Three Improvements to the BLASTP Search of Genome Databases

Shawn Delaney, Greg Butler, Clement Lam, Larry Thiel
Department of Computer Science, Concordia University,
1455 de Maisonneuve Blvd. West, Montréal, Québec, Canada, H3G 1M8
gregb@cs.concordia.ca

Abstract

The BLASTP program is a search tool for databases of protein sequences that is widely used by biologists as a first step in investigating new genome sequences. BLASTP finds high-scoring local alignments ($q_i q_{i+1} \ldots q_{i+k} || s_j s_{j+1} \ldots s_{j+k}$) without gaps between a query sequence q and sequences s in the database. The score of an alignment is the sum of the scores of individual alignments $q_{i+t} || s_{j+t}$ between amino acids that make up the protein. These individual scores come from a scoring matrix modeling the rate of evolutionary mutation.

Here we provide a detailed description of the original program and three separate optimisations to it. BLASTP consists of three steps, that we call neighbourhood construction, hit detection, and hit extension. The three optimisations target hit extension since it accounts for 93% of the execution time. The first optimisation alters the data representation of the query sequence and the related code for indexing the scoring matrix. The second optimisation performs extensions in step-sizes of two rather than one. The third optimisation forestalls the calling of the hit extension step in cases that are unlikely to lead to a high-scoring alignment. Individually, the three optimisations show speed ups of 15%, 48%, and 63% respectively.

1. Introduction

Worldwide, biologists are collaborating on genome projects to determine the complete genomes of many organisms. This has already produced large databases, such as Genbank [3], of nucleotide sequences and protein sequences. A nucleotide sequence is a string over a four-letter alphabet, while a protein sequence is a string over a twenty-letter alphabet, one letter for each of the twenty amino acids that occur in proteins. The first step in investigating a new sequence is to compare it against Genbank entries to find similar sequences. This comparison is an approximate match, taking into account the fact that individual nucleotides or amino acids mutate during the course of evolution. A *scoring matrix* S quantifies the rate of mutation: an entry $S(a_1, a_2)$ gives a normalized frequency for the mutation of amino acid a_1 into a_2 over one unit of evolutionary time. The score for an alignment of one sequence with another is derived from the score $S(a_1, a_2)$ for each amino acid pair, where a_1 is aligned with a_2. These alignments can be *local* or *global*. A local alignment is a match between a segment of each sequence, whereas a global alignment is a match between the complete sequences. There is an art to interpreting whether the alignment, the value of the score, and its level of statistical significance are really biologically meaningful, but generally biologists look for high-scoring alignments of their new sequence against the Genbank entries. BLASTP is the most widely used program for determining alignments of protein sequences against databases such as Genbank. It is one of the BLAST (**B**asic **L**ocal **A**lignment **S**earch **T**ool) suite of programs [1].

Search speed is a critical issue in scanning sequence databases whose sizes continue to grow [8]. We address this issue by optimising the existing BLASTP version 1.4 program. However, to do that, we must first understand the algorithm and the source code. While the source code is readily available, there is no clear description of the algorithm in the literature or the program documentation. Hence, our first step is a focused reverse engineering of the program to identify those parts of the program that, if optimised, would give the largest CPU speed-up. The description that resulted is that BLASTP is a three-step algorithm that succeeds in only scanning the database for exact matches. The first step is to create a *neighbourhood* for each (short) segment of length W of the query sequence. The neighbourhood consists of all sequences of W amino acids that match the query segment with a high-score. An automaton is built to recognize the union of all neighbourhoods. The second step is to scan the database for exact matches to any neighbour. These matches are called *hits*. The third step attempts to extend a hit into a *high-scoring pair of segments*(HSP) with approximate matches to the left and right of the hit. As each pair of aligned residues is included into the alignment, the score of

the aligned pair is looked-up in a score matrix and added to a running sum. Extension of a hit continues until the falloff value, *X*, is reached.

The execution profile of the BLASTP program under our test conditions 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.

We propose two types of optimisations: 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. There are three optimisations. 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 are coded and and plugged into the BLASTP version 1.4 program. For the third, or two-hit optimisation, the scanning procedure is 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* is 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. 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.

Two important results are described in this paper. 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, we present a parameterised description of the algorithm that clearly shows how that algorithm works; such a description was not previously available in the literature.

The layout of the paper is as follows. Section 2 introduces the BLAST program, its usage, and its relationship to other algorithms for comparing sequences. Section 3 presents a detailed description of the program structure and a parametrised description of the BLASTP algorithm and its three main steps. Section 4 presents the three optimisations, while Section 5 concludes the paper.

2. Background on BLAST

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. 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. 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*, 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.

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

blastp `[db_file]` `[qry_file]`

The FASTA format consists of a sequence descriptor and a sequence of characters that represent the protein. The

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

Figure 1. Example Amino-Acid Sequence

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.

The program has several command line options [6]. Those options that a typical user may specify are: (1) `-matrix`, which specifies the scoring 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 `[db_file] [qry_file] -matrix [m_file]`

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

blastp `[db_file] [qry_file]` `W=`n_W `T=`n_T `X=`n_X

The sensitivity can be increased, i.e. more *HSP*s can be detected, (1) by 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 3.

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 *HSP*s, 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 [6].

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 Figure 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 *HSP*s. The fourth column, `N`, displays the number of *HSP*s in the set 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 *HSP*s between two sequences, the lower is this probability. If not otherwise specified, the list of one-line summaries is sorted by increasing `P-value`.

The set of *HSP*s 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 is 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` 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 *HSP*s 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.

2.1. Algorithms for Protein Sequence Comparison

This section places the BLASTP algorithm in context. The popular protein sequence comparison algorithms fall within two groups: dynamic programming and heuristic (Table 1). 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.

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

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.

Table 1. Sequence Comparison Algorithms.

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

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 subsegments. 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-Wunsch [10] and Smith-Waterman [12]. The former finds the best alignment of two sequences over their entire length; the latter, the best local alignment. 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. they do not compute the positional mapping between residues. The alignment must be obtained by backtracking 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 10^5. The inapplicability of these algorithms to sequence database searching led to the development of heuristic algorithms that sacrifice sensitivity for speed. The most popular of these are FASTA [11, 13, 9], which can compute either global or local gapped alignments, and BLASTP, which can compute local ungapped alignments. (However, the recent BLAST version 2.0 suite of programs employs an algorithm that computes gapped alignments [2].)

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

```
>gi|1575707 (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  GFTIRNAASWTIHPQGCEDLTAAASTIIAPHDSPNTDGFNPESCRNVMISGVRFSVGDDC  96
             G T++N+   + I    GC +          +++P  +SPNTDG  + +S    V  I          GDDC
Sbjct:  156  GVTVQNSQMFHILVDGCHNAMIQGVKVLSPGNSPNTDGIHVQSSSGVS IMNSNIGTGDDC  215

Query:   97  IAVKAG  102
             I++   G
Sbjct:  216  ISIGPG  221
```

Figure 3. Example BLASTP Output — HSP List

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 $(N \times M)$, but the number of computations based on residue-residue comparisons is greatly reduced.

Q, query sequence, $q_0 q_1 q_2 \ldots q_l$
DB, subject sequence database, size = *K*
M, function that returns the alignment score
HSP, set of high scoring segment pairs
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, $s_0 s_1 s_2 \ldots s_m$
  N = **Build-Neighborhood**(*Q, M, W, T*)
    for (i=0 ; i<*K* ; i++)
      SB = *DB*[i]
      Scan(*SB*,*N*)

Figure 4. The BLASTP Program

3. A Description of the BLASTP Program

This section provides a parameterised description of the BLASTP program and each of the three steps of the algorithm. The BLASTP program (Figure 4) first builds the neighborhood, then iteratively retrieves a subject sequence from the database and scans it for hits. Upon detection of a hit, the extension step is invoked.

For sequences $q = q_0 q_1 \ldots q_n$ and $s = s_0 s_1 \ldots s_n$, a local alignment without gaps, $(q_i q_{i+1} \ldots q_{i+k} || s_j s_{j+1} \ldots s_{j+k})$, is a positional mapping from a segment of q to a segment of s so that the corresponding individual amino acids q_{i+t} and s_{j+t} are aligned, for t = 1 to k. The score $M(q_i q_{i+1} \ldots q_{i+k} || s_j s_{j+1} \ldots s_{j+k})$ of an alignment is the sum of the scores $M(q_{i+t} || s_{j+t})$ of the individual alignments $q_{i+t} || s_{j+t}$. These individual scores come from a scoring matrix $M[q_{i+t}, s_{j+t}]$ modeling the rate of evolutionary mutation.

The BLASTP algorithm works in three steps:

1. **Neighborhood Construction**. A set of words of length *W*, called 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 (exact) matches to a word in *N*.

3. **Hit Extension**. The match, or hit *H*, is extended into a potentially higher scoring alignment.

The neighborhood construction step (Figure 5) 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*.

The hit detection step (Figure 6) 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 exact matches to a member of the neighbor-

```
N = Build-Neighborhood(Q,M,W,T)
  AA, amino acid alphabet
  for ( i=0 ; i < l-W+1 ; i++ )
     if ∃ a word n0n1n2...nW-1, where nj ∈ AA
     such that M(qiqi+1...qi+W-1||n0ni...nW-1) >= T
     then N ⋃ < n0ni...nW-1,qi+W-1 >
```

Figure 5. Step One — Neighborhood Construction

hood. When a match is found, the extension step is invoked.

```
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 ((sj sj+1...sj+W-1) ∈ N)
        H.s_off = sj+W-1
        H.q_off = offset, where < sj sj+1...sj+W-1 , offset> ∈ N
        Extend(H)
```

Figure 6. Step Two — Hit Detection

The hit extension step (Figure 7) 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 (2–3). The first loop (4–9) 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* (5). When *sum* is positive (6) it is added to *score* and $q\_end||s\_end$ is advanced right. A negative *sum* (7) 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 (11–12). The second loop (13–21) extends in the left direction. Residue pair scores are accumulated in *sum* (17). If *sum* is positive, it is added to *score* then reset to zero and the alignment is extended (18–19). If *sum* falls below *x*, the extension terminates (21). Longer extensions are favored by allowing *x* to be set to *-score* (10,20,30). A right extension occurs if the conditions in (22) 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 (23). The third loop works in the same manner except that the extension proceeds in the right direction (24–32). The left extension may continue if the conditions in (34) hold. If so, *s*, *sum* and *score* are saved in the right extension parameters, *rights*, *rightsum* and *rightscore* respectively (35). The maximal scoring alignment is stored in the *HSP* set if *score* meets the threshold *S* (37–38). Figure 8 illustrates the dynamics of the extension algorithm.

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

Figure 7. Step Three — Hit Extension

4. Algorithm Optimisations

The optimisations described focus on the `BlastWordExtend` procedure. They 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. Employ-

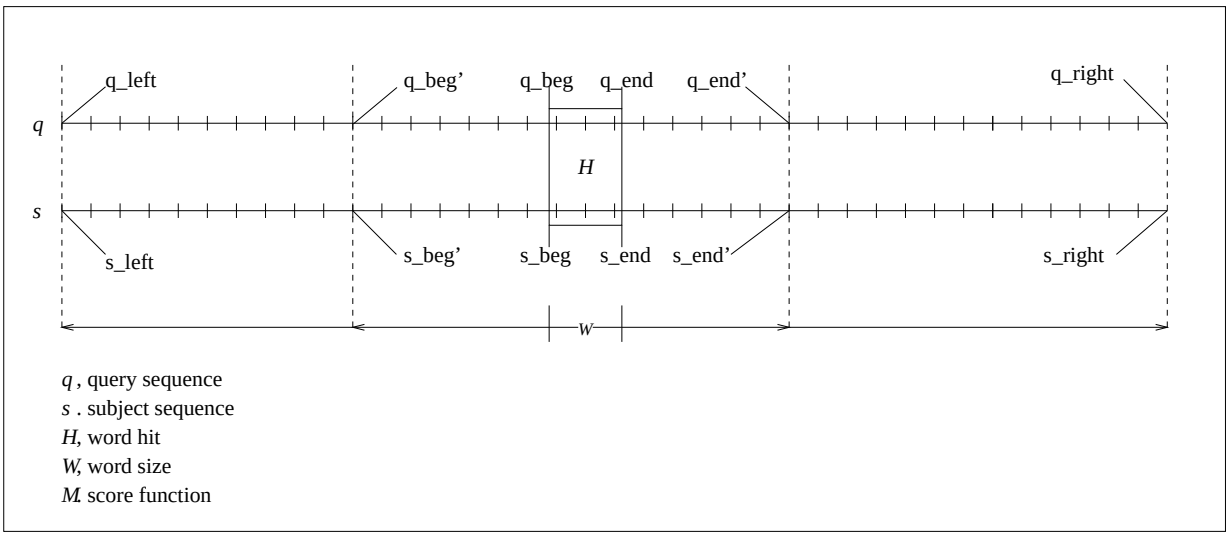

Figure 8. Dynamics of Extension Algorithm. A word hit H is a sub-alignment of length W, $H = (q_{beg}...q_{end}||s_{beg}...s_{end})$ whose score, $M(H) \geq T$. The extension algorithm finds the locally maximal alignment starting from H. Extension continues in either direction until $M(q_{left}...q^{'}_{beg}||s_{left}...s^{'}_{beg}) < X$ or $M(q^{'}_{end}...q_{right}||s^{'}_{end}...s_{right}) < X$. The total score of the alignment is $M(q^{'}_{beg}...q^{'}_{end}||s^{'}_{beg}...s^{'}_{end})$

ing 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.

The following three subsections present each optimisation in turn, including a discussion of the effect they have on the algorithm's sensitivity to detecting similarity. The fourth subsection presents the performance gains of the optimisations.

4.1. Row-Address Sequence Representation

The profile shows that the lines of code within the procedure `BlastWordExtend` that access the residue pair score matrix account for 63% of the execution time. The logical instruction sequence for the code fragment that accesses the score matrix is given in Figure 9 part (iii). 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 out 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. 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.

This optimisation is more effective for the DEC Alpha 64-bit architecture than for some other architectures. The

small integers, in the range 0–19, are stored as bytes, and have to be extracted from a 64-bit word before they can be used as indexes. On the DEC Alpha, this extraction is a particularly slow instruction.

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.

4.2. Residue-Doublet Sequence Representation

The unmodified extension algorithm works in steps of aligned residue pairs. The optimised algorithm performs extensions in steps of aligned residue-doublet pairs (Figure 10). 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.

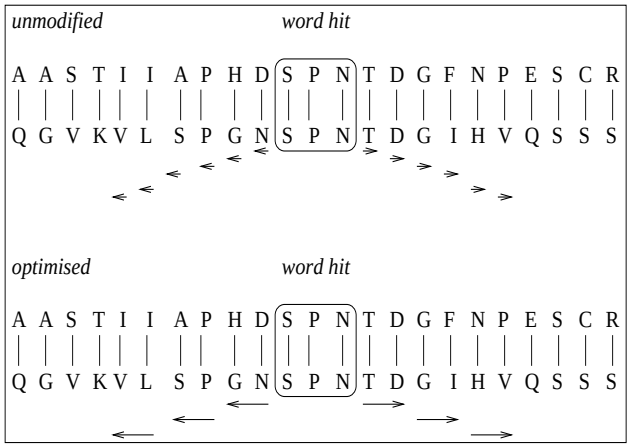

Figure 10. Extending in Steps of Two.

This optimisation requires new data structures for the scoring matrix, and the sequences. These are in additional to the structures used in unmodified BLASTP, since other steps of the algorithm depend on the original structures.

A doublet consists of two adjacent residues. Since there are 20 residues, there are 400 residue pairs, and they can be represented as integers in the range 0–399. A scoring matrix for doublets is required: it is indexed by the integers 0–399, and directly calculated from the original scoring matrix. Each sequence requires three representations: the original one, a representation as doublets beginning at an odd position, and a representation as doublets beginning at an even position. These last two representations may pad the original sequence, so it has even length.

Residue-Doublet Effect on HSP Detection. 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. One can easily (and quickly) compensate for these anomalies by post-processing the set of *HSPs*.

Anomaly 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 $S < 50$ are missed by the residue-doublet algorithm that are found by the unmodified algorithm.

Anomaly 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.

Anomaly 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 $S > 50$.

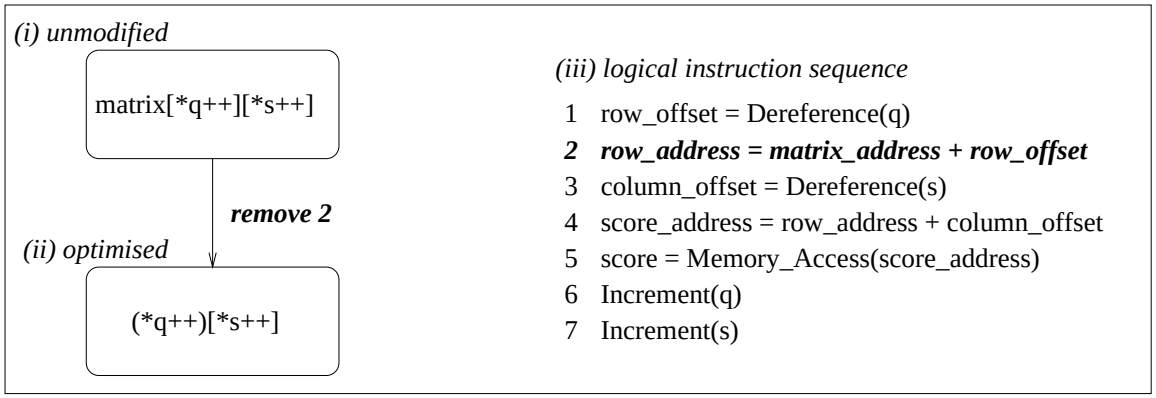

Figure 9. Logical Instruction Sequence for Retrieval from Score Matrix. Part (iii) shows the logical instruction sequence for retrieving a score from the matrix. Part (i) shows the line of C code for this access in the unmodified BLASTP, where q and s are references to residues, and residues are integers in the range 0–19. Part (ii) shows the line of C code for the optimised BLASTP, where now q is a reference to the row of the score matrix for the corresponding residue; s still references an integer in the range 0–19.

4.3. Two-Hit Detection

The third step of the BLASTP algorithm extends word hits to longer, potentially higher scoring, alignments. 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 scores above the threshold *S* and is an *HSP*.

Table 2. Alignments versus Number of Hits.

Hits	Alignments with n Hits	
n	% of Alignments	% of Extend Cycles
1	75	71
2	21	24
3	3	5
	99	100

Word hits are net positive scoring sub-alignments. The alignments that result from an extension contain the word hit from which it is 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 2 shows that virtually all extensions are alignments that contain either one, two or three word hits. As shown in Figure 11, the average score of these alignments rarely meets a score of 32, the default value for the cutoff parameter *S*. The plot indicates that for the values of word size, *W*=3, and threshold, *T*=11, the *HSP*s that meet the cutoff contain five word hits. Of the total time spent in hit extension, only a minute percentage actual extends a hit into a *HSP*.

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. 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*.

Two-Hit Effect on HSP Detection. The two-hit BLASTP program reports the same highest scoring alignments as the unmodified program. However, the two-hit BLASTP program does miss lower scoring *HSPs*. The highest scoring alignment is selected from the set of reported *HSPs*, and this

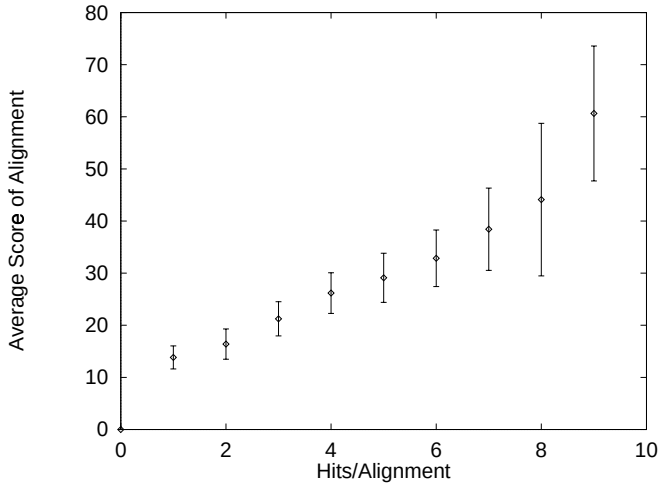

Figure 11. Average Alignment Score vs Hits per Alignment.

set is used to calculate the $P(N)$ value, and the value of N. Since the two-hit algorithm may not detect some *HSPs* that ar e detected by the unmodified algorithm, it reports lower N values. This effects the value for $P(N)$. For the experimental search this has no effect on the reported alignments since the missed *HSPs* are not the highest scoring ones.

4.4. Performance of Optimisations

Each of the optimisations results in a significant decrease in the computational time required to perform a search. Table 3 compares the program and extension performance of each optimised program with that of an unmodified program.

A single test search is used as a standard. The search program is BLASTP, version 1.4 [4]; the default values for all parameters are used. The database is the protein NRDB (non-redundant database) [5], 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 Figure 1. The average length of database sequence is 283 residues. The test search uses a DEC Alpha computer.

The column headed `Wall Clock time` is the elapsed time reported by the C library function `time`. The column headed `CPU cycles time` is the cpu time as reported by the profiler. The gain in performance is measured by the change in the number of CPU cycles. Individually, the three optimisations show speed ups of 15%, 48%, and 63% respectively. In general, these reported gains agree with the time figures. However, there is a glaring anomaly with the residue-doublet optimisation, in that its elapsed time is actually longer than the unmodified program. We do not have a clear explanation of this, but suspect it is due to the fact that the larger scoring matrix (400×400) is too large for the cache: we have no way to measure cache usage. Furthermore, we do not have clear information on how the profilers treat the timing of individual fetch instructions in the presence and absence of cache faults.

5. Conclusions

We provide a detailed description of the original BLASTP program and three separate optimisations to it. Such a description of the BLASTP program was not previously available in the literature. The description is developed using a reverse engineering process that iterates profiling, focused reading of the source code, and model construction. The execution profiles also suggest targets for optimisation.

BLASTP consists of three steps, and the optimisations target the third step, called hit extension, since it accounts for 93% of the execution time. The first optimisation alters the data representation of the query sequence and the related code for indexing the scoring matrix. The second optimisation performs extensions in step-sizes of two rather than one. The third optimisation forestalls the calling of the hit extension step in cases that are unlikely to lead to a high-scoring alignment. Individually, the three optimisations show speed ups of 15%, 48%, and 63% respectively.

There is some loss in sensitivity when one employs the optimisations. That is, there is a decreased ability of the optimised algorithm to detect *HSPs*. For the row-address optimisation, there is no loss in sensitivity at all. For the residue-doublet optimisation, there are edge effects on the alignments due to extending two residues at a time. However, these are minor, and can be easily corrected by post-processing. For the two-hit optimisation, there is generally no difference in the highest scoring alignment, however, the calculation of $P(N)$ value and N value are effected. If one is interested in all the *HSPs*, and not just the highest scoring alignment, then they are minor losses.

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 11. However, we make no claims of original discovery of this optimisation since it is implemented in the BLAST version

Table 3. Summary of Performance. (Time in seconds on DEC Alpha.)

Program	Extension CPU cycles		Total Wall Clock	Total CPU cycles		
	No.$\times 10^{10}$	time	time	No.$\times 10^{10}$	time	% Gain
BLASTP (row-address)	1.7	57.9		1.8	62.9	15
BLASTP (residue-doublet)	1.2	39.9	109	1.3	45.5	48
BLASTP (two-hit)	0.6	19.4	40	0.8	27.4	63
BLASTP (unmodified)	2.0	68.4	100	2.1	73.4	

2.0 program [2]. This paper 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.

Acknowledgements This work has been supported by the Natural Sciences and Engineering Research Council of Canada, and *Fonds pour la Formation de Chercheurs et l'Aide a la Recherche* of Québec.

Postscript We developed these optimisations independently of developments under way at NCBI for BLAST 1.4. The new BLAST version 2.0 program [2] from NCBI extends BLAST to include the two-hit optimisation, and to add gapped BLAST and PSI BLAST as methods to increase sensitivity at the loss of computation speed. The NCBI authors [2] discuss the loss in sensitivity of the two-hit optimisation, and how those losses may be detected by lowering the threshold, T, for neighborhood words. This essentially increases the probability of having two hits on the alignment.

Another independent line of development of BLAST is being carried out by Warren Gish [7], one of the original BLAST authors. This work is aimed more at increased sensitivity than at optimisations of performance.

References

[1] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215(3):403–410, Oct 1990.

[2] S. F. Altschul, T. L. Madden, A. A. Scaeffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, 1997.

[3] D. Benson, M. Boguski, D. Lipman, J. Ostell, and B. Ouellette. Genbank. *Nucleic Acids Research*, 26(1):1–7, 1998.

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

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

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

[7] W. Gish. Blast2. unpublished algorithm from Washington University, St Louis.

[8] P. James. Breakthroughs and Views — Of Genomes and Proteomes. *Biochemical and Biophysical Research Communications*, 231:1–6, 1997.

[9] D. J. Lipman and W. R. Pearson. Rapid and Sensitive Protein Similarity Searches. *Science*, 227:1435–1441, 1985.

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

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

[12] T. Smith and M. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147:195–197, 1981.

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