An Analysis of Pairwise Sequence Alignment Algorithm Complexities: Needleman-Wunsch, Smith-Waterman, FASTA, BLAST and Gapped BLAST

Alexander Chan
5075504
Biochemistry 218
Final Project

An Analysis of Pairwise Sequence Alignment Algorithm Complexities

Introduction

As databases of protein sequences and properties increase in size, it becomes more and more reliable to depend on previously classified proteins to determine the structure and function of a novel protein. One method of determining homology between two proteins is through a pair-wise sequence alignment of their primary structures. It has been found that two proteins that are homologous, such that they were evolutionarily derived from a common protein, tend to align well with a large number of identical or highly similar residues in similar positions along the sequences. Provided a large database of protein sequences and their matching functions and structures, performing a sequence alignment between a sequence of a novel protein and the proteins in the database will find other proteins which are highly related, thus potentially revealing the function of the new protein.

The challenge in performing sequence alignments has been the tradeoff between accuracy and efficiency. Older algorithms like the Needleman-Wunsch algorithm and the Smith-Waterman algorithm tend to have very high computational complexities, however manage to find the optimal alignment between a pair of proteins. Newer algorithms like FASTA and BLAST sacrifice some of this accuracy to make the alignments faster. As the database of proteins grows larger, faster algorithms become more important to be able to quickly compare a given sequence to the entire database.

In this paper, we shall look at five main algorithms: the older optimal alignment algorithms by Needleman-Wunsch and Smith-Waterman, and the newer, approximate alignment algorithms FASTA, BLAST, and Gapped BLAST. We shall look at the algorithm itself and the computational and space complexity of each algorithm. From this, we can compare the efficiencies of the various algorithms and see what sacrifices the algorithms make in exchange for speed. We shall also analyze two modifications of the older dynamic programming algorithms: affine gap penalties, and the Hirschberg improvement.

Needleman-Wunsch

The Needleman-Wunsch algorithm [1], published in 1970, provides a method of finding the optimal global alignment of two sequences by maximizing the number of amino acid matches and minimizing the number of gaps necessary to align the two sequences. Because the Needleman-Wunsch algorithm finds the optimal alignment of the entire sequence of both proteins, it is a global alignment technique, and cannot be used to find local regions of high similarity.

In pairwise sequence alignment algorithms, a scoring function, F, must exist such that different scores can be assigned to different alignments of two proteins relative to the number of gaps and number of matches in the alignment. Thus, the alignment with the largest score must be the optimal alignment. In this scoring function, let m be the score for two residues matching, s is the penalty for mismatches, and g is the penalty for inserting a gap. The Needleman-Wunsch algorithm realizes that the score of aligning the entire proteins is the same as the sum of the scores of two subsequences of the proteins,

$$F(x_{1:M}, y_{1:N}) = F(x_{1:i}, y_{1:j}) + F(x_{i+1:M}, y_{j+1:N})$$

where M is the length of sequence x, N is the length of sequence y, and $1<i<M$ and $1<j<N$. From this, we can see that the optimal score of two partial sequences is the sum of score of residue i in sequence x and residue j in sequence y, and the maximum score aligning the rest of the sequences. There are three possibilities:

- x_i and y_j are the same so $F(i, j) = s(i, j) + F(i-1, j-1)$
  *$s(i, j)=m$ if $x_i=y_j$; $s(i, j)=-s$ otherwise
- x_i aligns to a gap so $F(i, j) = -d + F(i-1\ j)$
- y_i aligns to a gap so $F(i, j) = -d + F(i, j-1)$

By taking the maximum of these three, we can get the optimal score for alignment of the two subsequences,

$$F(i,j) = score(i,j) + \max\begin{cases} F(i-1, j-1) \\ F(i-1, j) \\ F(i, j-1) \end{cases}$$

Starting at the c-terminus of the protein sequences, and working to the n-terminus, we can use scores of the subsequences it contains to find the optimal score of the subsequence plus the residue at *(i,j)*. Thus, the Needleman-Wunsch algorithm essentially creates a matrix in which the horizontal and vertical axes each correspond to one of the protein sequences. Each amino acid in the protein sequence is assigned to a row or column starting at the N-terminus. For every cell (i, j) where *i* is the row and *j* is the column, if the residue *i* is the same as residue *j*, the score *m* is entered into the matrix. In this case, let m=1, and s=d=0. In the example given in the original paper by Needleman and Wunsch, this matrix may look as follows:

		Sequence x												
		A	D	C	N	S	R	Q	C	L	C	R	P	M
Sequence y	A	1												
	S					1								
	C			1					1		1			
	S					1								
	N				1									
	R						1					1		
	C			1					1		1			
	K													
	C			1					1		1			
	R						1					1		
	D		1											
	P												1	

In this matrix, (1, 1) is the lower-right cell. To fill in the rest of the matrix, starting at (1, 1), if we want to fill in cell (*i*, *j*) we add it's initial score to the maximum score of the cells *(i-1, j-1)*, *(i-1, j)*, or *(i, j-1)*. Thus, row *m* is filled in, followed by row *m-1*, followed by row *m-2*, and so forth. To find the optimal alignment, we perform a "traceback" by starting at upper-left cell and if $x_i = y_j$ then we move to cell *(i-1, j-1)*, otherwise, we move to the larger of *(i-1, j)* and *(i, j-1)*. We mark every cell we come across. To find the final alignment, we draw a path from each of the marked residues in which $x_i = y_j$ to the next residue where $x_i = y_j$ and has a row and column strictly less than the first residue. So the final alignment looks as follows:

		Sequence x												
		A	D	C	N	S	R	Q	C	L	C	R	P	M
Sequence y	A	**8**	7	7	6	6	5	4	4	3	3	2	1	0
	S	7	**7**	**7**	6	6	5	4	4	3	3	2	1	0
	C	7	**7**	**7**	6	6	5	4	4	3	3	2	1	0
	S	6	6	6	**6**	**6**	5	4	4	3	3	2	1	0
	N	6	6	6	**6**	5	**5**	4	4	3	3	2	1	0
	R	5	5	5	5	**5**	**5**	4	4	3	3	2	1	0
	C	4	4	4	4	4	4	**4**	**4**	3	3	2	1	0
	K	3	3	3	3	3	3	3	3	**3**	**3**	2	1	0
	C	3	3	3	3	3	3	3	3	**3**	**3**	2	1	0
	R	2	2	2	2	2	2	2	2	2	2	**2**	1	0
	D	2	2	1	1	1	1	1	1	1	1	1	**1**	0
	P	1	1	1	1	1	1	1	1	1	1	1	**1**	0

From this, we can see that when an arrow skips a row, it is a gap in sequence x, and when it skips a column, it is a gap in sequence y. This particular example also has two optimal sequence alignments. The final result looks as follows:

```
ADC-NSRQCLCR-PM
|  |  |  |  ||  |
ASCSN-R-CKCRDP-
```

To analyze the time complexity of the Needleman-Wunsch algorithm, we can essentially analyze each individual part of the algorithm. To initialize the matrix, we need to input the scores of the row 0 and column 0 with $-j*d$ and $-i*d$ respectively (in our example, these are both 0). This has a time complexity of $O(M+N)$.

The next step is filling in the matrix with all the scores, $F(i,j)$. For each cell of the matrix, three neighboring cells must be compared, which is a constant time operation. Thus, to fill the entire matrix, the time complexity is the number of entries, or $O(MN)$.

Finally the traceback requires a number of steps. The first step is marking the cells according to the rules above. We can move a maximum of N rows and M columns, and thus the complexity of this is $O(M+N)$. The second step is finding the final path which involves jumping from cells of matching residues. Since this step can include a maximum of N cells (where we assume $N>M$), this step is $O(N)$.

Thus, the overall time complexity of this algorithm is

$$O(M+N)+O(MN)+O(M+N)+O(N)=O(MN)$$

Since this algorithm fills a single matrix of size MN and stores at most N positions for the traceback, the total space complexity of this algorithm is $O(MN)+O(N)=O(MN)$.

It is important to note here that the Needleman-Wunsch algorithm supports different scores for exact residue matches, similar residues, and gaps. A PAM or BLOSUM weight matrix can be used to weight residue matching scores. These weighted scores can affect the final alignment of the two protein sequences and the biological relevance of the alignment, but will not affect the time or space complexity of the algorithm because the number of operations will not change. This alignment is limited, however, because it can only align entire proteins. A different algorithm was developed to create local alignments

Smith-Waterman

The Smith-Waterman algorithm was published in 1981 [2] and is very similar to the Needleman-Wunsch algorithm. Yet, the Smith-Waterman algorithm is different in that it is a local sequence alignment algorithm. Instead of aligning the entire length of two protein sequences, this algorithm finds the region of highest similarity between two proteins. This is potentially more biologically relevant due to the fact that the ends of proteins tend to be less highly conserved than the middle portions, leading to higher mutation, deletion, and insertion rates at the ends of the protein. The Smith-Waterman algorithm allows us to align proteins more accurately without having to align the ends of related protein which may be highly different.

The Smith-Waterman algorithm can be implemented by changing only a couple things in the Needleman-Wunsch algorithm. The algorithm can be written using the pseudocode below.

Initialization:

```
F(0,j) = 0
F(i,0)= 0
```

Filling Matrix:

```
for each i,j = 0 to M,N {
      F(i,j) = max(0,
                  F(i-1,j-1) + s,
                  F(i-1,j) – d,
                  F(i,j-1) – d,)   }
```

Traceback:

```
F_OPT = max(F(i,j))
traceback(F_OPT)
```

Only two things were changed in the Needleman-Wunsch algorithm to obtain the Smith-Waterman algorithm. When filling the matrix, we do not let any of the matrix values become negative, and thus we consider 0 as potentially being the maximum value of the three other cases (where $x_i = y_j$, or there is a gap in x or a gap in y). By not letting any of the values go below zero, we stop considering regions of high dissimilarity which have no good alignments. This allows the algorithm to focus on only those regions of the protein which are similar. The second change in the algorithm is in the traceback. Instead of starting at the n-terminus of both sequences, we start at the cell with the highest score in the entire matrix. This allows for the alignment of the similar subsequences of the proteins.

The complexity of the Smith-Waterman algorithm can also be computed. The time complexity of the initialization is $O(M+N)$ because we need to initialize row 0 and column 0. In filling the matrix, we traverse each cell of the matrix and perform a constant number of operations in each cell, and thus the time complexity for this part is $O(MN)$. Thus far, the complexity of the Smith-Waterman algorithm is exactly the same as that for the Needleman-Wunsch algorithms. However, in the traceback, the algorithm requires the maximum cell be found, and this must be done by traversing the entire matrix, making the time complexity for the traceback $O(MN)$. It is also possible to keep track of the largest cell during the matrix filling segment of the algorithm, although this will not change the overall complexity. Thus the total time complexity of the Smith-Waterman algorithm is

$$O(M+N)+ O(MN) + O(MN) = O(MN)$$

which is identical to the complexity of the Needleman-Wunsch algorithm. The overall running time of this algorithm is actually slightly slower than the Needleman-Wunsch algorithm however, because more comparisons must be made when comparing the scores to 0, and when finding the largest cell during the traceback.

The space complexity of the Smith-Waterman algorithm is also unchanged from the Needleman-Wunsch algorithm. This is due to the fact that the same matrix is used and the same amount of space is needed for the traceback. Thus, there is no definite space or time advantage of one algorithm over the other. However, the Smith-Waterman algorithm tends to model protein homology better because it ignores misalignments at the ends of the proteins which are often not highly conserved. Thus, database searches are usually done with the Smith-Waterman algorithm over the Needleman-Wunsch algorithm which tends to model homology better in distantly related proteins. The Needleman-Wunsch algorithm will tend to be better for proteins which are closely related, with fewer mutations because the ends of the protein in closely related sequences will not be changed significantly.

Affine Gap Penalty

In the Needleman-Wunsch and the Smith-Waterman algorithms, there existed a constant gap penalty, d, for a single missing or inserted residue. Thus, to insert a gap of size l, the total penalty

would be *d*l*. However, in biological systems, a deletion or insertion of a large number of residues may be significantly less rare than this, and thus, a different model of gap penalties must be used.

Realistically, gaps of different sizes would all have different penalties, but using this model increases the complexity of either algorithm from *O(MN)* to $O(M^2N)$. This is because when computing the score of each cell, instead of finding the maximum of three adjacent cells, we must find the number of cells to the right or down which also are included in the gap. Thus, we must look at *i+j+1* cells, which increases the time complexity to $O(M^2N)$.

To get around this increase in complexity, we can use affine gap penalties in which the initial gap opening penalty is set at a constant value, *d*, and extending the gap by a single residue is set at a constant, lower value, *e*. This linear gap penalty function is easier to deal with. In this case, we must keep track of two things for each cell in the matrix. We must keep track of the score of the aligned subsequences $x_{1:i}$ and $y_{1:j}$ plus the score of aligning x_i and y_j. We can store these values in matrix *F(i,j)*. We must also keep track of the score of the aligned subsequences $x_{1:i}$ and $y_{1:j}$ plus the score of inserting a gap at either x_i or y_j. We can store these values in *G(i,j)*. To fill in each of the cells in both matrices, we apply the following rules,

$$F(i,j) = \max\begin{cases} F(i-1,j-1) + score(i,j) \\ G(i-1,j-1) + score(i,j) \end{cases} \qquad G(i,j) = \max\begin{cases} F(i-1,j) - d \\ F(i,j-1) - d \\ G(i,j-1) - e \\ G(i,j-1) - e \end{cases}$$

Here *F(i,j)* is the max score when x_i and y_j are aligned (either ending a gap at *G(i-1,j-1)*, or continuing an alignment in *F(i-1,j-1)*). *G(i,j)* is the max score when either starting a gap in *F* with a penalty of *d* or extending a gap in *G* with an extension penalty of *e*.

The initialization, in this case, is also *O(M+N)* because row 0 and column 0 must initialized to the linear gap penalty, *d+(j-1)e* or *d+(i-1)e* respectively. In the iterative phase, we now have two matrices to fill, but each cell of both matrices still only requires a constant number of operations. Each matrix has a time complexity of *O(MN)* yielding *2O(MN) = O(MN)* complexity. Finally, the traceback is still *O(M+N)* because it is unchanged. Thus, the total time complexity is *O(MN)* which is the same as the Needleman-Wunsch and Smith-Waterman complexities.

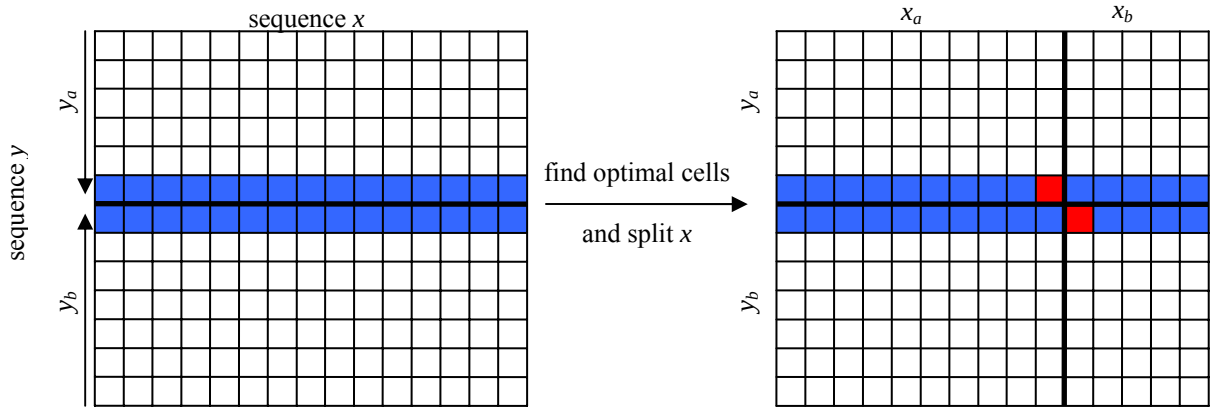
The space complexity must take into account both matrices and the space needed for traceback on both matrices. Since the space complexity of a single matrix is *O(MN)*, the space complexity for two matrices is *2O(MN)=O(MN)*. Thus, the space complexity is also unchanged. However, the actual space used is two times the space used for Needleman-Wunsch and Smith-Waterman, and the running time is also about two times as long for the affine gap model. Thus, we see that increasing biological accuracy involves a sacrifice in efficiency.

Hirschberg Improvement

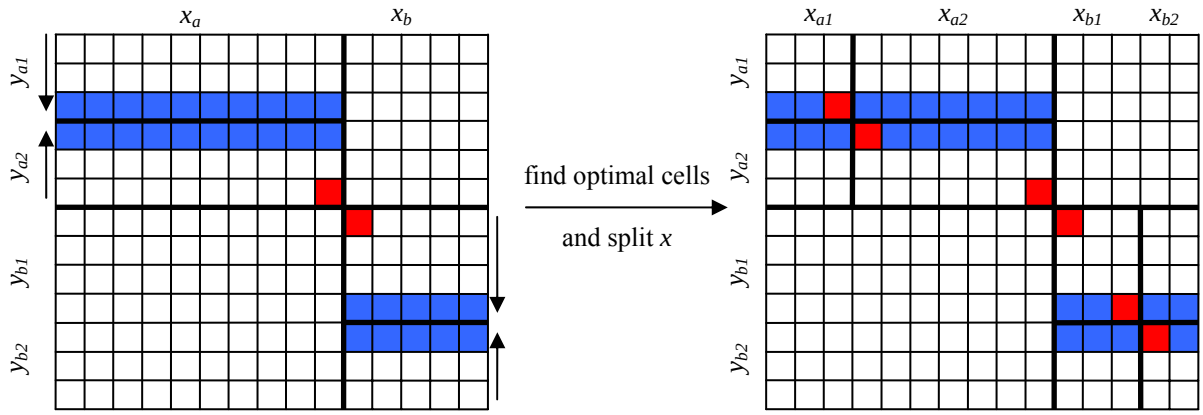
In 1975, Hirschberg published an improvement [3] to pairwise sequence alignment which allowed the space complexity to be lowered to *O(N)*. This linear space algorithm keeps the time complexity at *O(MN)* and thus is a significant improvement over the traditional Needleman-Wunsch and Smith-Waterman algorithms. The Hirschberg algorithm is a recursive algorithm and divides the matrix into smaller parts solving each of the smaller problems separately.

Given two sequences *x* and *y* (assuming *x* is longer), the Hirschberg algorithm splits sequence *y* near the middle resulting in two subsequences y_a and y_b. Then a pairwise sequence alignment is performed from the bottom-right of subsequence y_b and from the upper-left of sequence y_a. Hirschberg realized that to fill in a single row of the matrix, only the row below it is needed. Thus the algorithm fills in the matrix, yet only stores the row it is working on, and the row below it until it reaches the top of y_b. For y_a, the algorithm stores the row above the row it is working on, and stops when it reaches

the bottom of y_a. Once the algorithm finishes for both y_a and y_b, it looks for the point where the optimal paths converge, and splits sequence x into two subsequences, x_a and x_b, at this point.



After splitting sequence x into x_a and x_b, the algorithm is run again on the upper-left submatrix and the lower-right submatrix. The lower-left and upper-right do not need to be analyzed further because we know that when aligning sequences, the alignment must always be decreasing and thus will not be found in this sections. The matices are split recursively until the entire optimal path is revealed.



The Hirschberg method only requires enough space to store the row it is working on, *O(N)*, the previous row, *O(N)*, and all the found optimal cells, *O(M+N)*. Assuming the longer sequence is length N, the space complexity of the Hirschberg algorithm is,

$$O(N)+O(N)+O(M+N)=2O(N)+O(2N)=O(N)$$

Thus, we have reduced the space complexity from *O(MN)* to *O(N)*. The time complexity of this algorithm must take into account the recursive aspect of the method. The first iteration requires a scanning of the two halves of the matrix and it thus $O\left(MN\cdot 2\left(\frac{1}{2}\right)\right)$. The second iteration works on two matrices which are about a quarter of the size of the total matrix and thus the second iteration has a complexity of $O\left(MN\cdot 2\left(\frac{1}{4}\right)\right)$. Adding up the complexities of the entire algorithm yields,

$$O\left(MN\cdot 2\left(\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\ldots\right)\right)=O(2MN)=O(MN)$$

Therefore, the Hirschberg algorithm lowers the space complexity to a linear function of the longer sequence length, however does not significantly raise the time complexity; it roughly doubles the running time but keeps the complexity at *O(MN)*.

Heuristic Methods

Thus far, we have discussed optimal sequence alignment methods which find the highest scoring alignment for any pair of protein sequences. However, these algorithms tend to be slow, and when searching an entire database, these methods are often too slow to perform a search in reasonable time. Thus, heuristic, or approximate, algorithms like FASTA and BLAST were developed to speed up the process while attempting to keep as much sensitivity as possible.

FASTA

The FASTA algorithm was developed in 1985 by Lipman and Pearson [4]. Unlike the Needlman-Wunsch and Smith-Waterman algorithms, FASTA approximates the optimal alignment by searching and matching *k-tuples*, or subsequences of length *k*. The algorithm assumes that related proteins will have regions of identity, and by searching with *k-tuples*, the FASTA algorithm allows small regions of local identity to be found quickly. For proteins, these k-tuples tend be of length two.

FASTA creates a hash table of all possible *k-tuples* and goes through the entire query protein of length *N* and inputs the location of all the *k-tuples* into the table. Each *k-tuple* in the database sequence can be looked-up in the hash table, and any matches will allow the algorithm to mark the matching cells in the matrix. This results in a matrix in which all points of local identity of length *k* are marked.

The FASTA algorithm then identifies the ten highest scoring diagonal runs by identifying each marked point on the matrix, and adding a positive score for every other marked cell along a diagonal, and subtracting a penalty for unmarked cells between marked cells along the diagonal. These ten highest scoring segments are kept, and all other segments of local alignment are discarded. The ten diagonals are scored once again using an amino acid weight matrix (PAM or BLOSUM matrix) and any diagonals with scores below a threshold are discarding again. The highest scoring diagonal is termed $init_1$. Thus, we are left with ten or fewer regions in which the two proteins align with no gaps (although mismatches are allowed in the form of missing marked cells along the diagonal).

The algorithm then calculates the scores of joining every combination of diagonals, as long as the diagonals are downstream from one another. To calculate the score of a joined series of diagonals, the individual scores of each of the diagonals are summed, and a constant joining penalty is subtracted each time two subsequences are joined. The maximum of the joined alignments is termed $init_n$ and joined alignments which have a score below a threshold are discarded. The $init_n$ scores are used to rank each of the alignments with sequences from the database.

The final step in creating the sequence alignment is to define a diagonal band of around 32 residues wide around the $init_1$ diagonal from the upper-left of the matrix to the lower-right. The FASTA algorithm assumes that the optimal alignment will include or be near the $init_1$ diagonal. A dynamic programming algorithm is then performed in this band to find the final optimal alignment, and it essentially merges the regions of local alignment into a single alignment.

The FASTA algorithm is substantially faster than the Needleman-Wunsch or Smith-Waterman alignments and thus can be more easily used in database queries. However, the time complexity of this algorithm does not seem to suggest this. In building the hash table, each *k-tuple* of the query sequence needs to be scanned and inserted into the table. Since there are $N–k+1$ *k-tuples*, this yields an $O(N)$ complexity. Next, every *k-tuple* in the database sequence must also be compared to the hash table, and since the lookup and marking of cells can be done in constant time, the time complexity depends on the length of the database sequence, $O(M)$.

Then the algorithm creates diagonals from marked cells, and thus the running time depends on the number of marked cells. In the worst case, all cells could be marked yielding an $O(MN)$ complexity. However, this is rarely ever the case. Assuming the probability of a *k-tuple* of amino acids exactly matching another *k-tuple* of amino acids to be $\frac{1}{20^k}$ (which is not completely correct), and assuming the actual alignment takes around $M+N$ cells, we have lowered the number of cells

considered from *MN* to $M + N + \frac{MN}{20^k}$ which is a significant decrease. We then are left with ten or fewer diagonals after each of the scores have been computed.

In the joining step, we must consider at most every combination of diagonals with other diagonals. If all the diagonals are strictly decreasing, we can combine any combination of them, and the total number of combinations must be less than or equal to ${}_{10}C_1 + {}_{10}C_2 + {}_{10}C_3 + ... + {}_{10}C_{10} = 1023$. However, many diagonals cannot be combined with others because the start of one diagonal may not be to the lower-right of the other, not allowing them to be joined. Nonetheless, the number of computations in the joining phase is bound by 1023 and is not dependent on sequence length.

Finally, the dynamic programming algorithm is run on the band of cells 32 residues wide. Assuming the maximum diagonal of the matrix is $\sqrt{M^2 + N^2} \approx M + N$, the total number of cells considered is about $32(M + N)$. Thus, in the worst case, the time complexity of FASTA is,

$$O(N) + O(M) + O(MN) + O(1023) + O(32(M + N)) = O(MN)$$

However, this algorithm is *O(MN)* only because when creating the diagonals, in the worst case, *MN* cells must be analyzed. But on average, very few cells of the total matrix are marked. Thus, the average-case complexity would be about,

$$O(N) + O(M) + O\left(M + N + \frac{MN}{20^k}\right) + O(1023) + O(32(M + N)) = O\left(\frac{MN}{20^k}\right)$$

Thus, the complexity of the FASTA algorithm depends on the size of the *k-tuples*, and the larger the *k-tuples*, the faster the algorithm. The true complexity is difficult to determine because the speed at which the algorithm can align two sequences depends on total number marked cells and the total number of diagonals which is extremely variable. For example, it is possible for the FASTA algorithm to produce no alignment even if the proteins are very similar. If the proteins have *k-1* exact matches in a row followed by a single mismatch, repeating the entire length of both proteins (meaning the proteins will be 75% identical), no *k-tuples* will ever match and thus no alignment can be produced.

The space complexity of this algorithm is also *O(MN)* like the Needleman-Wunsch and Smith-Waterman because it uses a matrix. However, it is possible that less space will be used because not all cells in the matrix are marked, and thus marked cells can be represented in a different, non-matrix, form.

Although the FASTA algorithm is faster than any of the previous algorithms, it is not guaranteed to find the optimal alignment between two proteins. Because it uses *k-tuples* to speed things up, it can miss smaller areas of similarity and thus there is a possibility of misaligning two proteins. Also, limiting the width of the area searched with dynamic programming can yield a sub-optimal alignment if gaps are wider than the width of the band. Thus, it is clear that there is a tradeoff here between speed and sensitivity. Yet with even the speed of the FASTA algorithm is not enough to keep up with the exponentially growing protein databases, and thus a faster algorithm is necessary.

BLAST

The BLAST (Basic Local Alignment Search Tool) algorithm was developed by Altschul et al. in 1990 [5] and similar to the FASTA algorithm, is also a heuristic pairwise sequence aligner. However, the basis of the BLAST algorithm is the use of words and High-scoring Segment Pairs (HSPs) instead of *k-tuples*.

BLAST begins by finding all words, or subpeptides of length *w* (typically 3), which exist in the protein sequence. Using a substitution matrix, a list of other words, called a neighborhood, is created for each word found in the protein sequence; these words must be related to the original word and must have a substitution matrix score higher than *T*, else they are not considered. For fast access to these data, the word positions are entered into a hash table. The each word in the database sequence can be

compared to the hash table, and only those matches which are deemed statistically significant by a statistical method developed by Karlin and Altschul[6], will be kept. This significantly reduces the number of hits which must be analyzed.

Every match of a word in the database sequence with one of the neighbor words is called a High-scoring Sequence Pair (HSP) and these act as "seeds" to start a local sequence alignment. The algorithm extends the alignment (without gaps) to the left and the right of the seed and calculates the score of the alignment at every residue with the substitution matrix. The algorithm stops extending the alignment once the score decreases a quantity *X* from the maximum score found at any point in the alignment; this is possible because scores can easily be negative. If the final score of the local alignment is below a threshold value, called *S*, the alignment is discarded. Here, again, statistical significance is calculated and statistically insignificant alignments are discarded.

To the determine the statistical significance of an HSP, the expected number of HSPs with a score larger than *S* is calculated via the formula,

$$E = KMNe^{-\lambda S}$$

Here, K is parameter for the search space size. λ can be calculated by solving, $\sum_{i=1}^{r}\sum_{j=1}^{r} p_i p_j e^{\lambda s_{ij}}$ where p_i and p_j are the probabilities of each of the residues in the sequence and s_{ij} is the substitution score from the substitution matrix (PAM or BLOSUM).

Once all the low scoring and statistically insignificant HSPs are discarded, the highest scoring HSP is kept. This highest scoring segment created by extending the seeds is called the Maximal-scoring Segment Pair (MSP). Since the alignment contains no gaps, the MSP is used to build the final alignment by simply elongating this segment to the end of the protein.

The computational complexity of the BLAST algorithm can also be calculated. First, in finding the neighborhood of words, we must consider all $N–w+1$words in the query sequence. To find all the neighbors, we must compare each word with all other combinations of words and score them. Since there are 20^w possible words, we must compare a total of $N(20^w)$ strings. Although this seems like a huge amount, this step in the alignment must only be performed once on the query sequence regardless of the number of database sequences which will be aligned against it. Thus, the complexity of the pre-alignment algorithm is,

$$O(N)+O(N(20^w))= O(N(20^w))$$

The actual alignment part of the algorithm which is performed for every database sequence has a very different complexity. To find the seeds, each of the words in the database sequence must be compared to hash table created for the neighbors of the query sequence words, and thus we must perform *M* lookups. The end product of the *M* lookups is on the order of *N* seeds total, because there are only $N–w+1$ words in the query. Each of these seeds starts an alignment, and the maximum length of the alignment is the length of the query sequence, *M*, assuming $M<N$. Since calculating λ must only be done once, and calculating the statistical significance of each HSP is a constant time operation, these have a complexity of $O(1)$. Thus, the total complexity of the BLAST algorithm is,

$$O(M)+O(MN)+O(1)=O(MN)$$

This is the same time complexity as all of the other algorithms, however, using the statistically significant elimination of HSPs and words, BLAST significantly lowers the numbers of segments which need to be extended and thus make the algorithm run faster than all the previous algorithms. Since the expected number of HSPs is $E = KMNe^{-\lambda S}$, it is clear that the actual number of segments which must be considered is small and depends on the threshold values *S* and *T*. Thus, the higher *S* and *T* are, the faster the algorithm runs.

The space complexity of both of these algorithms is different than any of the previous algorithms. We must first take into consideration the hash table. The table contains 20^w rows, one for every possible word of length *w*. The rows contain the locations for each of the words, and the total

number of positions is on the order of *N*. Thus, there should on the order of N seeds which can each lead to a local alignment of a maximum of length M. The total space complexity is,

$$O(20^w)+O(N)+O(MN)=O(20^w+MN)$$

Thus, the space complexity is slightly higher than the other algorithms, however the actual space used may not be significantly larger than the dynamic programming algorithms. This is because many of the local alignments will be discarded because they do not meet the threshold, and also because the alignments which do meet the threshold will significantly shorter than length *M*.

However, BLAST only produces ungapped alignments and is not the most biologically relevant algorithm for protein sequences which may have insertions or deletions. Thus, a modification to BLAST, called Gapped BLAST was introduced. Gapped BLAST [7] uses a "two-hit" approach in which a word can be followed by a second word which is within a certain gap threshold. These matches will then essentially be extended using a dynamic programming algorithm in all directions until the score falls below a certain percentage threshold of the highest score computed.

Since the same words and hash table are used in this algorithm, the pre-alignment complexity is unchanged. The two hit approach also does not significantly increase the complexity because the database sequence only needs to be scanned a small, constant distance to the left and right of already found words to find a second match. Thus the complexity of finding seeds is still *O(MN)*. Finally, the dynamic programming portion of the algorithm is O(MN). Thus the total complexity of the algorithm is,

$$O(M)+O(MN)+O(MN)=O(MN)$$

This improvement does not change the overall computational complexity of the algorithm, however, it does make the algorithm slower. Yet Gapped BLAST creates more biologically relevant alignments than the ungapped BLAST algorithm; insertions and deletions are common in protein evolution.

From this analysis, it is clear that BLAST is significantly faster than the older, slower algorithms, yet BLAST does not always give the optimal alignment. It is possible for blast to miss segments of similarity smaller than the word size, and ungapped BLAST often produces alignments which are not biologically relevant. Gapped BLAST can also produce suboptimal alignments because when it performs the dynamic programming at the end, the best alignment may lie outside of the limited band. Thus, it is clear that a tradeoff exists between the sensitivity of an algorithm and the speed at which it works.

Conclusion

It is interesting to note that all of the algorithms essentially had the same computational complexity of *O(MN)*, except BLAST which had a pre-alignment complexity of $O(20^w)$. Yet despite this, each of the algorithms had very different running times, with BLAST being the fastest and the dynamic programming algorithms being the slowest. The space complexities of all the algorithms was also essentially identical, around *O(MN)* space. It is also clear that BLAST and FASTA had to make sacrifices in specificity by considering longer subsequences, *k-tuples* and *words*, to be able to achieve higher speeds. Thus, a tradeoff exists between speed and sensitivity. However, the optimal alignment is not necessarily the best biological alignment, and thus the sacrifice of accuracy in exchange for speed may not be as harmful as it may seem.

Modeling biological relevance is a difficult task and also involves tradeoffs as seen in the affine gap algorithm presented here. To be able to model these gap properties correctly, a slightly more complex, more time consuming algorithm was necessary. Also, the Gapped BLAST modification to BLAST was also more complex than the original algorithm and required more computations and space. Thus it is clear that in pairwise sequence alignment, we must come to a compromise to be able to efficiently align sequences in a biologically relevant manner in a reasonable amount of time.

References

1. Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. J. Mol. Biol. 48, 443-453.

2. Smith, T. F. and Waterman, M. (1981). Identification of common molecular subsequences. J. Mol. Biol. 147, 195-197.

3. Hirschberg, D.S. (1975). A Linear Space Algorith for Computing Maximal Common Subsequences. *Comm. ACM.* 18(6), 341-343.

4. Lipman, D.J. and Pearson, W.R. (1985). Rapid and Sensitive Protein Simlarity Searches. Science 227, 1435-1441.

5. Altschul, S. F., Gish, W., Miller, W., Myers, E. W. and Lipman, D. J. (1990). A Basic Local Alignment Search Tool. *J. Mol. Biol.*, 215, 403-410.

6. Karlin, S. and Altschul, S. F. (1990). Applications and statistics for multiple high-scoring segments in molecular sequences. *Proc. Nat. Acad. Sci.* 87, 2264-2268.

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