect` to see an equivalent or better match by chance, (3) the `P-value` or probability (in the range 0-1) of observing such a match, (4) the number and percentage of total residues in the alignment which are identical and (5) the number and percentage of residue pairs for which the score is positive.

Below the statistical summary is the *HSP*; the alignment of the query segment with the subject segment. The offsets of the *HSPs* are placed at the beginning and end of the query and subject segments. In between, letters indicate exact matches while + indicates a non-identical, but positive scoring match. No symbol indicates a zero or negative score for that residue pair.

```
>gil1575707 (U70481) abscission polygalacturonase [Lycopersicon esculentum]
          Length = 387

   Score = 117 , Expect = 1.1e-15, Sum P(2) = 1.1e-15
   Identities = 22/66 (33%), Positives = 35/66 (53%)


Query:   37  GFTI RNAASWTI HPQGCEDLTAAASTI I APHDSPNTDGFNPESCRNVMI SGVRFSVGDDC  96
             G T++N+    + I     GC +          +++P +SPNTDG  + +S     V  I           GDDC
Sbjct:  156  GVTVQNSQMFHI LVDGCHNAM IQGVKVLSPGNSPNTDGI HVQSSSGVS IMNSNI GTGDDC  215

Query:   97  I AVKAG   102
             I ++  G
Sbjct:  216  I SI GPG   221
```

Figure 3: *Example BLASTP Output - HSP List*

1.3 Summary of Thesis

This section provides a summary of the thesis contents. In essence, this work involved reverse engineering the BLASTP program for the sole purpose of optimising its performance. First, a brief description of the BLASTP algorithm is given followed by a summary of the reverse engineering methodology and the results obtained from that process. Next, a summary of the three optimised algorithms is presented along with their performance evaluations. Finally, this section outlines the contributions made by this thesis to the fields of bioinformatics and software engineering.

The BLASTP Algorithm

The algorithm works in three steps. First, the neighborhood is constructed. The neighborhood consists of the set of W-mers that score at least T with some W-mer in the query. The second step scans the database for matches or hits to a neighborhood word. The third step attempts to extend a hit into an *HSP* before the falloff value of X is reached. Extensions start from the hit and proceed in steps of aligned residue pairs in both directions. At each step the score of the aligned pair is looked-up in a score matrix and added to a running sum.

Reverse Engineering of the BLASTP Program

This work uses a reverse engineering approach to identify those parts of the program that, if optimised, would give the largest CPU speed-up. This is an application of Amdahl's Law [5] - the largest speed-up is obtained by optimising those parts that use the most CPU cycles. The reverse engineering method consists of 1) obtaining an execution profile of the program which provides a breakdown of execution time according to procedures and lines of code and 2) constructing a program model which consists of a set of sub-architectures each of which implements a particular functionality and accounts for a relatively large fraction of the overall execution time. The execution profile indicates where optimisations should be made while the program model shows the context and extent of any modification. The execution profile of the BLASTP program under the conditions described in 3.2.1 shows that over 90% of the CPU time is spent in the procedure that implements the third step of the algorithm, extending word hits. In addition, within that procedure, three do-while loops, which localise the starting points of extension and perform the left and right extensions

respectively, account for more than 76% of the overall execution time. Each loop contains either one or two lines of code that retrieve a score from the matrix. These lines alone account for more than 63% of the overall time. The focus for optimisation is clear. Any modified program that more efficiently accesses the matrix, executes fewer extension loops or invokes the extension procedure less frequently will provide a substantial speed-up. Reverse engineering the program provides a model of the program's design. Such a model is necessary to show where additional code should be added and what effect that modification might have on the rest of the program.

Optimisations to the BLASTP Algorithm

This thesis proposes three optimisations that are of two types: 1) New sequence representations that are used explicitly in the extension procedure and 2) A constraint on the number of times the extension procedure is invoked. The first optimisation uses a score matrix row address representation of the query sequence which reduces the number of instructions required to access the matrix. The second optimisation uses a sequence representation for both the query and subject sequences that groups residues into pairs or residue-doublets; each residue-doublet in the amino acid alphabet is assigned an integer. This effectively halves the lengths of the sequences, allowing extensions to be done in approximately half the time since the number of extension loops executed is greatly reduced. The third optimisation, rather than modify the extension procedure, invokes the procedure less frequently. The second step of the algorithm, scanning for hits, is modified so that the extension procedure is called only when two hits are found within a given distance. For the first and second optimisations, termed row-address and residue-doublet respectively, new extension procedures were coded and and plugged into the BLASTP version 1.4 program. For the third or two-hit optimisation, the scanning procedure was augmented with code that counts the number of hits per aligned segment of the query and subject sequences. Compared to the unmodified program, those implementing the three optimised algorithms show speed-ups of 15%, 48% and 63% respectively. In addition, the effect of each of the optimisations on the detection of *HSPs* was studied. The row-address optimisation is in fact more of a change to the implementation of the algorithm rather than a change to the algorithm's heuristics. Thus, the matrix-row algorithm finds all *HSPs* found by the unmodified algorithm. However the residue-doublet and two-hit

optimisations change the heuristics of the algorithm and do in fact miss some lower-scoring *HSPs*. As a consequence of grouping residues into pairs, the residue-doublet algorithm misses some *HSPs* with $S < 50$. The two-hit algorithm misses those *HSPs* that contain a single hit since it only attempts to extend those hits that are within a given distance of a previous hit. Any *HSP* detected by the two-hit algorithm will always contain at least two hits. The program implementing the two-hit algorithm reports all *HSPs* reported by the unmodified program. The only difference is that the *P-values* of some of the *HSPs* reported by the modified program are slightly lower.

Contributions

This work makes three valuable contributions. First, each of the optimised BLASTP algorithms provides a significant speed-up of the program that enables scientists to obtain results of BLASTP searches much faster with an acceptable compromise in search sensitivity. Second, the reverse engineering of the program resulted in a parameterised description of the algorithm that clearly shows how that algorithm works. Such a description was not previously available in the literature. Finally, this work is a case study in reverse engineering in which the design of a program was extracted from the source code and modelled with the aid of execution profilers.

1.4 Organisation of Thesis

There are four additional chapters. Chapter 2 describes the role of a database search tool such as BLASTP in the experimental process of identifying homologous sequences. In addition, this chapter discusses four important issues in protein sequence comparison: (1) Score Methods (2) Sequence Databases (3) Sequence Comparison Algorithms and (4) Assessment of Results. Finally, the chapter provides a parameterised description of the BLASTP algorithm.

Chapter 3 describes the reverse engineering of the BLASTP program. It presents the reverse engineering methodology which includes descriptions of (1) the test search, (2) how execution profilers were used as reverse engineering tools, and (3) how particular sub-architectures of the program were modelled. Finally, the chapter presents the profile results and the program models obtained from revere engineering the program.

Chapter 4 describes the three optimisations to the BLASTP algorithm. Each optimisation is presented in four parts: (1) Concept - explains the idea behind the optimisation and why its implementation would result in a performance speed-up, (2) Design and Implementation - describes how the optimisation was implemented in the BLASTP version 1.4 program, (3) Results - compares the performance of the optimised program to that of the unmodified one and (4) Effect on *HSP* Detection - describes the cost in search sensitivity by comparing the *HSPs* detected using the optimised algorithm to those of the unmodified one.

Chapter 5 summarises the performance gains obtained from each of the three algorithm optimisations. In addition, a discussion of whether or not the cost in search sensitivity of using the optimised algorithms is acceptable. Finally, an outline of the contributions of this work to the fields of software engineering and bioinformatics is provided.

Chapter 2

Topics in Protein Similarity Searching

2.1 Biology Primer - Protein Structure, Function and Homology

Proteins are biological macromolecules that are comprised of amino acids of which there are twenty. Proteins differ in their amino acid composition. The simplest view of protein structure is that of a linear sequence of amino acids, referred to as primary structure. However, the amino acids that comprise a protein interact to produce more complex structures. Secondary structure refers to the spatial arrangement of amino acids that are near one another in the linear sequence. Some of these structures contain repeating sub-structures such as α-helices and β-sheets. Tertiary structure refers to the spatial arrangement of amino acid residues that are far apart in the linear sequence; it is the three-dimensional conformation of the protein over its entire length that includes regions of secondary structure. Another level of structure is quaternary structure which refers to the spatial arrangement of two or more interacting proteins.

The tertiary structure of a protein implies the protein's functionality. It is the interaction among the protein's amino acids both locally and globally that provides the structural integrity required for a protein to perform its biological function. "The information needed to specify the complex three-dimensional structure of a protein is contained in its amino acid sequence - sequence specifies conformation" [6], page 33.

Two proteins are homologous if they share a common ancestor [7]; i.e. they are related in an evolutionary context. Homologous proteins always share a common three-dimensional folding structure and often share active sites or binding domains [8].

2.2 Protein Sequence Comparison

2.2.1 Inferring Homology

Comparing one protein to another is one of the most informative computational activities in biology. Sequence comparison is used as a method to infer homology and is most informative when it detects homologous proteins. Information about the secondary and tertiary structure of a protein for which only the primary sequence is known can be inferred by finding a homologue for which this information is known. Similarity is a measured quantity while homology is inferred. Two sequences whose primary structures are similar within a certain level of confidence can be inferred to be homologous. This inference can be later reinforced by comparison techniques that go beyond primary structure and provide measures of similarity based on secondary and tertiary structure.

The process of inferring homology consists of several sequential steps each of which builds on facts gathered in the previous one to build a case for homology [9] [10]. First an unknown protein is compared to a sequence database to find the set of known sequences to which it is similar. The BLASTP [1] or FASTA [11] programs are appropriate tools for this step since they perform searches quickly with an acceptable compromise in sensitivity. In addition, both programs provide a statistical measure of significance for each match that contributes greatly to the assessment of the biological significance of a match. Matches that have a borderline score or ambiguous significance can be compared again using more rigorous, but computationally expensive, programs that implement dynamic programming algorithms. Subsequently, possible homologues identified by these techniques can be compared to structural databases such as HSSP [12] or FSSP [13] which may give further evidence of homology based on similarity of secondary or tertiary structure.

The context of this work is the first step of this process in which an unknown is compared to a sequence database. The issues in this step are: (1) choosing a score method (2) choosing a sequence database to search (3) choosing a search algorithm and (4) assessing the significance of matches. The following subsections provide a brief introduction to each of these issues.

2.2.2 Score Matrices

One of the most frequently used matrices is that of Dayhoff [14] who counted the number of point-accepted mutations (PAMs) in 71 groups of closely related proteins(85% similarity). An accepted point mutation is defined as "a replacement of one amino acid by another, accepted by natural selection". From the mutation data a mutation probability matrix was derived which gives the probability of replacement for amino acid pair at a particular evolutionary distance. This distance is inherent in the relatedness of the proteins within the group from which the mutation data was obtained and is given the measure of 1 PAM. Successive multiplications of the PAM1 matrix by itself N number of times provide a measure of relatedness at greater evolutionary distances (N PAMs). However, at a distance of 2034 PAMs the change in replacement probability with increasing PAM is asymptotic. Therefore, the distance at which the PAM family can provide a measure of similarity is limited. This limitation is a function of the relatedness of the proteins within the original group; i.e. if the proteins within the original group were more distantly related, a PAM matrix may be derived which may be used to detect more distant similarities. The score matrix used in sequence comparison is the log odds matrix of the mutation probability matrix at a particular N PAMs. Dayhoff et al.[14] have found that the log odds matrix for 250 PAMs (PAM250) is an effective scoring matrix for detecting distant relationships.

The BLOSUM series of matrices, Henikoff and Henikoff [15], are also constructed using replacement or substitution frequencies, but from over 2000 sets of similar segments or blocks. A single block can be considered as representing a conserved region of protein family. Unlike the PAM series which infers evolutionary distance by multiplying mutation frequencies, the BLOSUM series infers distance from the percent relatedness of the protein segments that constitute a block. For example,

a BLOSUM62 matrix is derived from a block whose members are 62% similar. To detect similarity between more distantly related proteins one would use, for example, a BLOSUM matrix constructed from a block whose members differ by less than 62%. Henikoff and Henikoff [15] have found that matrices derived from blocks, the BLOSUM series, perform better in alignments and homology searches than those based on accepted mutations in closely related groups as in the PAM series.

2.2.3 Protein Sequence Databases

When comparing an unknown protein to a database, it is of vital importance that the database contain the most recent sequence information [16]. "Sequence relationships critical to important discoveries have on occasion been missed because old or incomplete databases were employed" [9]. There are several independently maintained protein sequence databases; Swiss-Prot [17], PIR-International [18], GenPept [19] and the PDB [20]. The curators of these databases use different strategies for updates, a consequence of which is that one database may contain sequences that others do not. To ensure that an unknown is compared to all known sequences, an investigator should search all databases. This cumbersome task is simplified by searching a non-redundant database which merges sequences that are similar in amino acid composition. For example, NCBI [3] maintains the protein NRDB (non-redundant database) [21] which is constructed from Swiss-Prot, PIR-International, GenPept and the PDB. The criteria for membership in a non-redundant database is generally quite low; the database may contain sequences that differ in composition by less than ten percent or even by only one residue. The sequences in these databases are non-redundant with respect to amino acid composition, but not necessarily with respect to biological information. A search that produces matches that are 80% identical is essentially producing matches to the same sequence. Non-redundant databases are larger than necessary and expectation values calculated from them are artificially high and may be less significant [7]. However, these databases provide the benefit of a 50% reduction in search space over the set of independent databases. "More sophisticated methods for creating derived, composite views of protein and DNA sequences data provide even further reductions [9]. One promising method is to cluster sequences into groups based on percent compositional similarity and select a representative member to which a query is compared. This not only further reduces the

search space, but adds value to the statistical interpretation of results by reducing the number of matches that contain the same biological information. The best approach for database comparison is to perform searches against a database that contains all known compositionally non-identical sequences that are clustered into groups that contain disjoint biological information.

2.2.4 Statistical Assessment of Similarity Scores

The basic output of a similarity search is a list of scores which provide a measure of the extent of similarity between the query and database sequence. Similarity searching programs are greatly improved if this basic output is augmented with a measure of the statistical significance for each score. A natural question is: what is the probability that an alignment could occur by chance alone? Such measures reduce the likelihood that an investigator will call sequences homologous when they are not [7]. In searching protein sequence databases, avoiding high similarity scores with unrelated sequences can be just as important as calculating high scores for related sequences [8]. This subsection describes the statistical measures used in the BLASTP program to assess the significance of scores.

The statistics of local similarity scores for alignments without gaps have been described by Karlin and Altschul [22]. With each *HSP* (high scoring segment pair), an `Expect` and `P-value` is reported. The `Expect` value is the number of times one might expect to observe the occurrence of an *HSP* having score S by chance alone. The `P-value` is the probability of this occurrence in the range 0-1. These values are dependent on several factors [23]: (1) the scoring system, (2) the residue composition of the query, (3) an assumed residue composition for a typical database sequence, (4) the length of the query sequence and (5) the total length of the database.

In certain cases, multiple *HSPs* are found between two sequences. A `P-value` is calculated for each subset of the *HSP* set. The `P(N)` value reported in the output is that of the subset of *HSPs* with the lowest `P-value`. The value of `N` is the number of *HSPs* in the lowest `P-value` set. The `P-value` is a function of `N`; the greater the number of *HSPs* between two sequences, the lower the probability that the match is a chance similarity.

2.2.5 Algorithms for Protein Sequence Comparison

This section places the BLASTP algorithm in the context of the most widely-used algorithms for protein sequence comparison. The majority and most popular protein sequence comparison algorithms fall within two groups; dynamic programming and heuristic (Figure 4). The dynamic programming algorithms are more computationally expensive, but are less likely to overlook a significant match given a particular scoring scheme. They are the methods of choice when a rigorous comparison is required. Heuristic algorithms are less computationally expensive, but may miss borderline regions of similarity; i.e. regions in which the similarity measure exceeds a preset threshold only slightly. They are the methods of choice for database searching because of their relatively low computational requirements.

Algorithm Type	Alignment Type	
	Global	Local
Dynamic Programming	Needleman-Wunsch	Smith-Waterman
Heuristic	FASTA	BLAST

Figure 4: *Classification of Algorithms for Protein Sequence Comparison.* The Needleman-Wunsch [24] and Smith-Waterman[25] algorithms form the basis for most dynamic programming methods. FASTA [11] and BLASTP [1] are the most popular algorithms for protein database searching. These algorithms either report a measure of similarity based on a comparison of two sequences over their entire lengths (global) or among one or more aligned sub-segments (local).

The output of these algorithms is a similarity score(s) based on either a global or one or more local comparisons Algorithms that compute a global score optimally align both sequences over their complete lengths. In doing so, they may assign less than optimal scores to aligned sub-segments. The converse is true for algorithms that report scores computed from local alignments. In fact, no global score is computed. Instead, the output consists of a set of scores computed from aligned sub-segments whose scores are locally optimal. Global alignment algorithms are often the choice if two sequences are known a priori to be closely related. However, distantly-related proteins are more likely to be similar in sub-regions such as an active site rather than over their complete lengths. Therefore, when comparing an unknown protein to a database, local alignment algorithms are better than global alignment algorithms at detecting distantly-related similarity.

Dynamic programming algorithms have been successfully applied to biological sequence comparison problems. The two algorithms that form the basis of most methods are those of Needleman and Wunsch [24] and Smith-Waterman [25]. The former finds the best alignment of two sequences over their entire lengths; the latter, the best local or sub-alignment.

A variant of the Needleman-Wunsch algorithm is shown in Box 2, while the Smith-Waterman algorithm is shown in Box 3. Both algorithms compute a score for the best alignment. The time complexity of these algorithms derives from the traversal of a comparison matrix whose size is proportional to the product of the two sequence lengths. In addition, it is important to note that these algorithms compute a score for the optimal alignment, not the alignment itself; i.e. the positional mapping between residues. The alignment must be obtained by back-tracking the optimal-scoring path through the similarity matrix which takes time proportional to the length of the longer of the two sequences. Thus, the time complexity for obtaining the best score as well as as the actual alignment is of the order $(N \times M) + M$ where N and M are the lengths of the two sequences and M is the longer of the two.. The time complexity is further increased if more than a single best score and alignment are requested.

Dynamic programming algorithms applied to database searches are impractical. The time to search a database of K sequences is of the order $K(N \times M)$ where K is typically of size 10^4 or 10^5. The inapplicability of these algorithms to sequence database scarching lead to the development of heuristic algorithms that sacrifice sensitivity for speed. The most popular of these are FASTA, for computing global alignments, and BLASTP, for computing local alignments.

Box 2

1. S(0 , 0) <- 0
2. **for** j <- 1 to N **do**
3. S(0 , j) <- S(0 , j-1) + ($\sigma[^{-}_{b_j}]$)
4. **for** i <- 1 to M **do**
5. S(i , 0) <- S(i-1 , 0) + ($\sigma[^{a_i}_{-}]$)
6. **for** j <- 1 to N **do**
7. S(i , j) <- **max** { S(i-1 , j-1) + ($\sigma[^{a_i}_{b_j}]$), S(i-1, j) + ($\sigma[^{a_i}_{-}]$), S(i , j-1) + ($\sigma[^{-}_{b_j}]$) }

A Variant of the Needleman-Wunsch Algorithm [26]

The similarity matrix, S, stores the scores of all sub-alignments $[^{a_0 \cdots a_i}_{b_0 \cdots b_j}]$. Each entry is initialised to zero (1). The algorithm traverses the matrix from left to right (4) and from top to bottom (6). The operator, σ, returns the score of an aligned pair which may be residue-residue, $[^{a_i}_{b_j}]$. residue-gap or deletion, $[^{a_i}_{-}]$ or gap-residue or insertion, $[^{-}_{b_j}]$. Scores may be negative. The score of the optimal or highest scoring alignment up until a particular aligned pair is the maximum of the entries surrounding *S(i,j)*; either *S(i-1 , j-1)*, above-left, *S(i-1 , j)*, left or *S(i , j-1)*, above. The algorithm extends the optimal alignment by adding the least costly aligned pair, the score of which is stored in *S(i,j)* (7). The score of the optimal global alignment is thus the value of *S(M,N)*. The algorithm penalises end gaps (2,3,5). The penalty increases with the length of the gap.

Box 3

1. **best** <- 0
2. **for** j <- 1 to N **do**
3. S(0 , j) <- S(0 , j-1) + ($\sigma[^{-}_{b_j}]$)
4. **for** i <- 1 to M **do**
5. S(i , 0) <- S(i-1 , 0) + ($\sigma[^{a_i}_{-}]$)
6. **for** j <- 1 to N **do**
7. S(i , j) <- **max** { 0, S(i-1 , j-1) + ($\sigma[^{a_i}_{b_j}]$), S(i-1 , j) + ($\sigma[^{a_i}_{-}]$), S(i , j-1) + ($\sigma[^{-}_{b_j}]$) }
8. **best** <-**max** { S(i,j), **best**

The Smith-Waterman Algorithm [26]

This algorithm is similar to that of Needleman-Wunsch except that the score of a particular sub-alignment, *S(i,j)*, is at least 0 (7). This allows the local alignment, *S(i,j)*, to re-start at any aligned pair. The score of the optimal local alignment is stored as S is saved in *best* (8). The end-gap penalties are zero (2,3,5).

FASTA and BLASTP operate on the premise that each residue of both sequences need not be compared to detect the highest scoring alignments. Both algorithms first identify short, highly-similar segments which are then expanded. The main assumption made by these algorithms is that any significant alignment encompasses one or more of these segments. The difference between dynamic programming and heuristic algorithms is best understood in terms of a comparison matrix where one sequence is positioned horizontally and the other vertically. Each entry is a measure of similarity, or score, between two residues. The dynamic programming algorithms fill each entry in the matrix. The heuristic algorithms first fill a subset of entries forming common sub-sequences of high similarity. Then, neighboring entries are filled or calculated until the score of an extended aligned segment is maximised. The complexity of the heuristic algorithms remains on the order of $O(N \times M)$, but the number of computations based on residue-residue comparisons is greatly reduced.

The FASTA algorithm [11] is an improvement over the initial FASTP algorithm [27] [28]. It operates in four steps: (1) Exact matches of length one or two are identified, (2) Segments that contain identity matches are re-scored using a score matrix. These are termed "initial regions", (3) Initial regions that meet a "joining threshold" are linked and (4) The highest scoring initial region is scanned using a dynamic programming, usually Needleman-Wunsch. The output of the FASTA program is the highest scoring region of global similarity, including gaps.

BLAST also first determines short segments of high similarity. However, BLAST does not only search solely for identity matches, but uses a score matrix to identify short similar segments (hits). Another difference is that BLAST does not build global alignments by linking these segments, but attempts to extend each one into an alignment whose score is locally maximal. A significant simplification that is made by the BLAST algorithm is to exclude gaps in the alignment. This improves search time, but is detrimental to the detection of similarity between distantly related proteins which are likely to contain insertions and deletions. Recently, the BLAST algorithm has been enhanced with the ability to produce gapped alignments [29].

2.3 The BLASTP Algorithm

This section provides a parameterised description of the BLASTP program and each of the three steps of the algorithm. The description of the program is given in Box 4, while those for each of the three steps - neighborhood construction, hit detection and hit extension - are given in Boxes 5, 6 and 7 respectively

The BLASTP program (Box 4) first builds the neighborhood (Box 5), then iteratively retrieves a subject sequence from the database and scans it for hits (Box 6). Upon detection of a hit, the extension step (Box 7) is invoked.

Box 4
The BLASTP Program

Q, query sequence, $q_0q_1q_2...q_l$
DB, subject sequence database, size = K
M, function that returns the alignment score
HSP, set of high scoring segment pairs or alignments
W, word size
T, threshold word score
X, falloff score
S, threshold alignment score

HSP = **BLASTP**(Q,DB,M,W,T,X,S)
    N, neighborhood word set
    SB, subject sequence, $s_0s_1s_2...s_m$
    N = **Build-Neighborhood**(Q, M, W, T)
        for (i=0 ; i<K ; i++) {
            SB = DB[i]
            Scan(SB,N)

The BLASTP algorithm works in three steps:
1. **Neighborhood Construction.** A set of words of length W, the neighborhood N, is computed. Each word scores at least T with some word of equivalent length in the query sequence Q.
2. **Hit Detection.** Each subject SB in the database DB is scanned for matches to a word in N.
3. **Hit Extension.** The match or hit H is extended into a potentially higher scoring alignment.

Box 5

Step One - Neighborhood Construction

N = **Build-Neighborhood**(Q,M,W,T)

AA, amino acid alphabet

for (i=0 ; i < l-W+1 ; i++)

if $\exists$ a word $n_0 n_1 n_2 ... n_{W-1}$, where $n_j \in AA$

such that $M(q_i q_{i+1} ... q_{i+W-1} || n_0 n_i ... n_{W-1}) >= T$

then $N \bigcup < n_0 n_i ... n_{W-1}, q_{i+W-1} >$

This step is parameterised by the query sequence Q, the score function M, the word size W and the threshold word score T. The alphabet of residues is AA. This step outputs the neighborhood, N. The query sequence Q is scanned. Each query word may have zero or more neighbors. The set of neighbors of all query words is the neighborhood. The neighborhood is a set of tuples of the form $<$ *neighbor*,*offset*$>$ where *neighbor* is the word that matched the query word at *offset*.

Box 6

Step Two - Hit Detection

Scan(SB,N)

H, word hit, composed of (*s_off*,*q_off*), the offsets of H on SB and Q.

for (j=0 ; j < m- W+1 ; j++)

if $((s_j s_{j+1} ... s_{j+W-1}) \in N)$

$H.s\_off = s_{j+W-1}$

$H.q\_off =$ *offset*, where $< s_j s_{j+1} ... s_{j+W-1}$, *offset*$> \in N$

Extend(H)

This step is parameterised by a subject sequence SB and the neighborhood N. A word hit H is an alignment of a query word and subject word whose offsets are *q_off* and *s_off* respectively. The subject SB is scanned for matches to a member of the neighborhood. When a match is found, the extension step is invoked.

Box 7

Step Three - Hit Extension

```
1  Extend (Q,SB,W,H,M,X,S)
2      q_beg = q = H.q_off-W, q_end = q_pos = H.q_off, s_beg = s = H.s_off-W, s_end = H.s_off
3      do
4          sum += M(Q_q||SB_s)
5          if(sum > score) then score = sum, q_end = q, s_end = s
6          else if(sum <= 0) then sum = 0, q_beg = q, s_beg = s
7          q++, s++
8      while(q < q_pos)
9      if((x = -score) < X) then x = X
10     leftq = q_beg, lefts = s_beg, rightq = q_end, rights = s_end
11     leftsum = rightsum = leftscore = rightscore = 0
12     Left Extension:
13     q = leftq , s = lefts, sum = leftsum
14     do
15         q--, s--
16         sum += M(Q_q||SB_s)
17         if(sum > 0) then
18             score += sum, sum = 0, q_beg = q, s_beg = s
19             if((x = -score) < X) then x = X
20     while (sum >= x)
21     if (score > rightscore) ∧ (rightsum > X) ∧ (-rightscore > X) then
22             leftq = q, lefts = s, leftsum = sum, leftscore = score;
23             Right Extension:
24             q = rightq, s = rights, sum = rightsum
25             do
26                 sum += M(Q_q||SB_s)
27                 if(sum > 0) then
28                     score += sum, sum = 0, q_end = q, s_end = s
29                     if((x = -score) < X) then x = X
30                 q++, s++
31             while (sum >= x)
32             rightq = q
33             if (score > leftscore) ∧ (leftsum > X) ∧ (-leftscore > X) then
34                 rights = s, rightsum = sum, rightscore = score
35                 goto Left Extension
36     if (score >= S) then HSP ⋃ ((Q_q_beg...Q_q_end||SB_s_beg...SB_s_end)
```

Box 7 continued

This step is parameterised by the word hit H, the word size W, a scoring function M, the falloff score X, the threshold alignment score S and the query and subject sequences Q and SB (1). This step attempts to extend a hit into a longer, potentially higher scoring alignment. The offsets *q_beg*, *q_end*, *s_beg* and *s_end* mark the maximal scoring alignment. They are first set to the delimiters of the word hit (2). The first loop (3-8) then sets them to the delimiters of the maximal scoring sub-alignment within H. The hit is traversed from q to *q_pos*. Residue pair scores are accumulated in *sum* (4). When *sum* is positive (5) it is added to *score* and *q_end*||*s_end* is advanced right. A negative *sum* (6) causes *q_beg*||*s_beg* to be advanced right effectively excluding the negative scoring residue pair from the maximal scoring sub-alignment of H. Parameters specific to either the left or right extensions are initialised in (10-11). The second loop (12-20) extends in the left direction. Residue pair scores are accumulated in *sum* (16). If *sum* is positive, it is added to *score* then reset to zero and the alignment is extended (17-18). If *sum* falls below x, the extension terminates (20). Longer extensions are favored by allowing x to be set to *-score* (9,19,29). A right extension occurs if the conditions in (21) hold which is always the case at first. The values of *s*, *sum* and *score* are saved in the left extension parameters, *lefts*, *leftsum* and *leftscore* respectively (22). The third loop works in the same manner as the second except that the extension proceeds in the right direction (23-31). The left extension may continue if the conditions in (33) hold. If so, *s*, *sum* and *score* are saved in the right extension parameters, *rights*, *rightsum* and *rightscore* respectively (34). The maximal scoring alignment is stored in the *HSP* set if *score* meets the threshold S (36). Figure 5 illustrates the dynamics of the extension algorithm.

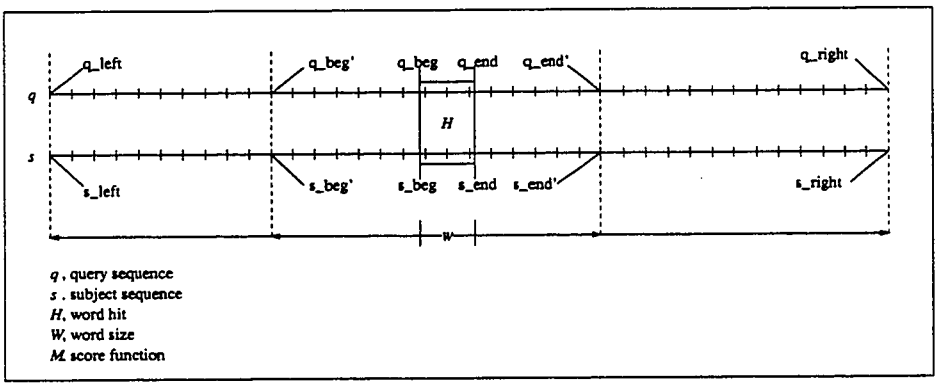

Figure 5: *Dynamics of Extension Algorithm.* A word hit H is a sub-alignment of length W, $H = (q_{beg} \cdots q_{end} || s_{beg} \cdots s_{end})$ whose score, $M(H) >= T$. The extension algorithm finds the locally maximal alignment starting from H. Extension continues in either direction until $M(q_{left} \cdots q'_{beg} || s_{left} \cdots s'_{beg}) < X$ or $M(q_{right} \cdots q'_{end} || s_{right} \cdots s'_{end}) < X$. The total score of the alignment is $M(q'_{beg} \cdots q'_{end} || s'_{beg} \cdots s'_{end})$

Chapter 3

Reverse Engineering of BLASTP Program

3.1 Introduction

Reverse engineering is an activity in which a design is derived from its implementation. In software programs, domain knowledge and design expertise are contained within the source code. The reverse engineering process extracts this design knowledge and expertise from the source code and presents it at a higher level of abstraction, using more understandable models.

This work uses a reverse engineering approach (Figure 6) to identify and describe those parts of the BLASTP program that, if optimised, would have the greatest impact on the program's performance. A program model is constructed which describes the design and implementation of such time expensive parts. The model provides a framework in which an understanding of how a modification to one part of the program might effect the overall performance. In addition, the model suggests how a modification might effect the rest of the program.

The program model is comprised of a set of sub-architectures each of which is presented in subsection 3.2.4.2. They were derived and described using the reverse engineering methodology given in section 3.2. The program model mostly describes the design and implementation of the BLASTP algorithm. From the program model, a

description of the algorithm is obtained that is more comprehensive than any description given in the literature. This parameterised description of the algorithm, already presented in subsection 2.3, essentially expands literature descriptions to include the sub-steps of each of the three major steps; neighborhood construction, hit detection and hit extension. Each sub-step is a potential focus for optimisation.

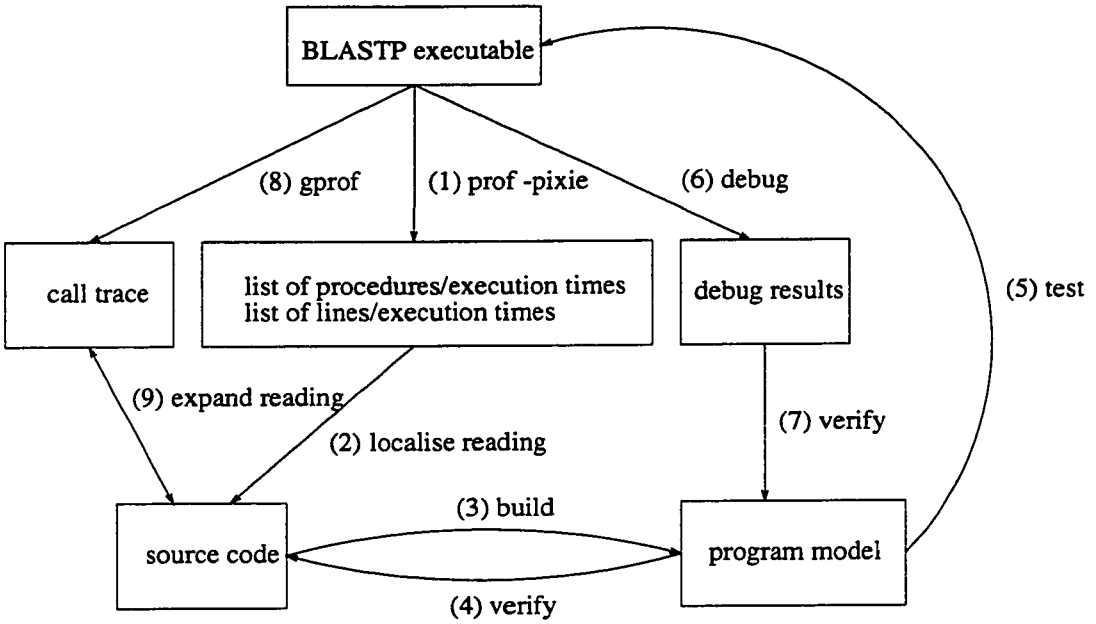

Figure 6: *Reverse Engineering Process.* The process consists of profiling the program using execution profilers and studying the source code to construct a program model that describes the sub-architecture of the program that accounts for a large proportion of the overall execution time. The profiler `prof` [30] was used to obtain a list of procedures and lines of code ordered according to execution time (*1*). This data indicated which parts of the program, if optimised, would give the largest speed-up. The `prof` data also provided a starting point for reading the source code (*2*). The model was constructed from a study of the source code (*3*) that involved forming an assumption about the functionality of a particular section of code and then verifying that hypothesis by re-examining the source code (*4*) and testing using debuggers (*5,6,7*). The model was expanded by linking already studied code to related code using the call trace provided by the `gprof` profiler [31] (*8,9*).

3.2 Methodology

3.2.1 Test Search

A single test search is used as a standard. The search program is BLASTP, version 1.4 [32]; the default values for all parameters are used. The database is the protein NRDB (non-redundant database) [21], a conglomerate of several protein sequence databases in which identical sequences are merged into one entry. The database contains 252,307 entries. The query sequence (length = 312 residues) is given in Box 1, section 1.2. The average length of database sequence is 283 residues. The performance results presented in subsection 3.2.4.1 are part of the execution profile of the test search run on a DEC Alpha computer.

3.2.2 Execution Profilers as Reverse Engineering Tools

Figure 6 outlines the reverse engineering process. Execution profilers account for the execution time of the procedures that comprise a program. They have two main outputs: 1) A flat profile which lists all the procedures and the seconds of execution time for which each of them is accountable. The procedures are listed in decreasing order of execution time. The individual times sum to the total execution time of the program. 2) A call graph which lists each procedure together with information about the procedures that are its parents and children. The graph is sorted by the sum of the time for the procedure itself plus the time inherited from its descendants.

This work uses the profilers `prof` [30] and `gprof` [33] [31] as reverse engineering tools to analyse the test search (subsection 3.2.1). The `prof` profiler with the `-pixie` [34] option provided a flat profile which gave a quick overview of the procedures used and showed which ones account for large fractions of the overall execution time. The `gprof` profiler provided a dynamic call graph which was used to deduce the program's architecture; i.e. control flow and structures. In addition, the profilers were run with various options to provide additional data. The `prof -pixie` program was run with the `-heavy` option in order to obtain a listing of code lines ordered from the most to least heavily used. Each code line is referenced to a line number in a source file. The `-numbers` option of `gprof` provided an alphabetical list of procedures. In this case, each procedure is referenced to a line number in a source file.

The profile data suggest a reverse engineering methodology which reveals areas of the program that, upon optimisation, may yield a substantial performance enhancement. Execution times are measured in CPU cycles, a consistent basis for comparison among individual or sets of procedures. The flat profile gives the most expensive procedures, providing a starting point for the textual study of the program code. Similarly, the `-heavy` option provides the most expensive lines of code which are also good points of focus for analysis. While the `prof` data provides a starting point, the `gprof` data provides a basis for continuation. The call graph was used to obtain the parent(s) and children of the expensive procedures. By expanding through the call hierarchy, a set of procedures that comprise a sub-tree of the call graph was constructed. The abstraction implemented by a sub-tree is revealed from a textual study of the code, which includes the analysis of structures that are a local to a procedure as well as those that are passed between procedures as parameters.

The result of this bottom-up process is a description of a sub-architecture whose implementation accounts for a relatively large fraction of the execution time, making it a potential area of optimisation. In this way, the sub-architectures that implement the second and third step of the algorithm were deduced and modelled. Other sub-architectures whose implementations are not expensive could not be deduced bottom-up. In such cases, a top-down approach was used. The call-graph was traced starting from the main procedure. Again, procedures were coupled with structures to construct a sub-architecture that implements a particular abstraction. The sub-architectures that implement the first step of the algorithm and the event loop, which first fetches a database sequence then invokes the algorithm, were constructed with this approach.

3.2.3 Modelling of Program Architecture

The sub-architectures reverse engineered using the process described in subsection 3.2.2 are modelled using elements of the Object Modelling Technique (OMT) [35] and the Unified Modelling Language (UML) [36]. These techniques provide established models and notations through which the relationships among a program's entities can be effectively communicated. Although BLASTP is a procedural program written in the C programming language, its design can be accurately described using these object-oriented models. For a discussion on which modelling concepts and notations are used in this work, consult appendix A.

The representation of C programming constructs with object-oriented modelling elements is exemplified in Figure 7. The user-defined `struct` type of C or structure [37] [38] is modelled well as a class (Figure 7-A). Procedures that contain, within its parameter list, a reference to a structure are modelled as operations of the class that represents that structure. The members of the structure are modelled as the attributes of that class. In addition, several procedures may contain a local variable used for the same purpose. This variable can also be abstracted as a class attribute. A structure member that is a reference to another structure implements an association(Figure 7-B). Nested structures that logically form a "part-whole" hierarchy can be modelled using aggregation. One structure may send a message to another if a procedure that is grouped with the first structure invokes a procedure that is grouped with the second. The content of the message is a member of the first structure which is passed to the second procedure as an argument (Figure 7-C).

Algorithms are presented with pseudo-code, the level of abstraction of which is dependent upon the complexity of the code that implements the algorithm. For the scanning algorithm, the code is a sufficient description. For more complex algorithms, the code hides the fundamental concepts of the algorithm. This is the case with the neighborhood word algorithm, which is illustrated as invoking itself recursively, but is not implemented in this manner. With the extension algorithm, the code is simple but detailed. The pseudo-code presented for this algorithm is similar to the source code. Here, the omission of detail would result in an incomplete description of the algorithm.

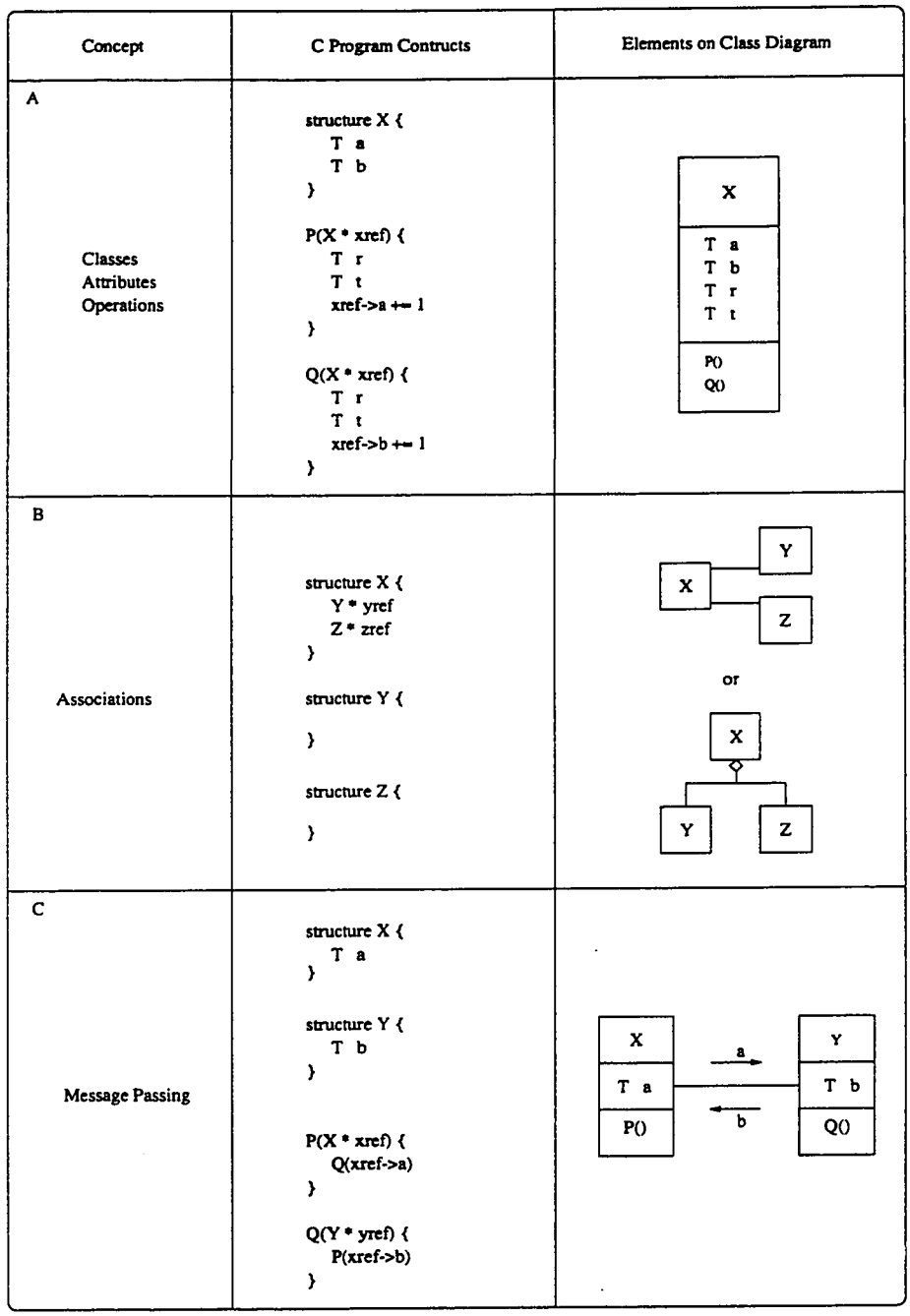

Figure 7: *Object-Oriented Modelling of C Language Elements.* Three examples are shown, *A*, *B* and *C*. The first column lists the object-oriented concept. The second shows a set of C structures and procedures in which that concept is implemented. The third shows the equivalent group of object-oriented elements.

3.2.4 Results of Profile Analysis and Reverse Engineering of Program

3.2.4.1 Profile Analysis

The execution profile of the program indicates those parts of the program whose optimisation would result in the greatest speed-up. The set of procedures that implement the BLASTP algorithm account for greater than 99 percent of the execution time (Figure 8). The third step, word hit extension, accounts for greater than 93 percent of the time. Within the extension procedure, the three extension loops account for greater than 76 percent of the time (Table 1). Furthermore, the profile reveals that the line of code that accesses the residue pair score matrix is the most expensive, accounting for greater than 63 percent of the execution time (Table 2).

Amdahl's Law [5] states that the greatest savings in execution time stands to be gained by optimising the most time expensive part of the program. Clearly the extension procedure should be the focus of optimisation. The profile further suggests that a finer focus should be the extension loops and the line within that accesses the score matrix. Therefore, an algorithm optimisation that invokes the extension procedure less frequently, executes less extension loops or reduces the cost of the matrix access can potentially improve the overall performance of the program.

Algorithm Step	CPU cycles x 10^9	% CPU cycles
Neighborhood Construction	0.002	1×10^{-4}
Hit Detection	1.2	93.2
Hit Extension	20.0	5.9
	21.2	99.1

Test Search

Program BLASTP version 1.4

Query: gil129937l, polygalacturonase, 312 residues

Database: GENBANK - proteinNRDB, 252,307 sequences, 71,280,657 residues

Platform DEC Alpha / UNIX

Figure 8: *Execution Time for Three Steps of Algorithm.* The execution time is measured in CPU cycles. The total cycles for a step is the sum of the cycles for the set of the procedures that implement the step. The algorithm accounts for more than 99 percent of the execution time. The third step, word hit extension, is the most expensive, accounting for more than 93 percent of the time.

Code	Reference to Figure 20	CPU cycles x 10^9	% CPU cycles
start extension loop	15-24	1.9	8.7
left extension loop	41-49	7.3	34.0
right extension loop	63-71	7.3	34.0
		16.5	76.7

Table 1: *Execution Time for BlastWordExtend Procedure Code Sequences (row-address).* The three extension loops account for more than 76 percent of the execution time.

Line	Ref. Figure 20	CPU cycles x 10^9	% CPU cycles
matrix access - start loop	16	1.4	6.7
matrix access - left loop	42	6.0	28.2
matrix access - right loop	64	6.0	28.2
		13.4	63.1

Table 2: *Execution Time for BlastWordExtend Procedure Lines (row-address).* The three lines that access the residue pair score matrix to retrieve a score account for more than 63 percent of the execution time.

3.2.4.2 Reverse Engineering of Code

The optimisations are implemented in the existing BLASTP version 1.4 program. A reverse engineering approach is used to obtain the set of program sub-architectures that implement the algorithm. Class diagrams [35] are used to communicate the program's design. An overview of the architecture that was reverse engineered from the source code is given in Figure 9. This model provides a sufficient understanding of the program in order to implement the optimisations described in Chapter 4 and to anticipate how any changes made might effect the rest of the program.

The structure that implements the algorithm is `BlastWordFinder`. `BLAST_Filter` implements the neighborhood construction step. `BlastWordFinder` uses `DFA` to scan the the subject for matches (step 2 of the algorithm). `DFA` is composed of non-accepting

states (DFA_State) and accepting states (DFA_Accept). DFA_Accept contains a list of query offsets DFA_Patlist for each neighborhood word. BLAST_WordExtender implements step 3 of the algorithm. The query and subject sequence are stored in BLAST_Str structures. BLAST_WFContext is composed of two BLAST_Str structures, which contain the query and subject sequence, and BLAST_Score Blk which contains the score matrix.

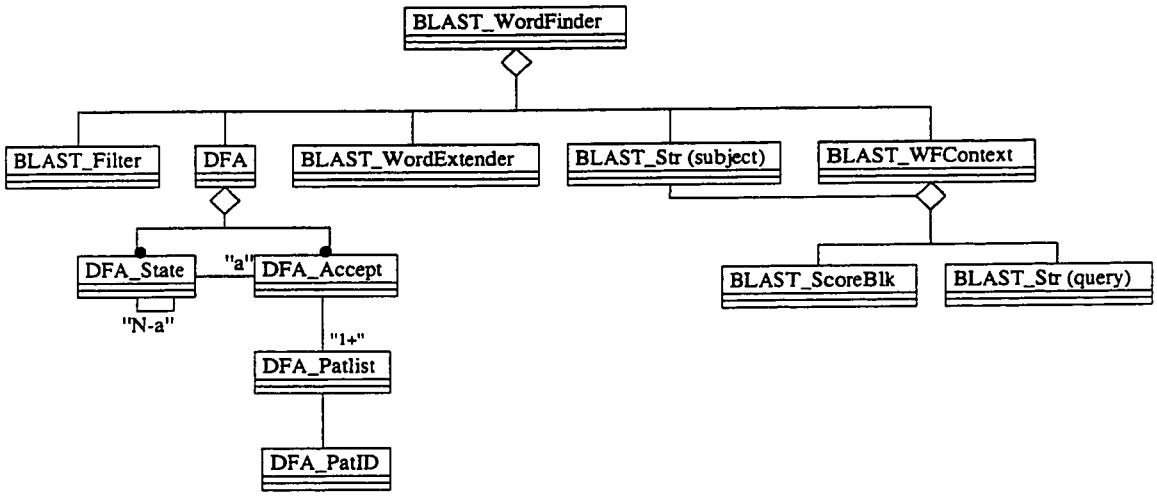

Figure 9: *Sub-Architecture that Implements the BLASTP Algorithm.*

3.2.4.2.1 Implementation of Event Loop BLASTP is essentially a protein database scanner. The program iteratively retrieves a subject from the database, then scans it for subsequences (hits) that are similar to the query (Figure 11). The search loop is implemented by the procedure blastp_dosearch. Two events occur with each iteration: (1) The protein sequence database db.seq is accessed to retrieve the subject which is placed in the BLAST_Str structure. This is initiated by the procedure db_get_seq and (2) The subject is scanned for hits by BLAST_WordFinder. The structures that comprise the program's interface to the protein sequence database are DBFile, BDBFile and MFILE. Each iteration consists of positioning the database file pointer to point to the next subject (db_seek), retrieving the next subject db_get_long and invoking the scanning step of the algorithm with the procedure BlastWordFinderSearch.

The BLASTP application is a client of the BLAST and GISH libraries (Figure 10). The former contains the structures and procedures that implement the algorithm, the latter those that implement database management and retrieval. Each library contains a structure that acts as an interface to the database. The DBFile structure fulfils this role for the BLASTP application, while the BDBFile and MFILE structures do the same for the BLAST and GISH libraries respectively. These three structures form a hierarchy; i.e. DBFile is a client of BDBFile which is a client of MFILE. Each contains a set of file operations; open, seek, close and get. Each like operation from each structure forms a procedural hierarchy; for example, db_seek calls blast_db_seek which invokes mfil_seek.

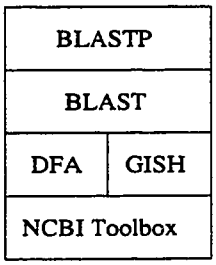

Figure 10: *Architecture of the BLASTP Application and Libraries.* The architecture is open. The BLASTP program uses for libraries: (1) BLASTAPP, (2) BLAST, (3) GISH, (4) DFA and (5) the NCBI ToolBox.

The protein sequence database is first processed by the setdb program prior to searching so that it is more efficiently accessible by the BLASTP program (Figure 12). Each FASTA entry consists of two components: (1) A header which contains the sequence id and a descriptor and (2) A letter representation of the amino acid sequence. The setdb program reads each FASTA entry, converts the sequence to an integer representation and writes it to the .seq file. In addition, the program writes the header to the .ahd file. The offset into .seq of the sequence is stored in seq_beg. Likewise, the offset into the .ahd file of the header is stored in hdr_beg. The array a_tob maps the character representation of the amino acid to its integer one. After all entries are converted, setdb writes summary information about the database to the .atb file. This includes the number of entries entry, the length of the longest sequence mxlen, the type of residue restype and the index arrays header_beg and seq_beg.

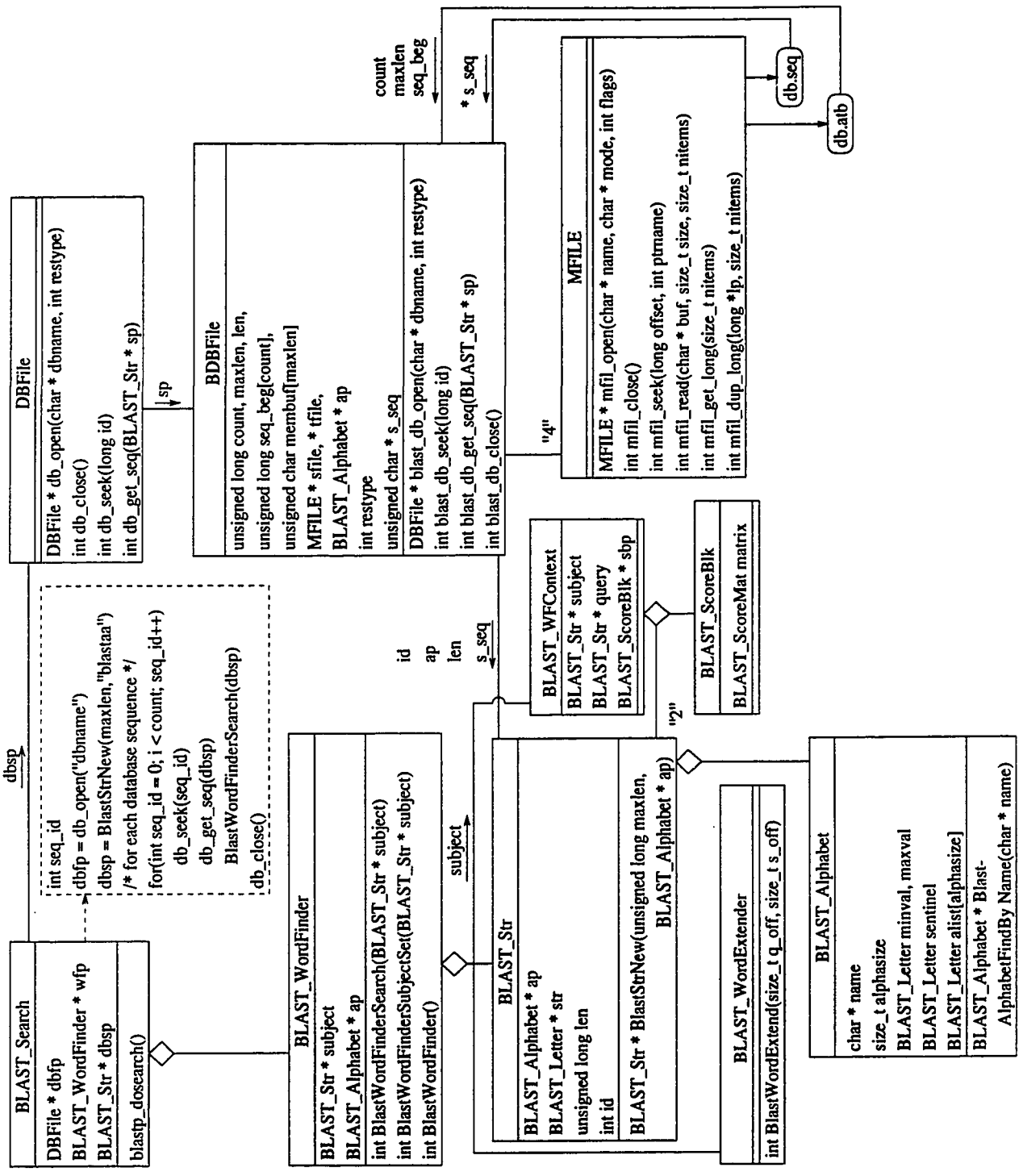

Figure 11: *Sub-Architecture that Implements the Event Loop.*

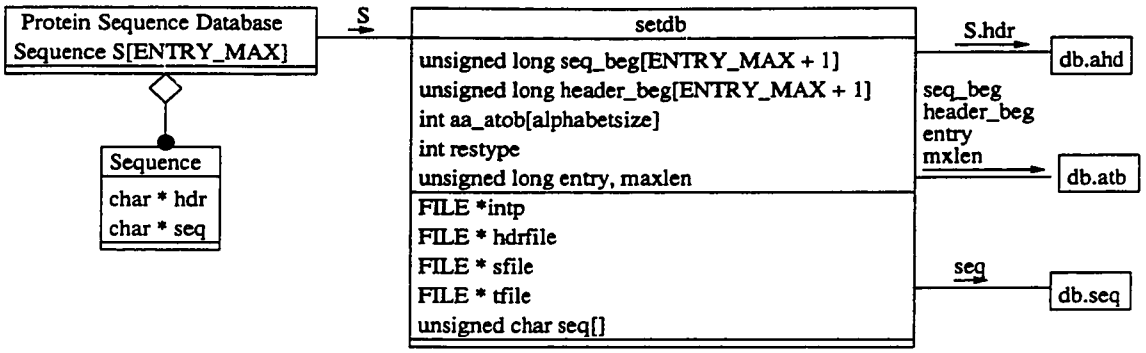

Figure 12: *Processing by the* `setdb` *Program.*

3.2.4.3 Implementation of Algorithm

3.2.4.3.1 Construction of the Neighborhood The program structure that implements the neighborhood construction step of the algorithm is shown in Figure 13. The neighborhood words are the patterns that are recognised by the `DFA`. The computation of the neighborhood is initiated by `BLAST_WFContext`. `BLAST_Filter` contains the set of procedures that calculate the subset of the neighborhood for a particular query word. Upon computing a neighbor, `BLAST_WFContext` adds it to the `DFA`.

The procedure `BLAST_WFContextWordFltrApply` traverses the query and, for each query word, $q_{i-W+1}...q_{i-1}q_i$, invokes the procedure `BlastStdWordFltr` which determines whether or not to calculate the subset of the neighborhood for that word. The procedure `BlastSelfScoreWordFltr` scores the query word against itself. The identity match, $q_{i-W+1}...q_{i-1}q_i||q_{i-W+1}...q_{i-1}q_i$, is the highest scoring neighbor. Therefore, a word whose self-score does meet T will have no neighbor that does so. If T is met, the neighborhood is computed by the procedure `BlastNeighborHoodWordFltr` which, upon finding a neighbor, calls back the procedure `BlastWFContextAddWordFltr`. This procedure augments the `DFA` with the neighbor pattern.

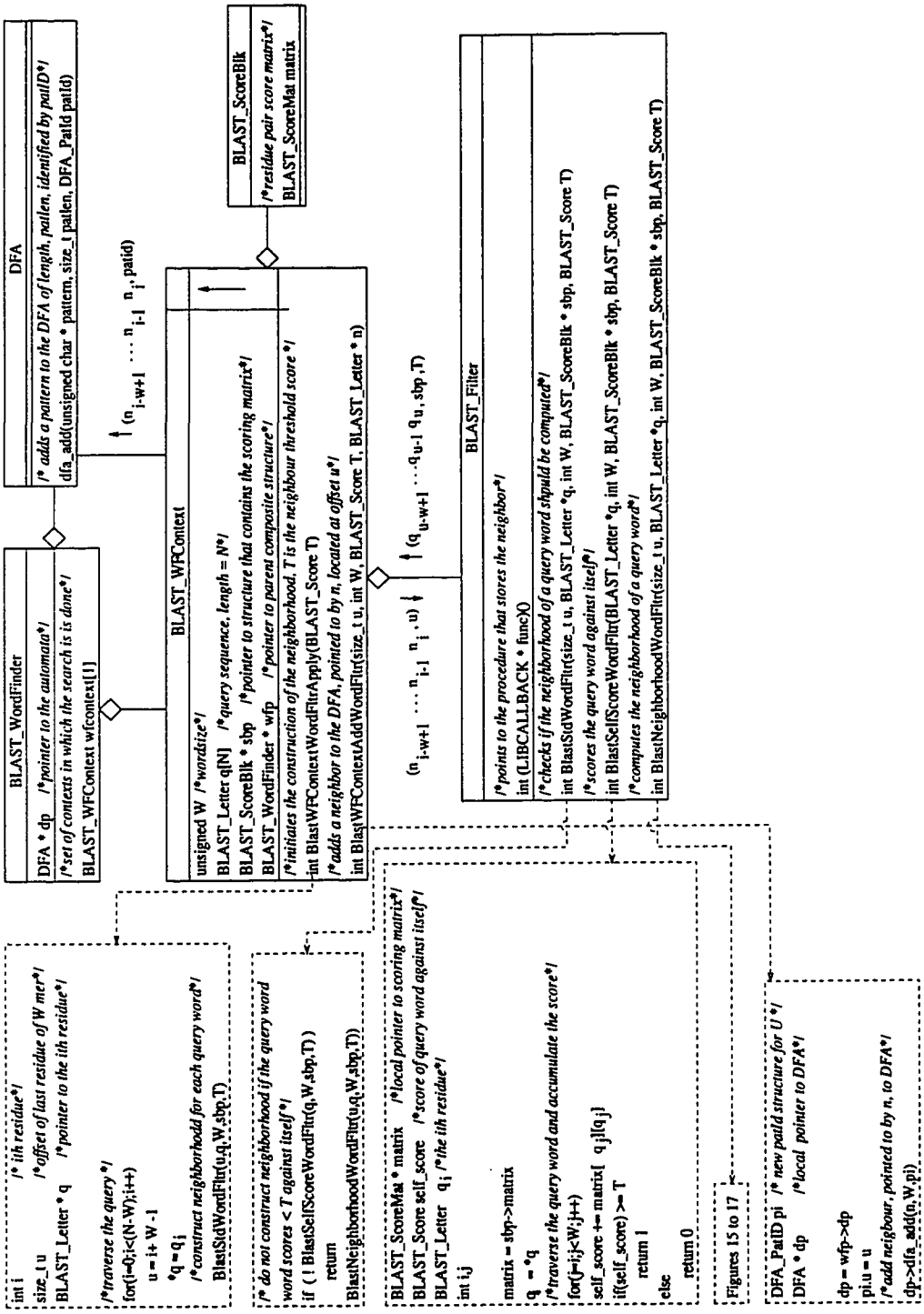

Figure 13: *Sub-Architecture that Implements the Neighborhood Construction Step.*

The complexity of the neighborhood word computation is proportional to the word size W, illustrated in Figure 14. A naive algorithm that computes all possible neighborhood words is given in Figure 15. The BLASTP program implements an optimised algorithm that constructs the set of neighbors using a special data structure, the order matrix (Figure 16) that significantly reduces the number of neighborhood words that need to be computed. The optimised algorithm (Figure 17) traverses the matrix such that the neighborhood words are tested in a partial order from highest scoring to lowest scoring.

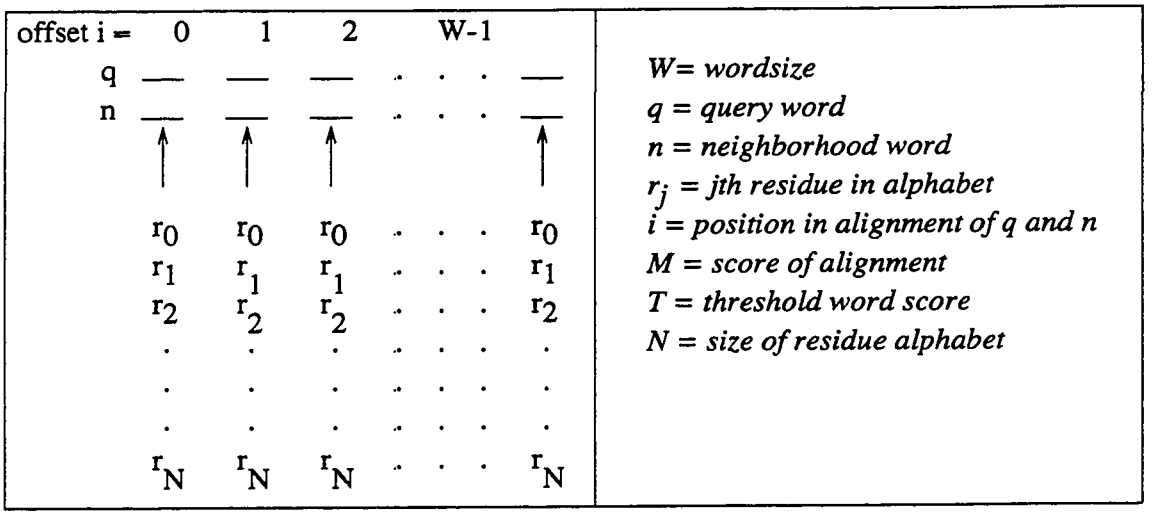

Figure 14: *Neighborhood Word Problem.* Given a query word, $q = q_i q_{i+1} \ldots q_{i+W-1}$, of length W, compute all neighbors, $n = n_i n_{i+1} \ldots n_{i+W-1}$, such that $M(q||n) >= T$. Each position i in n can be occupied by N residues therefore there are N^W neighbors to test.

```
matrix[N][N]        /* matrix = residue pair score matrix */
n[W]                /* N = size of residue alphabet */
i=0                 /* W = wordsize */
M_i = 0             /* M_i = score of sub-alignment q_0 q_1 ... q_n || n_0 n_1 ... n_i */

1  NeighborHood(M_{i-1}, i)
2     j = -1
3     do
4        j++
5        n_i = r_j
6        M_i = M_{i-1} + matrix[q_i][n_i]
7        if ( i == W-1 )
8           if ( M_i >= T )
9              SaveNeighbor(n)
10          else
11             NeighborHood(M_i, i+1)
12    while (j < N)
```

Figure 15: *Naive Neighborhood Word Algorithm.* The algorithm tests N^W neighbors - see Figure 14. It is recursive in nature. Each call to `Neighborhood` (1) tests each r_j at position i (3-12). The position corresponds to the level of recursion. At each position, M_i is computed (6). The neighbor is saved (9) if the current position is the last position in the word (7) and the score of the word alignment, M_{W-1}, meets the threshold (8). If the current position is not the last, a recursive call is made; i.e. n_{i+1} is tested with each r_j (11). The algorithm is depth first. It first tests each r_j at n_{W-1}, while placing r_0 at each position i in $n_0 n_1 ... n_{W-2}$. Next, it places r_0 at each position i in $n_0 n_1 ... n_{W-3}$, r_1 at n_{W-2} and again tests each r_j at n_{W-1}.

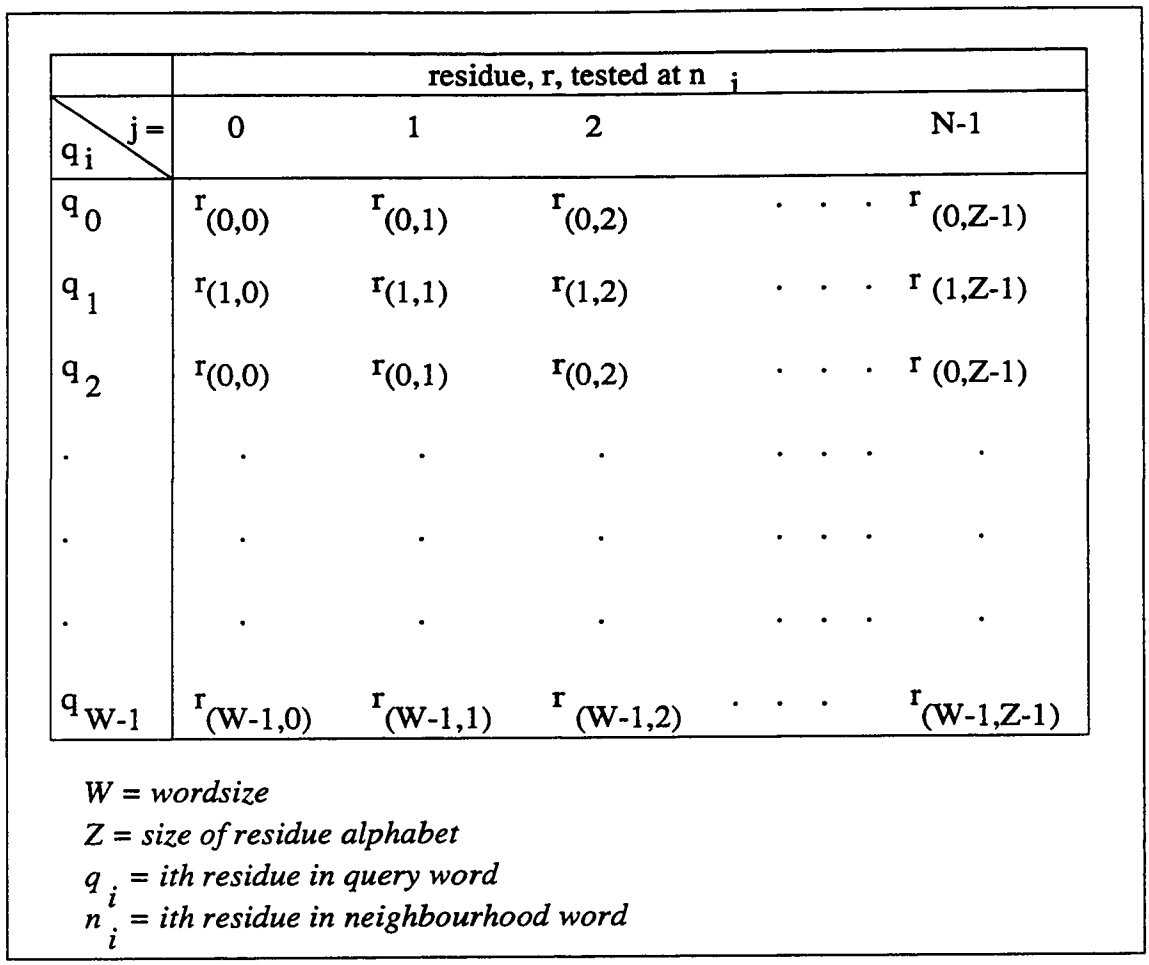

Figure 16: *Order Matrix.* The row indices are the residues q_i. Recall from Figure 15 that i is the position in the alignment $q_0...q_1q_{W-1}||n_0...n_1n_{W-1}$, and corresponds to the ith level of recursion. The column indices j are ordered from $0\,to\,N-1$. Each entry, $order[q_i][r_{(i,j)}]$ is the jth highest scoring residue pair of the form $(q_i, r_{(i,j)})$; i.e. the residues in each row are ordered from highest to lowest pair-wise score with the row index q_i. Note that the highest scoring neighbor is the identity match to q (Box 9). The row-ordering allows the algorithm to stop the traversal of a row prior to testing each $order[q_i][r_{(i,j)}]$ at n_i.

Box 9

$n_{highest} = order[q_0][0], order[q_1][0], ..., order[q_{W-1}][0]$

$n_{lowest} = order[q_0][N-1], order[q_1][N-1], ..., order[q_{W-1}][N-1]$

```
matrix[N][N]      /* matrix = residue pait score matrix */
n[W]              /* N = size of residue alphabet */
i=0               /* W = wordsize */
S_i = 0           /* M_i = score of sub-alignment q_0 q_1 ... q_i || n_0 n_1 ... n_i */
order[W][N]       /* order = order matrix */

1    NeighborHood(M_i-1, i)
2    j = -1
3    do
4       j++
5       n_i = order[q_i][j]
6       M_i = M_i-1 + matrix[q_i][n_i]
7       if ( i == W-1 )
8          if ( M_i >= T )
9             BlastWFContextAddWordFltr(n)
10         else
11            noProgress = 1
12      else
13         progressNextLevel =  NeighborHood(M_i, i+1)
14   while ( (j < N) && ( ! noProgress ) && (progressNextLevel > 0) )
15   return j
```

Figure 17: *Optimised Neighborhood Word Algorithm.* The algorithm is identical to the naive one in all but three aspects: (*1*) The residue n_i is obtained form *order* (5), (*2*) There are two additional loop conditions (14) which may terminate further testing of r_j at position n_i for values $> j$. These conditions are *noProgress* (Box 10) and *progressNextLevel* (Box 11) and (*3*) The procedure returns the extent of traversal j of a particular row which is assigned to *progressNextLevel* (13). `BlastWFContextAddWordFltr` adds the neighborhood word to the `DFA`.

Box 10

if $M(q_0q_1...q_i||n_0n_1...r_{(i,j)}) < T$

then $M(q_0q_1...q_i||n_0n_1...r_{(i,j+k)}) < T$, where $k > 0$

since $M(q_i||r_{(i,j)}) >= M(q_i||r_{(i,j+k)})$

Box 11

if $M(q_0q_1...q_iq_{i+1}||n_0n_1...r_{(i,j)}r_{(i+1,0)}) < T$

then $M(q_0q_1...q_iq_{i+1}||n_0n_1...r_{(i,j+1)}r_{(i+1,0)}) < T$

since $M(q_i||r_{(i,j)}) >= M(q_i||r_{(i,j+1)})$

3.2.4.3.2 Detection of Word Hits The program structure that implements the scanning phase of the algorithm is shown in Figure 18. Three structures implement the scanning step; `BLAST_WordFinder`, `DFA` and `BLAST_WordExtender`. `BLAST_WordFinder` traverses the subject and uses the `DFA` to detect word hits. Upon detection, the offsets of the hit are passed to `BLAST_WordExtender`.

The `DFA` structure recognises the neighborhood. `BLAST_WordFinder` uses the `DFA` to detect common subsequences of length W between the query and subject; i.e. a word hit. An automata is a network of states and transitions. The program employs a Mealy machine implementation wherein output activity is linked to state transitions (accepting transitions), rather than in association with the states themselves (accepting states) as in a Moore machine [39] [40]. Each state is an array of pointers to states. Its size is `N`; the size of the residue symbol alphabet. Thus, each state contains a pointer to a next state for each symbol. A transition is taken by dereferencing a pointer. A non-accepting transition dereferences a pointer to a state `DFA_State`. However, an accepting transition dereferences a pointer to `DFA_Accept`. Conceptually, this structure is part of the transition and contains a pointer to a next state which is the destination of the transition. An accepting transition indicates that the pattern in the subject sequence, $s_{i-W-1} \cdots s_{i-1} s_i$, is perfectly matched to a neighborhood word, but not necessarily to a query word. `DFA_Accept` contains a pattern list that is comprised of offsets, `u`, of the query such that $M(q_{u-W-1} \cdots q_{u-1} q_u || s_{i-W-1} \cdots s_{i-1} s_i) >= T$.

The scanning algorithm consists of two iterative steps (Figure 18, procedure `BlastWordFinder`): (1) Following transitions through the automata and (2) Processing an accepting transition. A pointer to the current state of the `DFA` is kept in `state`. As the subject is traversed from `i=0 to N-1` the transition whose label is s_i is taken; `state->next[`s_i`]`. The macro `DFA_ISACCEPTING` returns true if `state` points to a structure of type `DFA_Accept`; false if it points to a structure of type `DFA_State`. An accepting transition is processed by first checking if the end of the subject has been reached, `(i==N)`, then obtaining pointers to the pattern list, `plp`, and to the next state, `retState`. `BlastWordExtend` is invoked for each word hit; i.e. for each query offset `u` in `pl`, the pattern list.

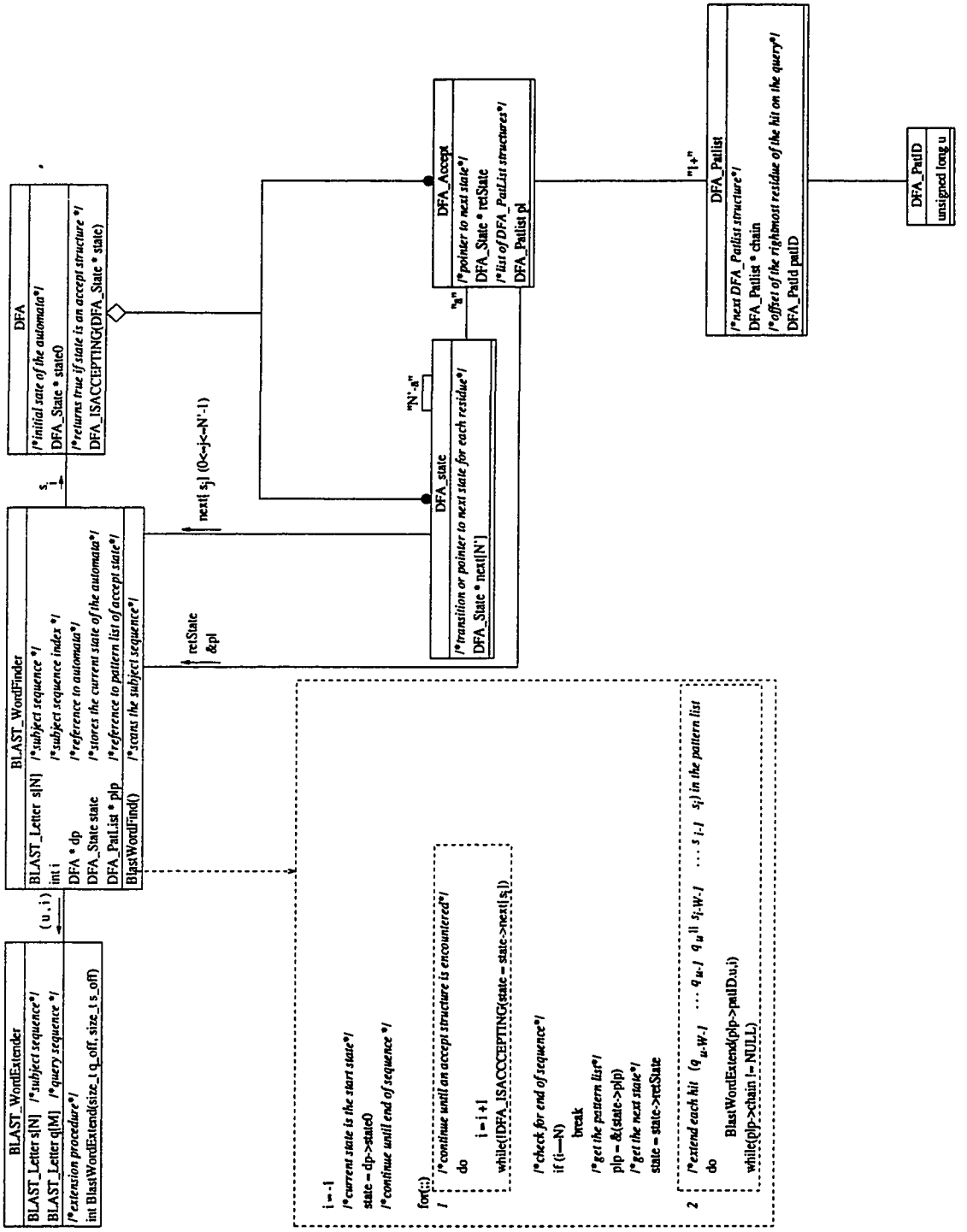

Figure 18: *Sub-Architecture that Implements the Hit Detection Step.*

3.2.4.3.3 Extending a Word Hit The extension algorithm is contained in the `BlastWordExtender` structure shown in Figures 19 and 20. The parenthesised numbers in the following description refer to the lines of code in the latter figure. This structure contains the query and subject sequences, `q` and `s`, the residue pair score `matrix`, a structure in which to store high scoring alignments, `hsp` and the procedure `BlastWordExtend`. An extension is a traversal of an alignment in both the left (41-49) and right (63-71) directions. A sub-extension is a traversal in a particular direction which commences from the delimiters of the word hit and proceeds in steps of residue pairs whose scores are accumulated in `sum` (17,43,65). Residue pair scores are obtained from `matrix` (16,42,64). A net positive `sum` is added to the alignment `score` after which `sum` is reset to zero. A sub-extension terminates when `sum` falls below `x` (49,71), whose value is determined by `SetFallOff` (25,48,70). Under certain conditions an extension may consist of more than one iteration of the left-right sub-extensions (50,74) as shown in Figure 21, but in most cases an extension consists of a single iteration. The delimiters of the aligned query segment are stored as offsets in `q_beg` and `q_end` (46,68). The corresponding offsets on the subject segment are obtained by adding `diag` (1,88,89). The offsets of the residue pair whose score gives `sum` a net positive value become the new alignment delimiters. The first loop (15-24) determines the delimiters of the maximal scoring sub-alignment of the word hit. The hit has a net positive score of at least T. However, it may contain residue pairs whose scores are negative. The staring point of extension are the delimiters of the maximal scoring sub-alignment of the word hit. The offsets of the rightmost residue pair are stored in `q_pos` and `s_pos`(5,9). In case either conditions in (50) or (74) hold, the left and right sub-extension respectively store the following values. Let "direction" be either "left" or "right". The offsets of furthest extension are stored in `direction_q` and `direction_s` (52-53,72,76). These offsets are the starting points of a sub-extension (37-38,58-59). The last `sum` which causes the extension to terminate is stored in `direction_sum` (32,33,54,77). The score of the sub-extension is stored in `direction_score` (34,35,55,78).

A hit is extended only if it has not been encompassed by a previous extension. The array `Extent` stores the offset of the rightmost extension for each diagonal (82). The offset of the hit is compared to the entry indexed by `diag` (3). Extension proceeds only if the condition in (3) holds. An alignment whose score meets S is stored as a high scoring segment pair (84-89).

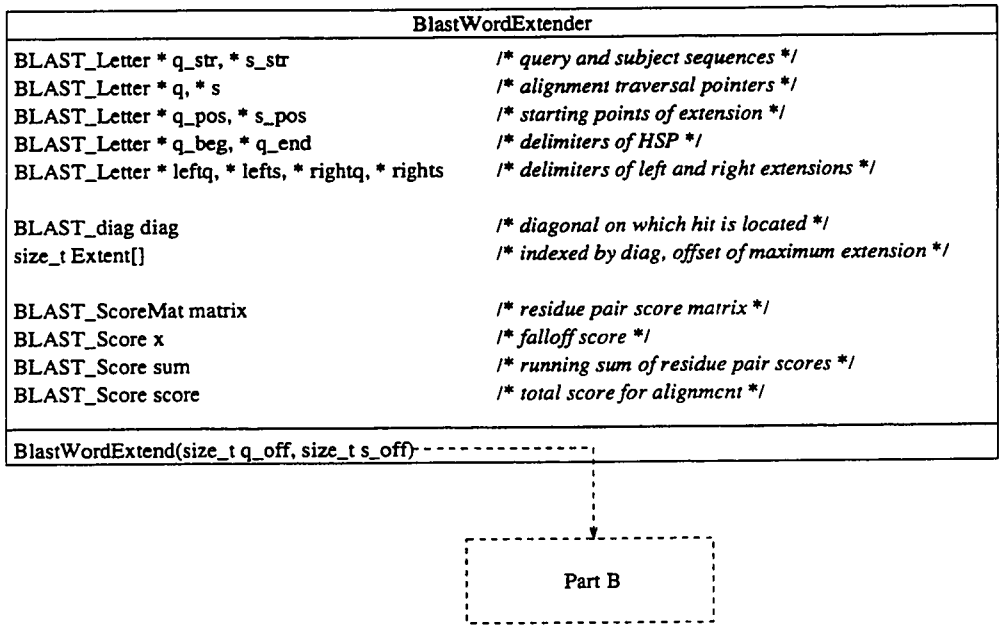

Figure 19: *Procedure that Implements the Extension Step - Part A.*

```
1   diag = s_off - q_off
2   /* check for previous extension through word hit */
3   if ! ( Extent[diag] > q_off )
4         /* locate end of word hit on query */
5         q_end = q = q_pos = q_str + q_off
6         /* locate start of word hit on query */
7         q_beg = q_end- W
8         /* locate end of word hit on subject*/
9         s = s_pos = s_str + s_off  - W
10        /* set intial values for score and sum */
11        score = 0
12        sum = 0
13        /* determine starting points of extension - where
14        positive scoring starts and ends on word hit */
15        do
16              pair_score = matrix[*q++][*s++]
17              sum += pair_score
18              if ( sum > score )
19                    score = sum
20                    q_end = q
21              else if ( sum <= 0)
22                    sum =0
23                    q_beg = q
24        while ( q < q_pos )
25        x = SetFallOff(score)
26
27        /* set initial values for extension variables */
28        leftq = q_end
29        rightq = q_beg
30        lefts = s_pos - (q_pos - leftq)
31        rights = s_pos + (rightq - q_pos)
32        leftsum = 0
33        rightsum = 0
34        leftscore = 0
35        rightscore = 0
36        /* set start parameters for left extension */
37        q = leftq
38        s = lefts
39        sum = leftsum
40        Extend_Left:    /* left extension loop */
41        do
42              pair_score = matrix[*--q][*--s]
43              sum += pair_score
44              if ( sum > score )
45                    score = sum
46                    q_beg = q
47                    sum = 0
48                    x = SetFallOff(score)
49        while ( sum >= x )
50
51        if ( Conditions_Right_Extend() )
52              /* save values of left extension parameters */
53              leftq = q
54              lefts = s
55              leftsum = sum
56              leftscore = score
57
58              /* set start parameters for right extension values */
59              q = rightq
60              s = rights
61              sum = rightsum
62
63              /* right extension loop */
64              do
65                    pair_score = matrix[*q++][*s++]
66                    sum += residue_pair
67                    if ( sum > score )
68                          score = sum
69                          q_end = q
70                          sum = 0
71                          x = SetFallOff(score)
72              while ( sum >= x )
73              rightq = q
74
75              if ( Conditions_Left_Extend() )
76                    /* save values of right extension parameters*/
77                    rights = s
78                    rightsum = sum
79                    rightscore = score
80                    goto Extend_Left
81
82        /* save offset of rightmost extension */
83        Extent[diag] = rightq - q_str
84        /* record HSP */
85        if ( score >= S)
86              hsp.score = score
87              hsp.start_query = q_beg - q_str
88              hsp.end_query = q_end - q_str
89              hsp.start_subject = hsp.start_query + diag
              hsp.end_subject = hsp.end_query + diag
```

Figure 20: *Procedure that Implements the Extension Step - Part B.*

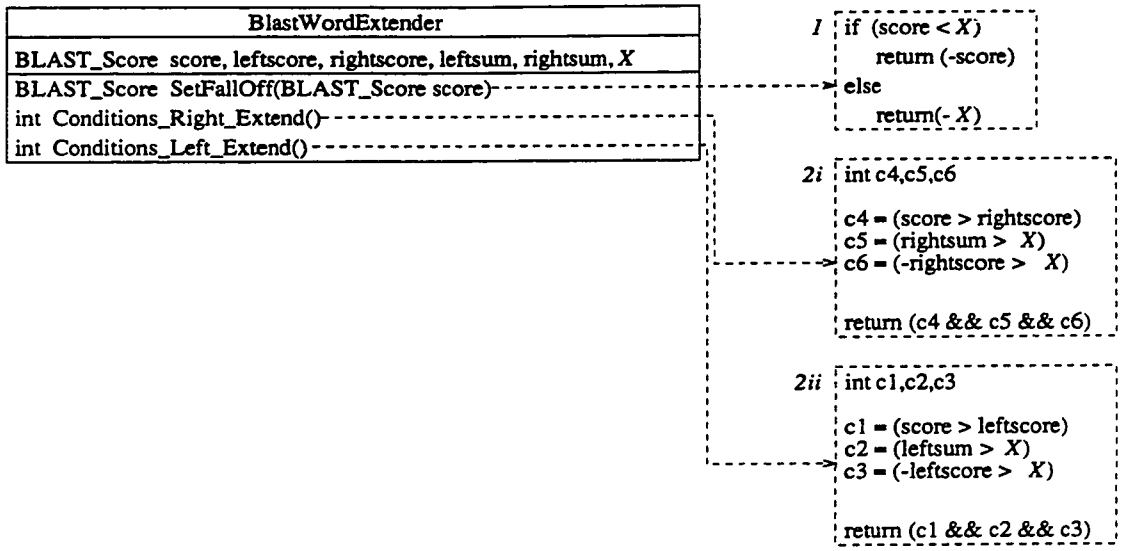

Figure 21: *Extensions that are Comprised of Multiple Left-Right Iterations.* By definition an extension terminates in a particular direction when `sum >= x`. However, there is a bias toward longer extensions; an extension in which `score` $< X$ uses a falloff value `x = -score`. Here, there may be more than one iteration of the left-right sequence. After the first right sub-extension, the left sub-extension may continue if three conditions, `Conditions_Left_Extend`, hold: (c1) The score must have increased in the right extension, otherwise `x` cannot more negative in the left-sub-extension. (c2) The last `sum` must not be more negative than `x`, otherwise, although c1 may hold, the last `sum` will be re-computed causing the extension to terminate. (c3) The value of `-leftscore` cannot be more negative than X, otherwise `x` is already at its most relaxed value. Likewise, right extensions may be continued under similar conditions , `Conditions_Right_Extend` (c4,c5,c6) which always evaluates to true prior to the first right sub-extension.

Chapter 4

Algorithm Optimisations

4.1 Introduction

This chapter describes the design and implementation of three optimisations of the BLASTP algorithm that enhance the overall time performance of the program. The profile results described in subsection 3.2.4.1 demonstrate that the extension step of the algorithm, implemented in procedure `BlastWordExtend`, accounts for approximately 90% of the program's execution time. The optimisations described in this chapter focus on the this step and the `BlastWordExtend` procedure.

The optimisations are of two types: (1) new sequence representations that facilitates extension and are used only in the extension procedure; and (2) restricting the number of calls to the extension procedure. The first two optimisations are of the first type while the third is of the second type. The first optimisation represents the query as a sequence of memory addresses. These addresses are those of the rows in the score matrix whose indices correspond to particular residues. Employing this representation decreases the number of operations required to access the matrix, thus reducing the overall time to perform an extension. The second optimisation represents the query and subject sequence as a sequence of residue-doublets. A doublet consists of two adjacent residues. Integers in the range 0-399 are used to represent the alphabet of residue-doublets; there are 20 residues in the alphabet, therefore there are 400 residue pairs. This representation facilitates the extension of word hits in steps of residue-doublet pairs instead of residue pairs. This reduces the number of iterations

of the extension loop required to perform an extension, thus reducing the overall time for extensions. The third optimisation constrains the number of invocations of the extension procedure. The scanning procedure counts the number of word hits per aligned segment of the query and subject sequences and invokes the extension procedure only if the number of hits per segment meets a threshold criteria. The overall time for extensions is reduced since the procedure is invoked less frequently.

Each optimised algorithm is described in a subsection of this chapter. Each subsection is composed of four parts: (1) A conceptual description of the optimisation and its potential to improve performance, (2) A description of the design and implementation of the optimised algorithm in terms of the program structures that were augmented with new functionality, (3) A performance comparison between the optimised and unmodified algorithms using CPU time and wall clock time for the programs and CPU time for the scanning and extension procedures and (4) A discussion of the effect of the optimisation on the detection of *HSPs* compared to the unmodified algorithm.

4.2 Row-Address Sequence Representation

4.2.1 Row-Address Concept

The profile analysis, subsection 3.2.4.1, revealed that one procedure, `BlastWordExtend`, accounts for greater than 90% of the program's execution time. Furthermore, within that procedure, the lines of code that access the residue pair score matrix account for 63% of the program's execution time.

The logical instruction sequence for the code fragment that accesses the score matrix is given in Figure 22. Two address calculations are executed; one to calculate the address of the matrix row and the other for the matrix entry (lines 2 and 4 respectively). The optimised algorithm lifts instruction 2 outside of the extension procedure, thus removing an instruction from a frequently executed line of code. The query sequence is represented as a sequence of matrix row-addresses instead of matrix row indices (Figure 23). In the optimised extension procedure, instructions 1 and 2 are equivalent to one instruction in which the row-address corresponding to a particular residue is obtained by dereferencing the query traversal pointer q. It is cost effective to use such a representation for the query since there is only one query per database search and that same sequence is traversed in the extension procedure for each scan. Using an equivalent representation for the subject sequences would not be cost effective since each subject would need to be translated into the row-address representation. Any increase in extension performance would be offset by the time required to perform this translation for each subject.

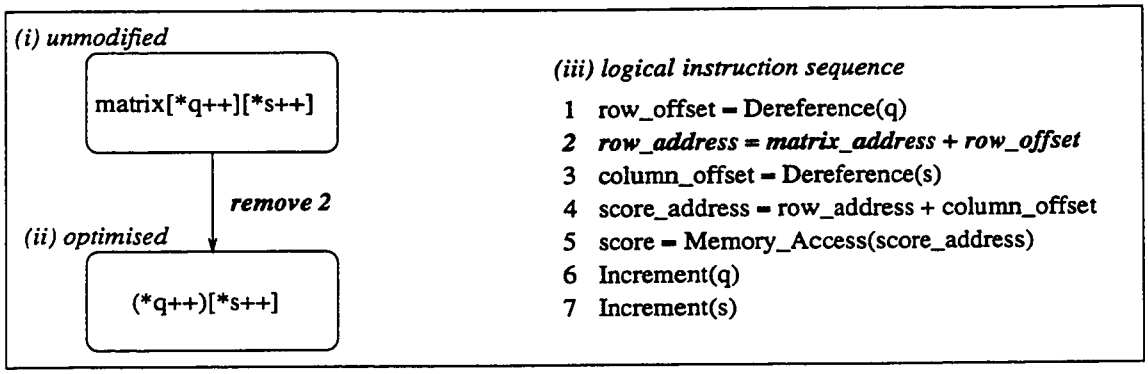

Figure 22: *Logical Instruction Sequence for Retrieval from Score Matrix.* In the unmodified algorithm a residue is represented as an index of the score matrix; i.e. amino acids are represented with integers in the range 0-19. Given that q and s are references to residues on the query and subject, the repeatedly executed line of C code that retrieves a score from the matrix is shown in *part (i).* The logical instruction sequence for this line is shown in *Part (iii).* Each scan consists of comparing the same query to a new subject. Instruction 2 can be lifted out of this line and done once per database search. Therefore, query residues are represented as row-addresses of the score matrix. The optimised extension procedure uses a query sequence comprised of row-addresses which are offset by the subject index to obtain the matrix entry *Part (ii)* .

Given the sequence of residues,
$s = s_0 s_1 s_2 \ldots s_n$ where for each s_i , min <= i <= max

Let M be a function that returns the memory address of a particular row in the residue pair score matrix.

Let s_a be the address representation of the sequence.

Then, $s_a = M(s)$
$s_a = M(s_0), M(s_1), M(s_2), \ldots, M(s_n)$
s_a = matrix[s_0], matrix[s_1],matrix[s_2], . . .,matrix[s_n]

Figure 23: *Row-Address Algorithm.* The BLASTP program represents amino acid residue as integers which are the row indices of the residue pair score matrix. Proteins are represented as sequences of row indices. Alternatively, residues can be represented as addresses which point to the particular matrix row indexed by that residue. Proteins are thus represented as sequences of matrix row-addresses. The largest and smallest integers in the residue symbol alphabet are `max` and `min` respectively. The residue pair score matrix is `matrix`.

4.2.2 Row-Address Design and Implementation

The BLAST_WFContext structure (Figure 24) was augmented with a member that stores the residue pair score matrix row-address representation of the query sequence, q_ra. The procedure, `Calculate_Row-Address-Rep`, constructs an equivalent sequence of row-addresses from the original sequence of row-indices.

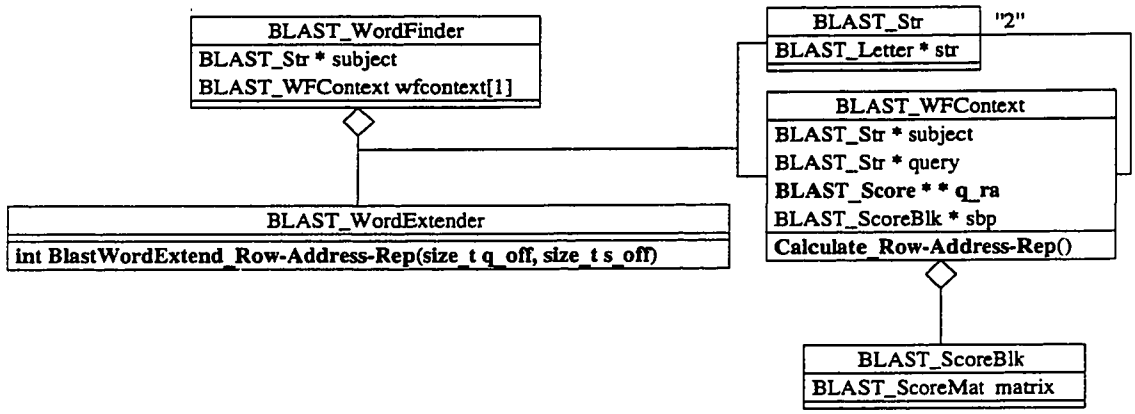

Figure 24: *Modified Sub-Architecture that Implements the Row-Address Algorithm.*

The optimised extension procedure (Figures 25 and 26) differs from the original in three aspects: First, members such as q, q_beg and q_end which reference a query residue are of type *pointer to pointer*. In the unmodified algorithm these members were of type *pointer to* `BLAST_Letter`, which is an index of the matrix. In the optimised algorithm, these members are of type *pointer to pointer to* `BLAST_Score` since a query residue is a row-address. Second, the lines of code that retrieve a score from the matrix (Figure 26, lines 19, 45 and 67) merely offset the address of the row by the index of a column which represents a subject residue. In the unmodified algorithm the address of the matrix was offset by the index of a row, which represented a query residue, to obtain the address of the row. This address was then offset by the index of the column to obtain the matrix entry or score. Finally, it is interesting to note that the address of the score matrix is not directly used in the optimised procedure.

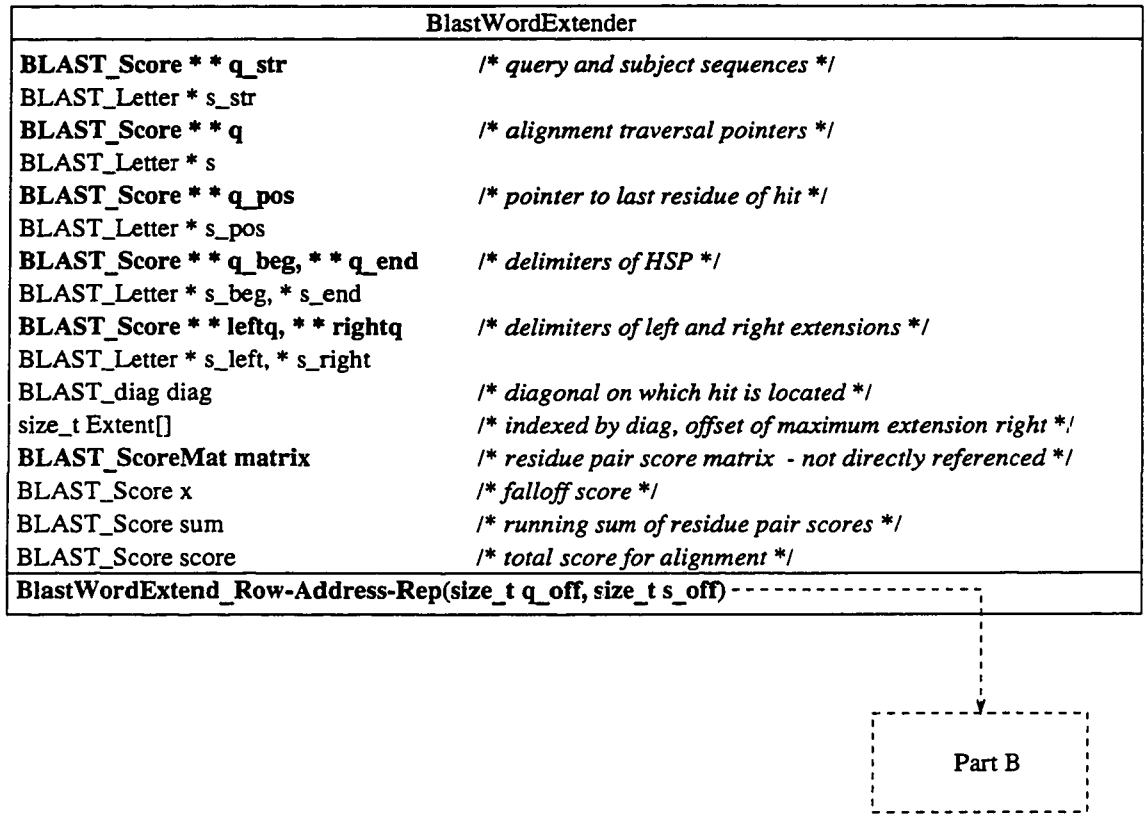

Figure 25: *Optimised Extension Procedure that Implements the Row-Address Algorithm - Part A.*

```
1   diag = s_off - q_off
2   /* check for previous extension through word hit */
3   if ! ( Extent[diag] > q_off )
4
5       /* locate end of word hit on query */
6       q_end = q = q_pos = q_str + q_off - W
7       /* locate start of word hit on query */
8       q_beg = q_end - W
9       /* locate end of word hit on subject*/
10      s = s_pos = s_str + s_off
11
12      /* set intial values for score and sum */
13      score = 0
14      sum = 0
15
16      /* determine starting points of extension - where
17      positive scoring starts and ends on word hit */
18      do
19          pair_score = (*q++)[*s++]
20          sum += pair_score
21          if ( sum > score )
22              score = sum
23              q_end = q
24          else if ( sum <= 0)
25              sum =0
26              q_beg = q
27      while ( q < q_pos )
28      x = SetFallOff(score)
29
30      /* set initial values for extension variables */
31      leftq = q_end
32      rightq = q_beg
33      lefts = s_pos - (q_pos - leftq)
34      rights = s_pos + (rightq - q_pos)
35      leftsum = 0
36      rightsum = 0
37      leftscore = 0
38      rightscore = 0
39      /* set start parameters for left extension */
40      q = leftq
41      s = lefts
42      sum = leftsum
43      Extend_Left:    /* left extension loop */
44      do
45          pair_score = (*--q)[*--s]
46          sum += pair_score
47          if ( sum > score )
48              score = sum
49              q_end = q
50              sum = 0
51              x = SetFallOff(score)
52      while ( sum >= x )
53      if ( Conditions_Right_Extend() )
54          /* save values of left extension parameters */
55          leftq = q
56          lefts = s
57          leftsum = sum
58          leftscore = score
59
60          /* set start parameters for right extension values */
61          q = rightq
62          s = rights
63          sum = rightsum
64
65          /* right extension loop */
66          do
67              pair_score = (*q++)[*s++]
68              sum += residue_pair
69              if ( sum > score )
70                  score = sum
71                  q_end = q
72                  sum = 0
73                  x = SetFallOff(score)
74          while ( sum >= x )
75          rightq = q
76
77          if ( Conditions_Left_Extend() )
78              /* save values of right extension parameters */
79              rights = s
80              rightsum = sum
81              rightscore = score
82              goto Extend_Left
83
84      /* save offset of rightmost extension */
85      Extent[diag] = rightq - q_str
86      /* record HSP */
87      if ( score >= S)
88          hsp.score = score
89          hsp.start_query = q_beg - q_str
90          hsp.end_query = q_end - q_str
91          hsp.start_subject = hsp.start_query + diag
92          hsp.end_subject = hsp.end_query + diag
```

Figure 26: *Optimised Extension Procedure that Implements the Row-Address Algorithm - Part B.*

4.2.3 Row-Address Results

A program that employs the optimised algorithm uses approximately 14% less cycles than a program that uses the unmodified algorithm (Table 3). This saving is realized in the time spent in the `BlastWordExtend` procedure (Table 4). In particular, it is realized in the lines that access the residue pair score matrix (Table 5).

Program	CPU cycles x 10^{10}	CPU time (s)
BLASTP (row-address)	1.8	62.9
BLASTP (unmodified)	2.1	73.4

Table 3: *Comparison of BLASTP Program Time.* A program implementing the optimised algorithm uses approximately 14% less CPU cycles. There is an equivalent savings in CPU time.

Procedure	CPU cycles x 10^{10}	CPU time (s)
BlastWordExtend_Row-Address-Rep	1.7	57.9
BlastWordExtend	2.0	68.5

Table 4: *Comparison of Extension Procedure Time.* The optimised extension procedure uses 15% less CPU cycles than the unmodified procedure, BlastWordExtend. There is an equivalent savings in CPU time.

Code	Reference to Figure 26	net CPU cycles x 10^{10}
location of hit on query	6,8	- 0.13
start extension loop	18-27	+ 0.13
left extension loop	44-52	+ 0.56
right extension loop	66-74	+ 0.63
Total Net CPU Cycles		1.19

Table 5: *Net CPU cycle gain by BlastWordExtend_Row-Address-Rep over BlastWordExtend.* The comparison is based on the lines of code that were optimised to implement the row-address algorithm. The savings are realized in the three extension loops: `start`, `left` and `right`.

4.2.4 Row-Address Effect on HSP Detection

The row-address algorithm does not effect the sensitivity of the search. The optimised algorithm detects the same *HSPs* and their respective scores that the unmodified algorithm detects. This optimisation is in fact more of a change to the algorithm's implementation than a change to its heuristics.

4.3 Residue-Doublet Sequence Representation

4.3.1 Residue-Doublet Concept

The unmodified extension algorithm works in steps of aligned residue pairs. The optimised algorithm performs extensions in steps of aligned residue-doublet pairs (Figure 27). This requires that sequences be represented by an alphabet of residue-doublets. A sequence comprised of residues is logically equivalent to two residue-doublet sequences (Figure 28). The reason for having two representations becomes apparent in the context of mapping a particular point within an alignment of two residue sequences to the corresponding point within the equivalent residue-doublet alignment (Figure 29). As a consequence of representing sequences as residue-doublets, each extension cycle accumulates scores of residue-doublet pairs instead of scores of residue pairs. These scores are obtained from a doublet pair score matrix which is accessed in each iteration of the extension loop.

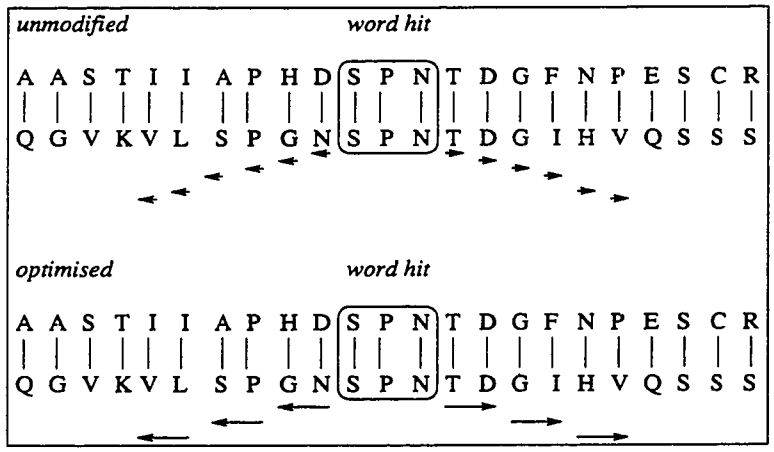

Figure 27: *Extending in Steps of Two Residue-Doublet Pairs.* In the unmodified algorithm, alignments are constructed by extending from the word hit in both directions in steps of aligned residue pairs. In each step, a score is looked up in a residue pair score matrix and added to a cumulative **sum**. An alternative is to proceed in steps of aligned residue-doublet pairs. A residue-doublet representation is used for sequences. In each extension step of the optimised algorithm, a score is looked up in a residue-doublet pair score matrix and summed.

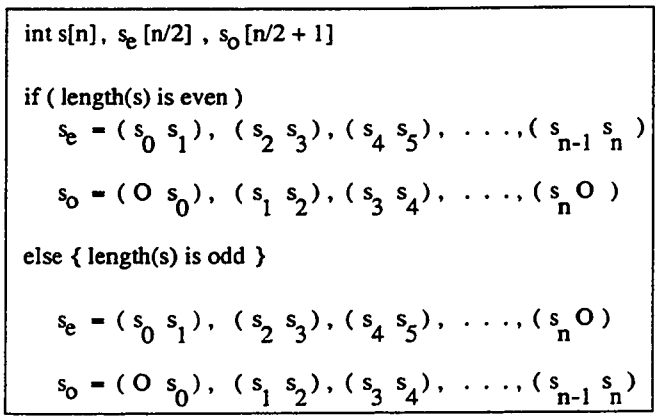

Figure 28: *Sequences of Residues Represented as Sequences of Residue-Doublets.* A sequence comprised of single residues, s, is logically represented by two sequences comprised of residue-doublets; s_e and s_o. Each residue-doublet in s_e is of the form (s_i,s_{i+1}), where i is even. Similarly, for s_o, each residue-doublet is of the form (s_i,s_{i+1}), where i is odd. The null residue, O, may be appended to either the beginning and/or end of s to ensure that s is evenly divisible into residue-doublets.

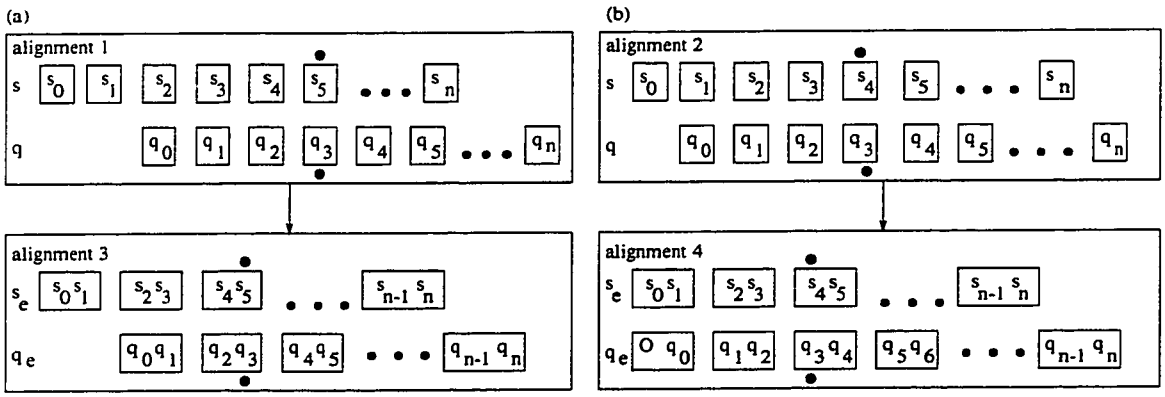

Figure 29: *Mapping of Word Hit from Residue Sequence Alignment to Equivalent Residue-Doublet Alignment.* The lengths of the sequences, s and q, are n and m and are assumed to be even. The darkened circles mark the offset of the last residue on the word hit. (a) In alignment 1, s and q are aligned along an even diagonal (diag = 5-3 = 2). The equivalent alignment of the corresponding residue-doublet sequences is alignment of s_e and q_e - alignment 3. (b) In alignment 2, s and q are aligned along an odd diagonal (diag = 4-3 = 1) on which a hit is located. The equivalent alignment of the corresponding residue-doublet sequences is that of s_e and q_o - alignment 4. In summary, a hit located on an even diagonal, as in (1), requires s_e to be aligned with q_e while a hit on an odd diagonal, as in (2) requires s_e to be aligned with q_o.

4.3.2 Residue-Doublet Design and Implementation

The program structures that implement the residue-doublet algorithm are shown in Figure 31. The program was augmented with functionalities to (1) compute residue-doublet sequences for the query and subject, (2) access a residue-doublet database, (3) access a residue-doublet pair score matrix and (4) extend residue-doublet alignments.

1. Functionality was added to compute the residue-doublet representations of the query and subject. A database was created to store a residue-doublet representation for each subject in the protein sequence database, `db.seq_rd`. In addition, summary information about the database was calculated and stored in `db.atb_rd`. These data include the number of residue-doublet sequences, `count_rd`, the length of the longest sequence, `mxlen` and an index, `seq_beg_rd`, which stores the offset into `db.seq_rd` of the beginning of each sequence. This was done by a modified `setdb` program (Figure 32) which implements an algorithm that constructs the residue-doublet sequences (Figure 33). This algorithm was also implemented in the procedure `Construct_Residue_Doublet_Sequences` to derive the two residue-doublet representations of the query; `str_rd_even` and `str_rd_odd`. It is only necessary to store two residue-doublet representations for either the query or the subject, but not for both. The reason, as illustrated in Figure 30, is that the four possible alignments of two residue-doublet sequences fall into two equivalence classes. The subject is the obvious choice to be represented with one residue-doublet sequence for reasons of space efficiency; i.e. there is a database of subjects, but only one query.

2. The program's interface to the database was expanded with functionality to retrieve the residue-doublet subject. The structures `DBFile` and `BDBFile` were each augmented with procedures to re-position the file pointer for `db_seq_rd`, `db_seek_rd` and `blast_db_seek_rd` respectively, and to retrieve a sequence, `db_get_seq_rd` and `blast_db_get_seq_rd` respectively.

3. An algorithm (Figure 34) was implemented in the procedure `Calculate_Residue-Doublet Matrix` to derive the residue-doublet pair score matrix from a residue pair score matrix. The residue-doublet matrix was computed prior to searching and loaded at run-time. This procedure is shown in Figure 31 as a logical component of the `BLAST_ScoreBlk` structure.

4. An extension algorithm that performs extensions in steps of residue-doublet pairs was designed (Figures 35 and 36) and implemented in the procedure `BlastWordExtend_Residue-Doublet`. In this optimised extension procedure, an aligned pair of residue-doublet sequences is traversed (`seq_rd` and `s_seq_rd`). Residue-doublets are integer types, therefore references to members of a sequence such as `q` and `s` are of type `int *`. The matrix, `matrix_rd`, contains a score for each possible pairing of residue-doublets. It's size is N^4 where N is the number of residues in the alphabet. Further differences from the unmodified procedure are listed below with references to lines of code in Figure 36:

   (a) The offset of the hit on each residue-doublet sequence is calculated (5-8).

   (b) The appropriate residue-doublet query sequence is selected based on the parity of the diagonal (11-14).

   (c) The delimiters of the word hit on the residue-doublet sequences are calculated using the offsets from (a) (17,19).

   (d) Scores are obtained from `matrix_rd` (25,52,74).

   (e) The offset of the maximum extension on the residue query is calculated from the equivalent offset on the residue-doublet query (92).

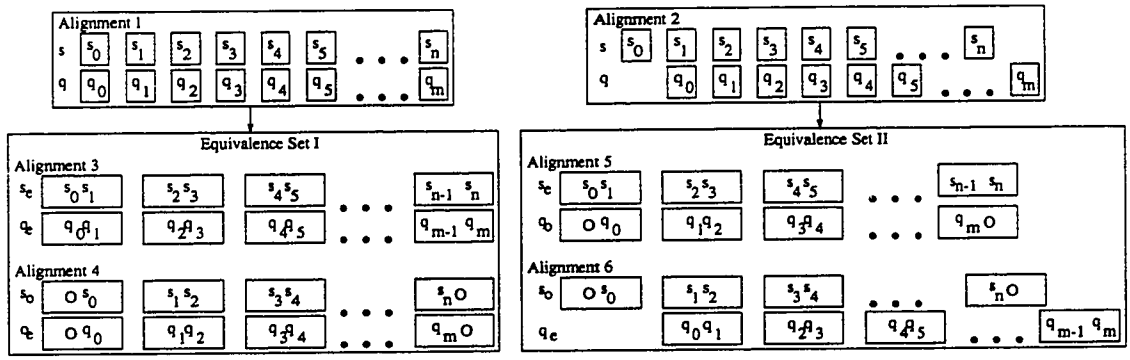

Figure 30: *Equivalence of Residue-Doublet Alignments.* Assume n and m to be even. An alignment of two residue sequences can be aligned along either an even or odd diagonal; alignments 1 and 2 respectively. An alignment along an even diagonal can be represented by either residue-doublet alignment in the first equivalence set; an alignment along an odd diagonal by either alignment in the second set.

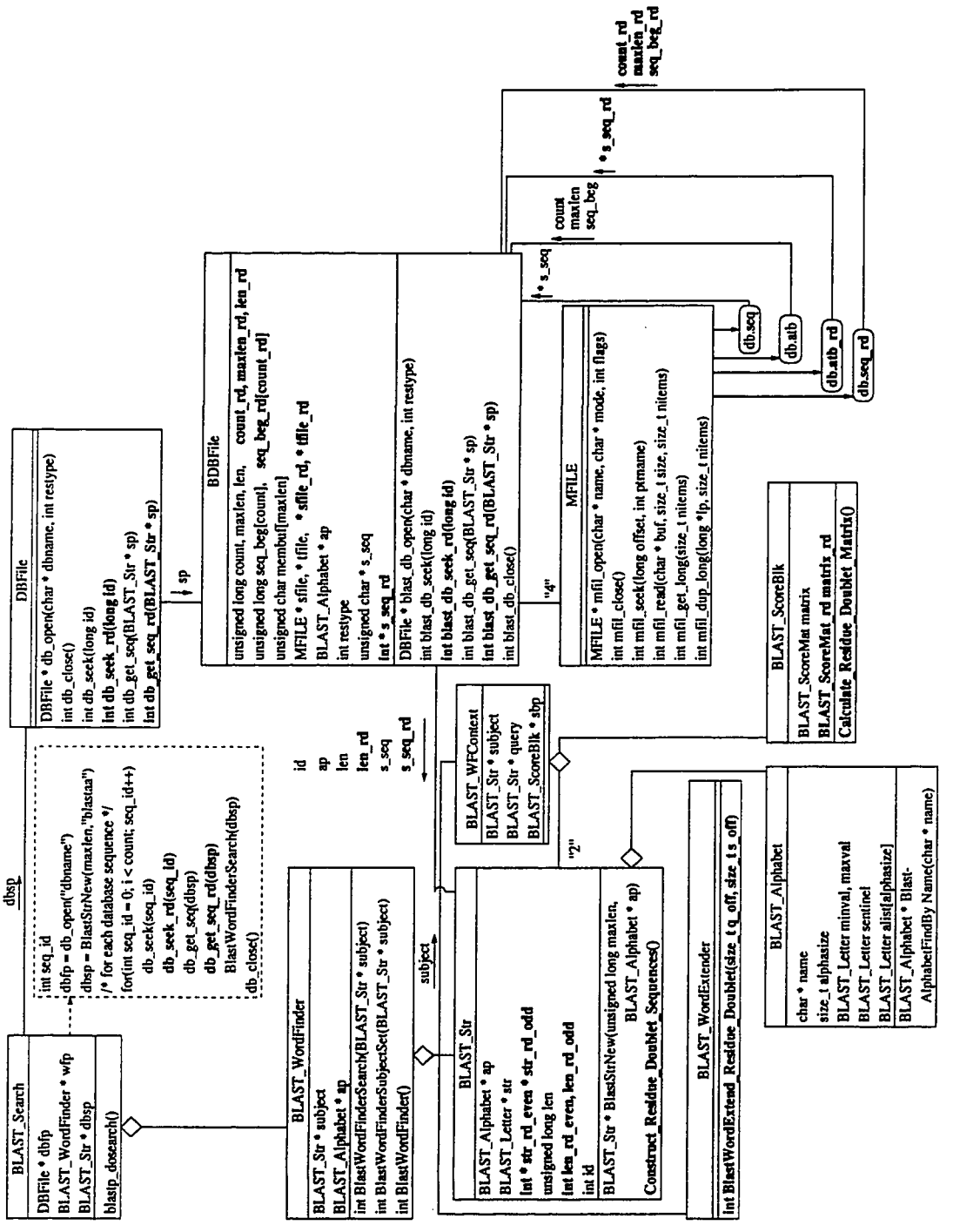

Figure 31: *Modified Event Loop Sub-Architecture that Implements the Residue-Doublet Algorithm.*

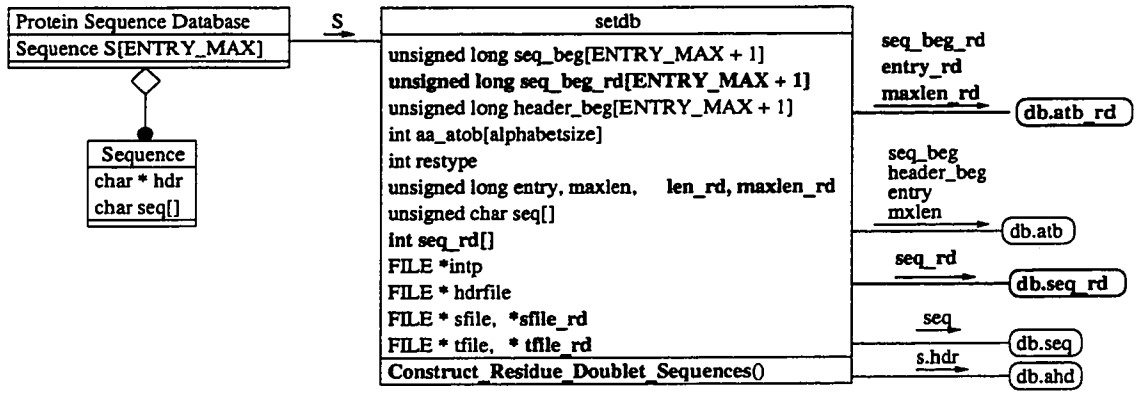

Figure 32: *Modified* `setdb` *Program for Residue-Doublet Algorithm.* The algorithm is implemented by the procedure `Construct_Residue_Doublet_Sequences`. The sequence is temporarily stored in `seq_id` then written to `db.seq_rd`. Upon writing, the offsets into `db.seq_rd` of the beginning and end of each sequence are recorded in `seq_beg_rd`. After scanning the protein sequence database, `setdb` records summary information about the residue-doublet database in `db.atb_rd`. These data include the index, `seq_beg_rd`, the total number of sequences, `entry_rd` and the length of the longest sequence, `mxlen_rd`.

```
1  Residue-Doublet ( s )
2     /* Length Calculations */
3        l_s = length(s)
4        l_s_e = (l_s / 2) + 2
5        l_s_o = ((l_s / 2) + 1) + 2
6     /* Allocate Memory for residue-doublet sequences */
7        s_e = allocate(l_s_e )
8        s_o = allocate(l_s_o )
9     /* Sentinels mark the beginning of residue-doublet sequences */
10       s_e [ ec++ ] = Sentinel
11       s_o [ oc++ ] = Sentinel
12    /* s_o always starts with the null residue */
13       s_o [oc++ ] = (O , s_0 )
14    /* build the residue-doublet sequences */
15       for i = 0 to l_s - 1
16          if (i mod 2) == 1
17             s_e [ec++ ] = (s_i , s_i+1 )
18          else   /* (i mod 2) == 0 */
19             s_o [oc++ ] = ( s_i , s_i+1 )
20    /* add null residue to end of either s_e or s_o */
21       if ( l_s mod 2 ) == 1
22          s_o [oc++ ] = (s_n , O )
23       else { ( l_s mod 2 ) == 0 }
24          s_e [ec++ ] = (s_n , O )
25    /* Sentinels mark the end of the residue-doublet sequences */
26       s_e [ ec++ ] = Sentinel
27       s_o [ oc++ ] = Sentinel
```

Figure 33: *Residue-Doublet Algorithm.* The algorithm derives the two residue-doublet sequences, s_e and s_o, from the residue sequence, s. The algorithm traverses s, computes the representation for the residue-doublet (s_i,s_{i+1}) and appends it to either s_e or s_o

```
1   Residue-Doublet_Matrix( BLAST_Score * m )
2
3       const N                 /* number of residue symbols in amino acid alphabet */
4       BLAST_Score * m         /* the residue pair score matrix */
5       BLAST_Score * dm        /* the residue-doublet pair score matrix */
6       r1, r2, c1, c2          /* row, r, and column, c, counters for m */
7       dr, dc                  /* row, dr, and column, dc, counters for dm */
8
9       /* allocate space for residue-doublet pair score matrix */
10      dm = [ N^2 ][N^2 ]
11      /* Build residue doublet pair score matrix */
12      for ( c1 = 0 to N-1 )
13              for ( c2 = 0 to N-1 )
14                      /* at the beginning of each row, set dc to 0 */
15                      dc = 0
16                      for ( r1 = 0 to N-1 )
17                              for ( r2 = 0 to N-1 )
18                                      dm[ dr ] [ dc ] = m[ r1 ] [ c1 ] + m[ r2 ] [ c2 ]
19                                      dc++
20              dr++
```

Figure 34: *Residue-Doublet Matrix Algorithm.* The algorithm constructs the residue-doublet pair score matrix *dm* from the residue pair score matrix m; in this implementation m is the BLOSUM62 residue pair score matrix. Let q and s be the query and subject sequence respectively. Then, the score of a particular residue-doublet pairwise alignment, $(q_i,q_j)\|(s_k,s_l)$ is the sum of the scores of the residue-residue pairwise alignments, $(q_i\|s_k) + (q_j\|s_l)$.

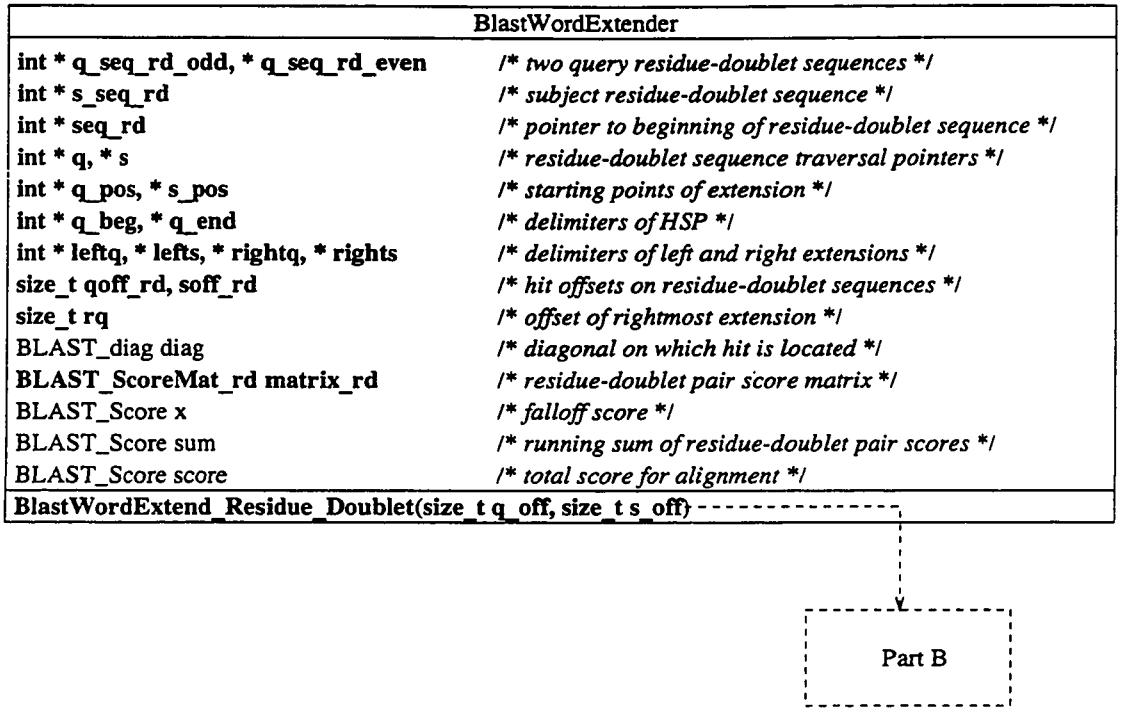

Figure 35: *Optimised Extension Procedure that Implements the Residue-Doublet Algorithm - Part A.*

```
1   diag = s_off - q_off
2   /* check for previous extension through word hit */
3   if ! ( Extent[diag] > q_off ) {
4         /* calculate hit offsets on rd sequences */
5         qoff_rd  = (q_off/2)+1
6         soff_rd = (s_off/2)+1
7         if((diag%2) != 0) && ((q_off%2) != 0))
8               qoff_rd++;
9
10        /* select appropriate rd sequence */
11        if  ( (diag%2)==0)
12              seq_rd = q_seq_rd_odd
13        else
14              seq_rd = q_seq_rd_even
15
16        /* locate beginning and end of word hit on query */
17        q_beg = q =  (q_end  = q_pos = seq_rd + qoff_rd) - ( W/2)
18        /* locate beginning and end of word hit on subject */
19        s_beg = s = (s_end = s_pos = s_seq_rd + soff_rd) - (W/2)
20
21        score = 0
22        sum = 0
23        /* determine starting points of extension */
24        do
25              pair_score_rd = matrix_rd[*q++][*s++]
26              sum += pair_score_rd
27              if ( sum > score )
28                    score = sum
29                    q_end = q
30              else if ( sum <= 0)
31                    sum =0
32                    q_beg = q
33        while ( q < q_pos )
34        x = SetFallOff(score)
35
36        /* set initial values for extension variables */
37        leftq = q_beg
38        rightq = q_end
39        lefts = s_pos - (q_pos - leftq)
40        rights = s_pos + (rightq - q_pos)
41        leftsum = 0
42        rightsum = 0
43        leftscore = 0
44        rightscore = 0
45        /* set start parameters for left extension */
46        q = leftq
47        s = lefts
48        sum = leftsum
49
50        Extend_Left:    /* left extension loop */
51        do
52              pair_score_rd = matrix_rd[*--q][*--s]
53              sum += pair_score_rd
54              if ( sum > score )
55                    score = sum
56                    q_end = q
57                    sum = 0
58                    x = SetFallOff(score)
59        while ( sum >= x )
60        if ( Conditions_Right_Extend() )
61              /* save values of left extension parameters */
62              leftq = q
63              lefts = s
64              leftsum = sum
65              leftscore = score
66
67              /* set start parameters for right extension values */
68              q = rightq
69              s = rights
70              sum = rightsum
71
72              /* right extension loop */
73              do
74                    pair_score_rd = matrix_rd[*q++][*s++]
75                    sum += residue_pair_rd
76                    if ( sum > score )
77                          score = sum
78                          q_end = q
79                          sum = 0
80                    x = SetFallOff(score)
81              while ( sum >= x )
82              rightq = q
83
84              if ( Conditions_Left_Extend() )
85              /* save values of right extension parameters */
86                    rights = s
87                    rightsum = sum
88                    rightscore = score
89                    goto Extend_Left
90
91        /* save offset of rightmost extension */
92        Extent[diag] = (rq = rightq - q_seq_rd) + rq
93        /* record HSP */
94        if  ( score >= S)
95              hsp.score = score
96              hsp.start_query = q_beg - q_str
97              hsp.end_query = q_end - q_str
98              hsp.start_subject = hsp.start_query + diag
99              hsp.end_subject = hsp.end_query + diag
```

Figure 36: *Optimised Extension Procedure that Implements the Residue-Doublet Algorithm - Part B.*

4.3.3 Residue-Doublet Results

A BLASTP program implementing the residue-doublet extension algorithm uses approximately 48% less CPU cycles than the unmodified program (Table 6). The `BlastWordExtend_Residue_Doublet` procedure uses 50% less CPU cycles than the unmodified procedure (Table 7). Lines of code were added to the procedure, `BlastWordExtend`, to implement the residue-doublet extension algorithm. Table 8 shows that the cost of executing this additional code is insignificant in comparison to the savings obtained in performing extensions in steps of residue-doublet pairs. However, according to Table 6, this savings is not realized in wall clock time. The profiler `prof` with the `pixie` option measures only the cycles used by the application program. It does not report cycles spent by the operating system for such operations as memory accesses. The residue-doublet extension procedure repeatedly accesses a score matrix three orders of magnitude larger than the original residue pair score matrix. One possible explanation for this unrealized real-time savings is that the unmodified program can keep the score matrix cached throughout its entire execution. In contrast, at a particular point in the execution of the optimised program, only part of the residue-doublet pair score matrix can be resident in cache. Furthermore, the order of accesses seems random, causing the cache to be swapped frequently.

Program	CPU cycles x 10^{10}	CPU time(s)	wall clock time(s)
BLASTP (residue-doublet)	1.3	45.5	109
BLASTP (unmodified)	2.5	87.1	100

Table 6: *Comparison of BLASTP Program Time.* A program implementing the modified algorithm uses 48% less CPU cycles. There is an equivalent savings in CPU time. However, this savings is not realized in wall clock time, reported by the program as real time [4], page 17, which may vary between runs depending on the availability of the CPU for the BLASTP process. However, the times reported from different runs consistently show longer wall clock times for the residue-doublet program. This unrealized savings in real time is most likely due to the relative increase in memory access time. The residue-doublet score matrix is two orders of magnitude larger than the residue pair score matrix, BLOSUM62. Its size prohibits the possibility of it being cache resident in its entirety, thus requiring more accesses to memory to load requested blocks to cache.

Procedure	CPU cycles x 10^{10}	CPU time (s)
BlastWordExtend_Residue-Doublet	1.2	39.9
BlastWordExtend	2.4	81.5

Table 7: *Comparison of Extension Procedure Time.* The modified procedure uses 50% less CPU cycles than the unmodified one. There is an equivalent savings in CPU time.

No.	Code Fragment	Reference to Figure 36	net CPU cycles x 10^8
1	calculate rd offsets	5-8	-3.3
2	select rd query	11-14	-0.1
3	locate delimiters of hit	17,19	-0.04
4	start extension loop	24-33	+13.7
5	left extension loop	51-59	+578.7
6	right extension loop	73-81	+671.3
7	save rightmost offset	92	-1.8
	Total Net CPU Cycles		+1258.5

Table 8: *Net CPU Cycle Gain by BlastWordExtend_Residue-Doublet over BlastWordExtend.* The savings is realized in the three extension loops; `start`, `left` and `right`(4-6). There are fewer iterations of the extend loop since the alignment is traversed in steps of residue-doublet pairs. The cost of modifying the procedure is in the addition of code to do the following: (*1*) calculate the offset of the hit on the residue-doublet alignment (1), (*2*) select the appropriate residue-doublet query based on the parity of the diagonal (2), (*3*) locate the delimiters of the word hit on both sequences and (*4*) save the rightmost offset by mapping the delimiters of the *HSP* on the residue-doublet alignment to the residue alignment (7). However, the time lost executing these additional lines of code is insignificant compared to the time saved in doing less extension loops.

4.3.4 Residue-Doublet Effect on HSP Detection

The heuristic character of the residue-doublet algorithm is stronger than that of the unmodified algorithm. There are three anomalies in the ability of the residue-doublet algorithm to detect *HSPs* when compared to the unmodified algorithm. These arise directly from grouping residues into doublets and from performing extensions using sequences of residue-doublets.

1. *Lower initial falloff score.*

   The extension procedure begins by calculating the score of the maximal scoring sub-alignment of the word hit. This score is assigned to the initial falloff value `x`. If a word hit contains a negative scoring pair, it will not be added to the score of the sub-alignment. In the residue-doublet case, the negative scoring pair may be grouped with an adjacent positive scoring one. The score of the doublet is net positive, thus it is added to the sub-alignment. However, the negative scoring pair makes the overall score is lower, making the initial falloff score lower than would be the case in the unmodified algorithm. Since the falloff value is lower or more stringent, extensions may terminate earlier in the residue-doublet case. In the test search, some *HSPs* with $S < 50$ were missed by the residue-doublet algorithm that were found by the unmodified algorithm.

2. *Lower scores for HSPs.*

   Scores of *HSPs* are lower by approximately one to five points in the residue-doublet case for the same reason as described in (1). The maximal scoring local alignment is delimited by `q_beg` and `q_end`. The residue pair immediately left of `q_beg` or right of `q_end` is negative scoring and is the first pair of the sub-alignment that brings `sum` below `x`. This negative scoring sub-alignment is not part of an *HSP*. However, in the residue-doublet case, the negative scoring pair may be grouped with the positive scoring one, lowering the overall score. In the unmodified algorithm, the negative score would not have been added to the overall score. In the test search, this anomaly occurred frequently.

3. *Higher scores for negative scoring falloff sub-alignment.*

   This anomaly is the converse of the first two in the sense that it arises by an unwanted grouping of a positive scoring pair with a negative scoring one. The negative scoring residue pair causing `sum` to fall below `x` are located at positions `q_left` and `q_right - 1` for the left and right extensions respectively. However, in the residue-doublet case these negative scoring pairs may be grouped with positive scoring ones at positions causing `sum` to remain above `x`, thus allowing the extension to continue. In the test search, this anomaly occurred twice out of 452 *HSPs* with $S > 50$.

4.4 Two-Hit Detection

4.4.1 Two-Hit Concept

The third step of the BLASTP algorithm extends word hits to longer, potentially higher scoring, alignments. Recall that a word hit is a short alignment of length W which scores at least T. The extension algorithm uses the word hit as a seed and extends it to the left and right to determine the maximal scoring alignment relative to the word hit. The score of an alignment is the cumulative sum of the scores of its residue pairs, making the score a function of the alignments length. The extension algorithm has a hill climbing character. As the extension proceeds in a particular direction, net scores for both positive scoring and negative scoring sub-alignments are computed. Net positive scores are added to the total score for the extension in a particular direction. The extension in that direction terminates when a net negative score matches or falls below the falloff parameter X. The greater the number of net positive scoring sub-alignments within an alignment, the greater the probability that the alignment is an *HSP*. Recall an *HSP*, or high scoring segment pair, is an alignment whose score meets the threshold set by the cutoff parameter S.

Word hits are net positive scoring sub-alignments. The alignments that result from an extension contain the word hit from which it was seeded and possibly others. Word hits are detected during the scanning phase (step 2) of the algorithm. Therefore, during scanning it is possible to obtain a measure of the number of word hits located on an alignment. The data of Table 9 shows that the result of virtually all extensions are alignments that contain either one, two or three word hits. The average scores of these alignments rarely meet a score of 32, the default value for the cutoff parameter S (Figure 37). The extension time spent on those word hits that are extended into *HSPs* comprises a minute percentage of the total.

The two-hit algorithm employs a scanning step (step 2) that constrains the number of times that the extension procedure is invoked. An aligned segment of two sequences must contain two word hits within a given distance constraint. The distance constraint is the average length of a negative scoring sub-alignment whose score meets or falls below the falloff parameter X. The two-hit algorithm uses the heuristic

that only an alignment of a query and subject segment which contains two hits within a distance that is less than the average falloff distance is likely to be a sub-alignment of an *HSP*. The concept of the two-hit algorithm is demonstrated in Figures 38 and 39; the former shows the case where the scanning constraint is satisfied, the latter the case where it is not.

Hits/Alignment(n)	% Alignments with n Hits	% Extend Loop Cycles
1	75	71
2	21	24
3	3	5

Table 9: *%Extension Time as a Function of Number of Hits per Alignment.* Extension time is measured in number of executed extend loops. Alignments containing a single word hit account for 75 percent of all alignments and 71 percent of the extension time.

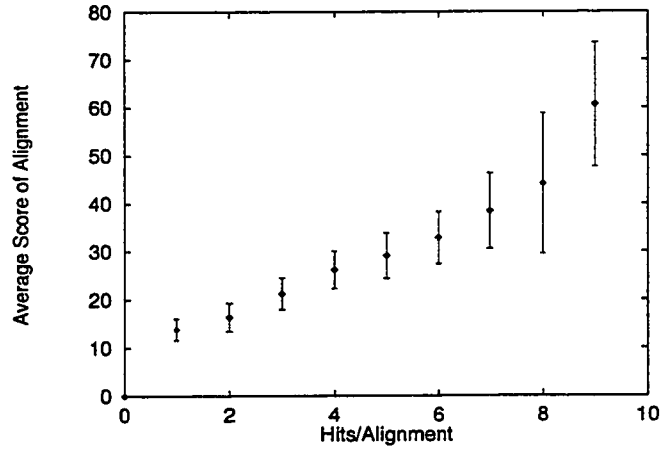

Figure 37: *Plot of Average Alignment Score v.s. Hits per Alignment.* The protein NRDB was scanned using the test search described in subsection 3.2.1. The set of data tuples of the form $< alignment\ number, hits, score >$ was collected. These data were analysed to obtain the plot. The default value for the cutoff parameter S is 32. This is the threshold score for an alignment to be recorded as an *HSP*. The plot indicates that, on average, *HSPs* that meet this threshold contain five word hits for the values of word size, W=3, and threshold, T=11.

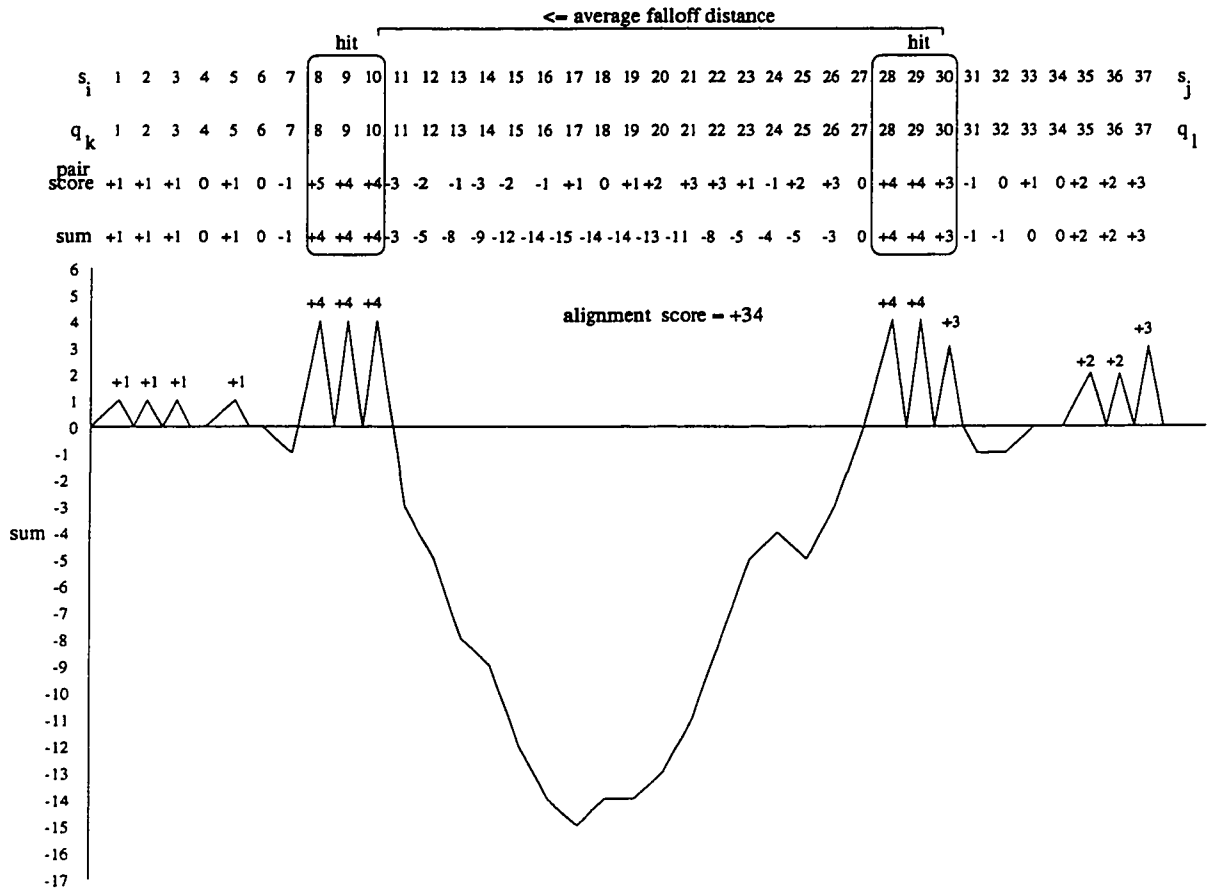

Figure 38: *Two-Hit Algorithm - Case where Extension is Called.* The example shows the alignment of two segments, $q_k - q_l \| s_i - s_j$, which contains two word hits, and a plot of sum v.s. alignment position, t. The value of `score` is the residue pair score matrix value for the residue pair $[q_{k+t}][s_{i+t}]$ where $k <= k+t <= l$ and $i <= i+t <= j$. The value of `sum` is the cumulative sum of residue pair scores which is reset to 0 after each positive scoring residue pair, but in which the scores of negative scoring residue pairs are accumulated. The alignment contains two word hits (W=3) at positions 10 and 30. Word hits are short positive scoring alignments of length W which correspond to positive scoring peaks on the plot of sum v.s. alignment position. The two-hit algorithm attempts to extend the word hit located at t=30 which is within the average falloff distance of the hit at t=10. Since the two hits satisfy the distance constraint, it is probable that the left extension of the hit at t=30 will encompass the hit at t=10 before `sum` accumulates to the value of the falloff parameter X. In this example, the negative scoring sub-alignment between the two word hits has a score of -15, which is below the default of X = -22.

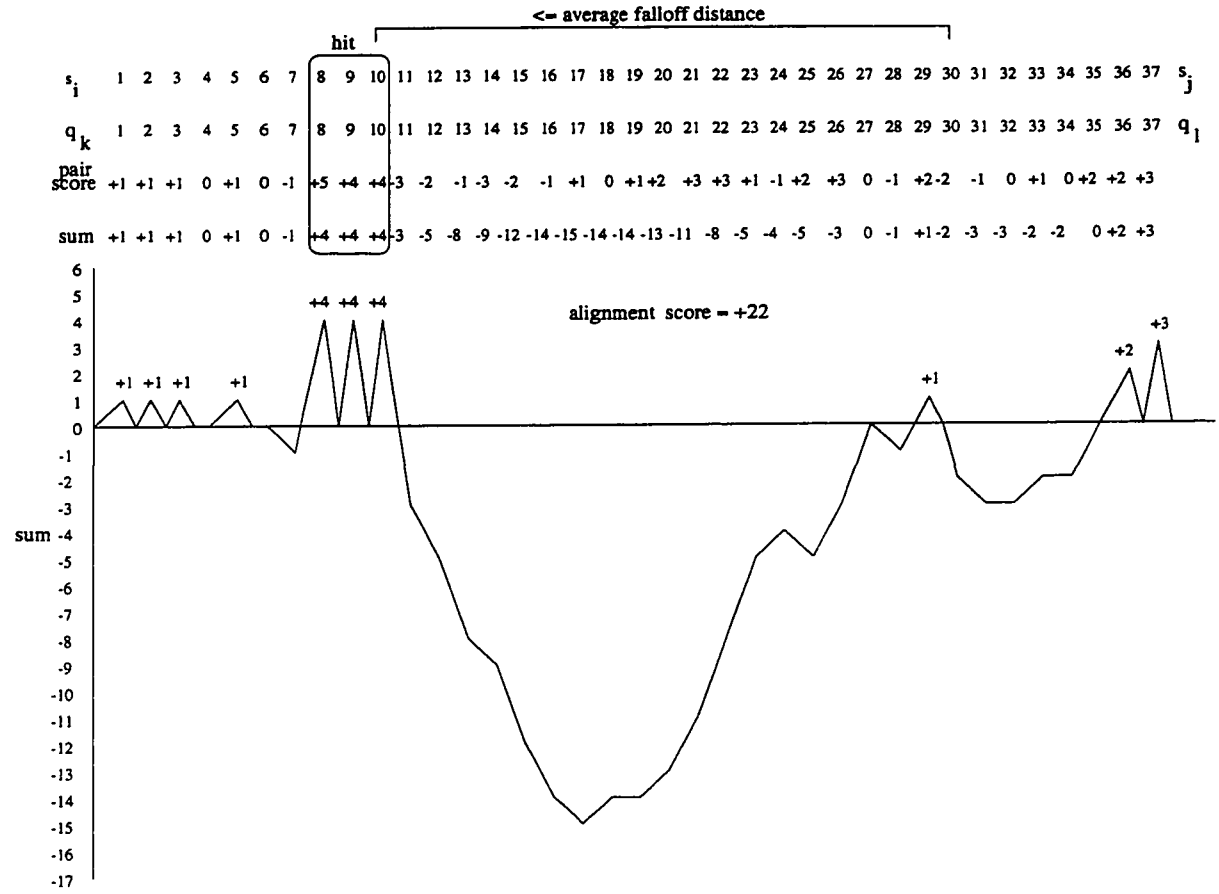

Figure 39: *Two-Hit Algorithm - Case where Extension is not Called.* This example is analogous to Figure 38 with the exception that the aligned segment $q_k - q_l \| s_i - s_j$ contains only a single word hit at t=10. There is no invocation of the extension algorithm since there is no other hit within the average falloff distance of the hit at *t=10.*

4.4.2 Two-Hit Design and Implementation

Figure 40 shows the design of the BLASTP program that implements the two-hit algorithm. The `BlastWordFind` procedure was augmented with functionality to count the number of hits per alignment. The `BlastWordFinder` structure was augmented with new members.

The `WordFind` procedure was modified to call the extension procedure only for hits within the average falloff distance of a previous hit on a particular diagonal. The following constant attributes were added to the `BlastWordFinder` structure; `MAX_DIAG`, `INC`, `DIAG_FTR`, and `FALLOFF`. The values of these four attributes were determined using the test search described in subsection 3.2.1. The constant `DIAG_FTR` is the value added to the diagonal, `diag`, to obtain a positive index to the array `HitsDiag`. The constant `MAX_DIAG` is the maximum value of this index. The constant `INC` is the value added to `current_inc` with each new subject scan. The average falloff distance is `FALLOFF`.

Three variable attributes were also added to the `BlastWordFinder` structure. The array `HitsDiag` stores the query offset of the last hit on a particular diagonal. It is indexed by `index`. The value of `current_inc` is added to `current_qoff` to uniquely localise that offset to a particular subject scan. This permits the re-use of the `HitsDiag` array, thus avoiding the time consuming re-initialisation of the array for each scan. This idea is borrowed from [32].

The scanning procedure, `WordFinder`, was augmented with a local variable `current_qoff` which stores the query offset of the current word hit. The inner scanning loop was modified to compute the `index` and `current_qoff`, check if a diagonal has two word hits within the falloff distance and store the query offset of the last hit on a particular diagonal in `HitsDiag`. Recall that the inner scanning loop iterates over the list of offsets of query W-mers to which some W-mer on the subject was matched; i.e. the subject W-mer is in the neighborhood of the query W-mer. At the end of the procedure, `current_inc` is set to its value for the next scan.

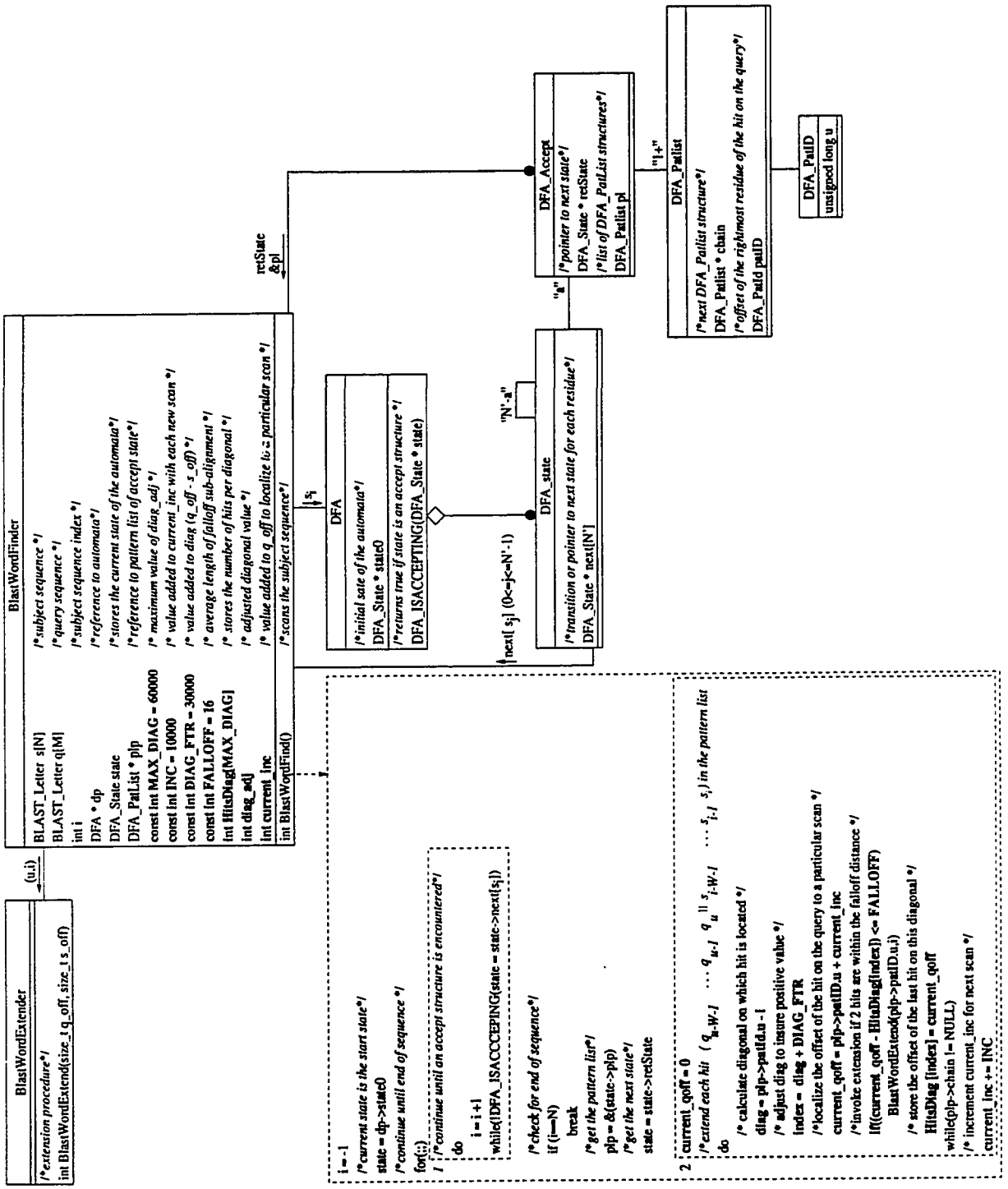

Figure 40: *Modified Scan Sub-Architecture that Implements the Two-Hit Algorithm.*

4.4.3 Two-Hit Results

A BLASTP program that employs the two-hit optimisation runs 63% faster than the unmodified program (Table 10). The execution time for extending is significantly reduced (Table 11), but the time for scanning is increased (Table 12). The increased amount of time spent scanning does not significantly offset any time saved in extending (Table 13).

Program	CPU cycles x 10^{10}	CPU time (s)	wall clock time(s)
BLASTP (two-hit)	0.8	27.4	40
BLASTP (unmodified)	2.1	73.5	102

Table 10: *Comparison of Program Time.* The program using the two-hit algorithm uses approximately 63% less CPU cycles than the program using the unmodified algorithm. There is a corresponding decrease in CPU time. The savings in wall clock time is approximately 61%. Wall clock time is the time reported by the program as real time [4], page 17 whose values may vary between runs depending on the availability of the CPU for the BLASTP process.

Procedure	CPU cycles x 10^{10}	CPU time (s)
BlastWordExtend (two-hit)	0.6	19.4
BlastWordExtend (unmodified)	2.0	68.5

Table 11: *Comparison of BlastWordExtend Procedure Time.* The procedure that implements the extension step of the algorithm, `BlastWordExtend`, accounts for 72% less time in the two-hit program than it does in the unmodified program. There is a corresponding decrease in CPU time.

Procedure	CPU cycles	CPU time (s)
BlastWordFinder (two-hit)	0.2 x 10^{10}	7.1
BlastWordFinder (unmodified)	0.1 x 10^{10}	3.8

Table 12: *Comparison of BlastWordFinder Procedure Time.* The procedure that implements the scanning step of the algorithm, `BlastWordFinder`, accounts for 90% more time in the two-hit program than it does in the unmodified program. There is a corresponding increase in CPU time.

Procedure	CPU cycles	CPU time (s)
BlastWordFinder	− 0.10 x 10^{10}	-3.3
BlastWordExtend	+ 1.43 x 10^{10}	+49.1
Net Total	+ 1.33 x 10^{10}	+45.8

Table 13: *Net Performance Results for Scanning and Extension Procedures.* There is a net savings in extension time using the two-hit program since `BlastWordExtend` is invoked less frequently. There is a net gain in scanning time since lines of code were added to `BlastWordFinder` to implement the two-hit algorithm. However, the performance cost in scanning is small relative to the performance gain in extending.

4.4.4 Two-Hit Effect on HSP Detection

The two-hit BLASTP program reports all alignments that the unmodified program does. Recall that the alignment reported in the output is the highest scoring one from the set of HSPs from which the $P(N)$ value was calculated. Scores are identical. However, the two-hit program reports lower N values. Here, N is the number of *HSPs* in the set which was ascribed the lowest P-value (subsection 2.2.4). This effects the value for $P(N)$. The two-hit algorithm may not detect some *HSPs* that are detected by the unmodified algorithm as a result of not extending aligned segments which contain a single hit. For the experimental search this has no effect on the reported alignments since the missed *HSPs* were not the highest scoring members of the *HSP* set; i.e. they were not the maximal scoring segment pairs or *MSPs*.

Chapter 5

Conclusions and Contributions

5.1 Conclusions

This section presents the speed-ups obtained and costs incurred from each of the programs employing an optimised algorithm.

5.1.1 Summary of Performance Measures

Each of the optimisations proposed in this thesis results in a significant decrease in the computational time required to perform a search. Figure 41 compares the program and extension performance of each optimised program with that of an unmodified program. Figure 42 summarises the CPU speed-ups for each optimised program.

BLASTP Program	Program Performance			Extension Performance	
	CPU cycles x 10^{10}	CPU time (s)	wall clock time (s)	CPU cycles x 10^{10}	CPU time (s)
BLASTP (row-address)	1.8	62.9		1.7	57.9
BLASTP (residue-doublet)	1.3	45.5	109	1.2	39.9
BLASTP (two-hit)	0.8	27.4	40	0.6	19.4
BLASTP(unmodified)	2.1	73.4	100	2.0	68.4

Figure 41: *Summary of Performance Measures.*

BLASTP Program	% Increase in Program Performance
BLASTP (row-address query)	15
BLASTP (residue-doublet)	48
BLASTP (two-hit)	63

Figure 42: *Performance Gain Per Optimisation.*

5.1.2 Gain in Performance vs. Cost in HSP Detection

This subsection weighs the cost of using an optimised algorithm against the gain in performance. Cost is measured as a loss in search sensitivity or the decreased ability of the optimised algorithm to detect *HSPs*.

row-address

The row-address optimisation enhances performance by 15%, but is not observable in wall clock time for a single search, but could be in the context of, for example, a BLAST server that is continuously executing searches; one estimate shows that two hundred additional searches per day could be performed. The optimised algorithm produces the exact *HSP* set produced by the unmodified algorithm.

residue-doublet

The residue-doublet optimisation decreased the CPU time by 48%, but increased time to access memory most likely because the residue-doublet score matrix exceeded the cache size of the DEC Alpha (4 KB). This savings could be realized in wall clock time on a machine whose cache is sufficiently large to store the entire residue-doublet matrix. Performing extensions in steps of residue-doublet pairs increases the heuristic character of the algorithm. As a consequence of grouping residues into doublets, in some cases, the residue-doublet algorithm, compared to the unmodified algorithm, misses lower scoring *HSPs*, reports overall scores that are slightly lower and continues extensions beyond there normal point of termination. These anomalies are described fully in subsection 4.3.4. These anomalies mean that an *HSP* may not be locally maximal as described in [1]. However, such deviations could be corrected by post-processing the *HSP* set or by adjusting the X parameter or may be accepted as a reasonable compromise to obtain the performance speed-up.

two-hit

The two-hit optimisation provides the greatest performance enhancement at 67%. Again, this optimisation increases the heuristic character of the algorithm. Those cases where a single hit is extended into and *HSP* will be missed. However, these cases may be detected by lowering the threshold score for neighborhood words, T [29], which essentially increases the probability of having two hits on the alignment.

5.2 Contributions of this Thesis

This work makes three valuable contributions to the fields of bioinformatics and software engineering: (1) An optimised BLASTP algorithm (2) A parametrised description of the algorithm and (3) A case study in reverse engineering using execution profilers.

The three optimisations provide a significant performance enhancement to a popular algorithm used for protein database scanning. The row-address and residue-doublet optimisation take advantage of the view that a sequence can have multiple equivalent representations each of which is used for a particular part of the overall computation. The development of the two-hit optimisation arose from an experiment in which a query was compared to a single database sequence and the number of hits per alignment was measured. This experiment was later extrapolated over the entire database, the results of which are shown in Figure 37. However, this thesis makes no claims of original discovery of this optimisation since it is implemented in the BLAST version 2.0 program [29]. This thesis provides a parameterised description of the BLASTP algorithm. Descriptions of the algorithm that exist in the literature provide only general textual and diagrammatic descriptions of each of the three steps of the algorithm from which a direct implementation is not possible. From a software engineering perspective, this thesis provides a reverse engineering methodology based on information obtained from execution profilers. In addition, this work demonstrates the use of modelling techniques to describe the design of a reverse engineered program.

Appendix A

Summary of Object-Oriented Concepts and Notation

A.1 OMT Concepts and Notation

This work uses concepts and notation from both OMT [35] and UML [36]. Class diagrams are part of the object model. *Classes* group *attributes* and *operations*, coupling a part of a program's state with its functionality. *Attributes* are data that are contained within the class while operations are functions or transformations that may be applied to these data. *Associations* are relationships among classes. A special kind of association is *aggregation* which is the "part-whole" or "a-part-of" relationship in which component classes are associated with an assembly class. An association may be constrained by its *multiplicity* which specifies how many instances of one class may relate to a single instance of an associated class.

Classes are represented using rectangles subdivided into three layers. The top layer contains the class name, the middle its set of attributes and the bottom its set of operations. Associations are lines between classes. Aggregation is indicated with a diamond which is drawn at the "assembly class" end of the association line. Multiplicity is indicated with special symbols at the end of the association line. A filled ball indicates "many" while an integer specifies an exact number of associated instances. A line without multiplicity symbols indicates a one-to-one association.

A.2 UML Concepts and Notation

Collaboration diagrams of UML label links among objects with messages. Messages travel from sender to receiver. In most cases they are invocations of a method of receiver's method by the sender. Often, an argument of this method is a member data of the sender. Thus, messages essentially send data from one object to another.

The notation for messages is an arrow above a link or association. The arrowhead indicates the direction in which the message is sent. The arrow can be labelled with a method name and may contain a sequence number.

This work uses class diagrams to describe the program sub-architectures. The diagrams use the OMT modelling entities just described. In addition, the diagrams show message passing. However, unlike collaboration diagrams, messages are sent between classes. The arrow notion is used, but the label is an attribute data of the sender class. The class diagrams are also augmented with algorithm boxes whose notation is a dashed line rectangle. A dashed arrow links the operation of a class to the algorithm box that displays its pseudo-code.

Appendix B

Future Work

B.1 Residue-Doublet Algorithm

B.1.1 Reducing Cache Misses

The residue-doublet extension algorithm shows a promising performance increase that is not realizable in wall clock time. A possible explanation is that any decrease in the number of instructions executed is offset by an increase in the time to access the score matrix because of cache misses. Perhaps a suitable ordering of the rows and columns can be found that improves the ratio of cache hits. A score matrix can be represented as an array whose entries are ordered from the most to least frequently accessed. Thus, a given entry is more likely to be cached (Figure 43). The frequency of occurrence of residue symbols in the protein NRDB was determined (Table 44). The BLOSUM62 residue pair score matrix can be rearranged so that the order of the column and row indices correspond to the order shown in Table 44. From this matrix, a residue-doublet pair frequency score matrix can be constructed using the algorithm of Figure 34. From this matrix, a residue-doublet pair frequency array can be constructed using the algorithm of Figure 45.

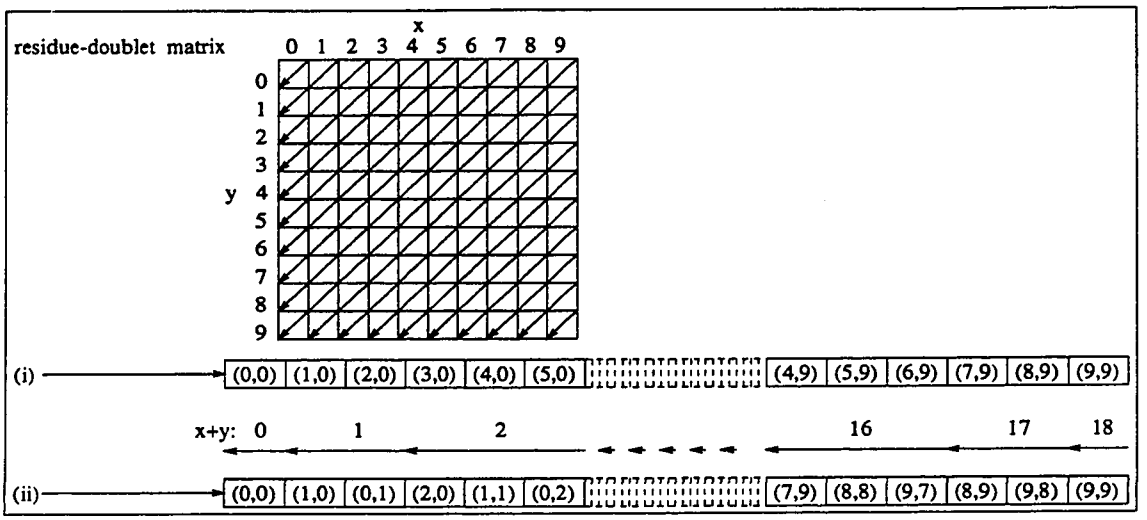

Figure 43: *Logical Organisation of the Frequency Residue-Doublet Score Matrix.* Matrices can be represented as arrays. (i) A matrix represented as a row-major order array. (ii) A matrix represented as a diagonal order array. The arrows above the sequences of matrix cells are the diagonals on which the cells are located. The organisation of (ii) is used to represent the residue-doublet pair frequency score matrix. The x and y coordinates of the matrix are residue-doublets in decreasing order of occurrence in a protein sequence database. The entries of the matrix are residue-doublet pair scores. The diagonal order array representation of this matrix orders the scores according to frequency of occurrence; from most to least frequently occurring.

Residue	Occurrences	Frequency (%)
L	6.6	9.2
A	5.3	7.36
S	5.2	7.35
G	4.9	6.9
V	4.6	6.4
E	4.4	6.2
T	4.2	5.9
K	4.1	5.8
I	4.0	5.7
R	3.70	5.19
D	3.67	5.16
P	5.2	5.1
N	3.3	4.6
Q	1.3	4.1
F	2.9	4.0
Y	2.3	3.3
M	1.64	2.3
H	1.60	2.2
C	1.3	1.8
W	1.0	1.3
X	0.1	0.01
Z	0.01	0.00001
B	0.01	0.00001

Figure 44: *Frequency of Occurrence of Residue Symbols in the Protein NRDB [21].* The number of occurrences of each symbol was done by counting. Column 1 gives the number of occurrences of each residue symbol. Column 2 shows the relative frequency of each symbol.

```
N = NUM_ROWS = NUM_COLUMNS

/* traverse the diagonals above and including the main diagonal and add their entries to array_frd */
for column = 0 to N
    temp_column = column
    for ( row = 0 ; row < column ; row++)
        array_frd[index++] = matrix_frd[row][temp_column]
        temp_column--

/* traverse the diagonals below the main diagonal and add their entries to array_frd */
for row = 1 to N
    temp_row = row
    for ( column = N - 1 ; column >= temp_row ; column-- )
        array_frd[index++] = matrix_frd[temp_row][column]
        temp_row++
```

Figure 45: *Algorithm for Constructing the Frequency Residue-Doublet Pair Score Array.* The array, `array_frd` is constructed from the frequency residue-doublet pair score matrix, `matrix_frd`. The algorithm consists of two parts. The first traverses the diagonals above and including the main diagonal; the second traverses the ones below. The order in which the entries of a particular diagonal are traversed are from lowest to highest diagonal value, $x + y$. The order in which the entries of a particular diagonal are traversed are from lowest to highest x value. As the diagonals are traversed their entries are copied to the diagonal order array, *array_frd.*

Bibliography

[1] 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.

[2] James Peter. Breakthroughs and Views - Of Genomes and Proteomes. *Biochemical and Biophysical Research Communications*, 231:1–6, 1997.

[3] Benson Dennis, Boguski Mark, Lipman David J., and Ostell James. The National Center for Biotechnology Information - Program Description, http://www.ncbi.nlm.nih.gov/. *Genome*, 6:389–391, 1990.

[4] National Center for Biotechnology Information. BLAST UNIX System V Manual Page, Oct. 23, 1994. Version 1.4.

[5] Patterson David A. and Hennessy John L. *Computer Architecture, A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.

[6] Stryer Lubert. *Biochemistry.* W.H. Freeman and Company, New York, third edition, 1988.

[7] Pearson William R. Effective Protein Sequence Comparison. *Methods in Enzymology*, 266:227–258, 1996.

[8] Pearson William R. Protein Sequence Comparison and Protein Evolution, 1998. Conference on Intelligent Systems for Molecular Biology Tutorial Series, Sunday June 28, Montreal, Quebec, Canada.

[9] Altschul Stephen F., Boguski Mark S., Gish Warren, and Wootton John C. Issues in searching molecular sequence databases. *Nature Genetics*, 6:119–129, Feb 1994.

[10] Barton G J. Protein Sequence Alignment and Database Scanning. This article will appear in: Protein Structure prediction - a practical approach, Edited by M.J.E.Sternberg, http://barton.ebi.ac.uk/barton/papers/rev93_1/rev93_1.html.

[11] Pearson William R. and Lipman David J. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85:2444–2448, 1988.

[12] Dodge C, Schneider R, and Sander C. The HSSP database of protein structure-sequence alignments and family profiles. *Nucleic Acids Research*, 26(1):313–315, 1998.

[13] Holm L and Sander C. Touring protein fold space with Dali/FSSP. *Nucleic Acids Research*, 26(1):316–319, 1998.

[14] Dayhoff M.O., Schwartz R.M., and Orcutt B.C. A Model of Evolutionary Change in Proteins. *Atlas of Protein Sequence and Structure*, 5(3):345–352, 1978.

[15] Henikoff Steven and Henikoff Jorja G. Amino acid substitution matrices from protein blocks. In *Proceedings of the National Academy of Science*, volume 89, pages 10915–10919, November 1992.

[16] Baxevanis Andreas D., Boguski Mark S., and Ouellette B.F. Francis. Genome Analysis: A Laboratory Manual. To be published by Cold Spring Harbor Laboratory Press, http://www.cshl.org/books/g_a/bk1ch7/.

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

[18] 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.

[19] Benson DA, Boguski MS, Lipman DJ, Ostell J, and Ouellette BF. Genbank. *Nucleic Acids Research*, 26(1):1–7, 1998.

[20] Bernstein FC, Koetzle TF, Williams GJ, Meyer EE Jr, Brice MD, Rodgers JR, Kennard O, Shimanouchi T, and Tasumi M. The Protein Data Bank:

a computer-based archival file for macromolecular structures. *J Mol Biol*, 112(3):535–542, 1977.

[21] National Center for Biotechnology Information. The Protein Non-redundant Database (NRDB). ftp://ncbi.nlm.nih.gov/blast/db/nr.Z.

[22] Karlin Samuel and Altschul Stephen F. Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proceedings of the National Academy of Sciences*, 87:2264–2268, 1990.

[23] National Center for Biotechnology Information. Blast Help Page. Version 1.4.

[24] Needleman S. and Wunsch C. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology*, 48:444–453, 1970.

[25] Smith T.F. and Waterman M.S. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[26] Pearson William R. and Miller Webb. Dynamic programming algorithms for biological sequence comparison. *Methods in Enzymology*, 210:575–601, 1992.

[27] Wilbur W.J. and Lipman David J. Rapid similarity searches of nucleic acid and protein data banks. *Proceedings of the National Academy of Sciences*, 80:726–730, 1983.

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

[29] Altschul Stephen F., Madden Thomas L., Scaeffer Alejandro A., Zhang Jinghui, Zhang Zheng, Miller Webb, and Lipman David J. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, 1997.

[30] UNIX. prof. Manual Page osf1-4.0B.

[31] Graham Susan L., Kessler Peter B., and McKusick Marshall K. gprof: a Call Graph Execution Profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, volume 17, pages 120–126. SIGPLAN Notices, June 1982.

[32] National Center for Biotechnology Information. BLAST Source Code Version 1.4. ftp://ncbi.nlm.nih.gov/blast/old1.4.

[33] UNIX. gprof. Manual Page osf1-4.0B.

[34] UNIX. pixie. Manual Page osf1-4.0B.

[35] Rumbaugh James, Blaha Michael, Premerlane William, Eddy Frederick, and Lorenson William. *Object-Oriented Modelling and Design.* Prentice Hall, 1991.

[36] Rational Software Corporation. The Unified Modelling Language, Notation Guide. http://www.rational.com/uml/html/notation/.

[37] Kernighan Brian W. and Ritchie Dennis M. *The C Programming Language.* Prentice Hall Software Series, 2nd edition, 1991.

[38] Darnell Peter A. and Margolis Philip E. *C A Software Engineering Approach.* Springer-Verlag, 2nd edition, 1991.

[39] Gish Warren. DFA Library UNIX Manual Page BLAST Version 1.4, July 1992.

[40] Hopcroft J.E. and Ullman J.D. *Introduction to Automata Theory , Languages, and Computation.* Addison-Wesley, 1979.