
ALIGNMENT ALGORITHMS IN DNA SEQUENCING

ECE 573: Data Structures and Algorithms
Spring 2016

Department of Electrical and Computer Engineering
Rutgers, the State University of New Jersey, NJ, USA.

Niloy Talukder **Faisal Mohammad** **Will Cheng**

Table of Contents

Introduction and Problem Statement

DNA (deoxyribonucleic acid) is a molecule that is made up of sequences of organic subunits called nucleotides. These sequences of nucleotides, when translated through bio-cellular processes, code for certain amino acids, which are the primary backbone for proteins. Thus, DNA sequences are biological instructions that are responsible for the development of all biological structures and species. However, for computational biologist, or scientists that work on studying these sequence at the pattern-based level, abstractions are developed. These abstractions portray the nucleotides of the sequences (Adenine(A), Guanine(G), Cytosine(C), and Thymine(T)) as a sequence of strings and letters eg. “AGTCGC”. Doing so allows scientists in bioinformatics to compare and contrast DNA sequences of one biological entity with another without worrying about the exact biology until after alignment.

Though some DNA sequences are only a thousand base-pairs in size, many sequences are as long as ten-million and more. When comparing DNA sequences, the goal is to find regions of similarity where segments of one sequence matches[1] with the other. Finding similarities between two or more DNA sequences of various species can help us understand and study the structural, biological, and evolutionary relationship between organisms. Human effort alone can only get us so far with a pencil and paper, thus, we need the computational power of algorithms. Several algorithms have been developed to produce a technique called *sequence alignment*. Such algorithms vary in their time efficiency, input sizes, and their general complexity. In this paper, we will aim to address the power of *dynamic programming* algorithms, mainly through methods referred to as global and local alignment, as well as their drawbacks, and then conclude with a discussion of modern day alignment softwares used today in laboratories. Our main problem to address is obtaining the balance of optimality under time constraints. Can we give up trivial alignments to achieve faster results? What techniques work best when the sequences differ in size greatly? vice versa.

One additional factor to mention is that such sequence alignment algorithms do not only apply to DNA sequences. Proteins are essentially comprised of long chains of amino acids. There are 20 essential amino acids as opposed to the 5 nucleotides (Uracil (U)). A complex sequencing tool called BLAST (Basic Local Alignment Search Tool), which will be discussed later, actually possesses a family of algorithms designed for specific genetic input sequences, e.g. nucleotide/nucleotide, protein/protein, translated nucleotide/protein, etc **[1]**. Though our goal is to address the algorithms’ use with nucleotide sequences specifically, paying attention to the mechanism of the algorithms allows the reader to ponder about the various applications where computational alignment techniques may be useful outside of bioinformatics **[2]**.

[1] Terminology: match: if two nucleotides are aligned and are the same; mismatch: if two nucleotides are aligned but are not the same; “indels”: stands for INsert/DELetion in sequences. Depicted as gaps in sequence alignments; please see page 3 for further details.

Recursion and Dynamic Programming

Recursion is often used where the solution to a problem depends on solutions to smaller instances of it. The idea is useful when the size inputs decrease dramatically such as in merge-sort where the input size is halved every call to sort. However, there is no way to work with cutting the sequence in half with sequence alignment. We are aligning and matching, and thus everything in one sequence needs to be checked with another sequence. Recursion would be a terrible solution, and usually is when the input size doesn't decrease dramatically each call. Thus, we bring forth the idea of dynamic programming. Dynamic programming (DP) is a technique to solve problems which exhibit a specific structure where a problem can be broken down into sub-problems which are similar to the original problem.

One particular method used in DP sequence alignment algorithms is memoization **[3]**. Memoization in DP is the act of storing sub-problem solution values to a lookup table so that they can be subsequently retrieved without needing to repeat same computations. Memoization reduces the time complexity of a previously exponential running time algorithm to an ordered N polynomial algorithm (eg. $O(N), O(N^2), O(N^3)$). This is because the lookup table saves us from having the computations done over and over again for each call to the algorithm.

Global and Local Alignment

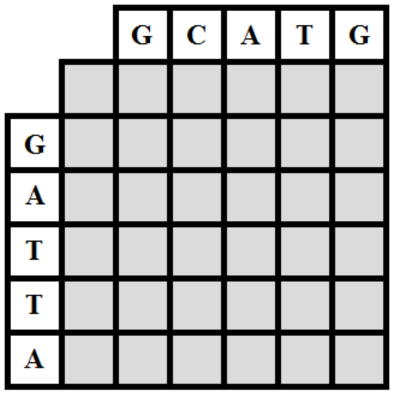
Following the brief discussion of dynamic programming, there are two foundational methodologies for sequencing alignment: global and local alignment. Understanding their purpose is obvious in the name. The term *global* usually equates to the matter of entirety; whereas *local* refers to a region within a bigger one. In global alignment, the two sequences are aligned in their entirety. Intuition may point at this method as weak or 'too bold', which is true in some cases if the two sequences are not similar and have unequal size lengths. Global alignment works best when the two sequences are similar, and roughly equal in length. Local alignment on the other hand attempts to align portions of one sequences with the other sequence in order to detect regions of similarities between genes. Local alignment is useful when two sequences contain both regions of similarities and lack thereof. The common global alignment algorithm is the Needleman-Wunsch (NW) algorithm which utilizes dynamic programming. Likewise and very similar to the NW algorithm is the Smith-Waterman (SW) algorithm, the standard local alignment algorithm **[4]**. Both algorithms are discussed in detail in later sections.

Scoring Matrix and Substitution Matrix

A scoring matrix is the main data structure of the algorithm. Represented as a 2D array of integers in code, the size of the matrix is N x M, where N is the length of the first sequence and

M is the length of the second sequence. Thus, we can already confirm a space complexity of $O(MN)$ so far. For the algorithms that follow, certain operations are performed on the matrix's cells as to invoke each algorithm's functionality. Shown below is a scoring matrix of the following input sequences: *GCATG* and *GATTA*. To the right of the figure 1.1 is a sample substitution matrix[2] for a simple scoring scheme.

A substitution matrix provides the scoring guideline for matches and mismatches in alignment. Usually a match results in adding a positive score to the overall alignment score, and a mismatch results in a negative score. The scoring guideline may vary. When aligning sequences, if a bioinformatics scientist thinks mismatches are highly detrimental to alignment, he/she may choose to assign a higher negative penalty to mismatches so that only sequence alignments with the least mismatches are returned. The one shown below is a very simple example. Substitution matrices used often in the laboratory are much more complex. Two well-known types of such matrices are the PAM (Point Accepted Mutation) matrix and BLOSUM (Block Substitution Matrix) matrix **[5]**. The PAM matrix is often used for protein sequence alignment for amino acids. As mentioned in the introduction, there are many more amino acids than there are nucleotides. In fact, constructing the PAM matrix involves making a 20 x 20 matrix, with each cell containing a value correlated to the probability of the amino acid in the ith row before a mutation being aligned with a jth column amino acid after the mutation **[5]**. The discussion of the specification and full use of that matrix, as well as BLOSUM is beyond the scope of this paper. However, both are commonly used in current sequence alignment tools such as BLAST **[1]**. As explained before, the matches in the matrix indicate a nucleotide-nucleotide alignment, and thus there is a positive value associated, +5. We can also confirm every alignment with a mismatch using this substitution matrix will receive a penalty of -3. This scoring scheme is eventually used later for our experimentation stage.



	A	G	C	T
A	5	-3	-3	-3
G	-3	5	-3	-3
C	-3	-3	5	-3
T	-3	-3	-3	5

Figure 1.1: The scoring matrix (left) is a (N+1)x(M+1) matrix corresponding to the sequence lengths of N and M. To the right is a sample substitution matrix with match score of +5 and mismatch penalty of -3.

[2] Substitution matrix is sometimes also referred to as *similarity matrix*.

Needleman-Wunsch Algorithm

Developed and published by Saul B. Needleman and Christian D. Wunsch in 1970 **[6]**, the NW algorithm is a widely used global alignment technique in DNA sequencing. Therefore, it is optimal for end to end matching between two DNA sequences. First an empty scoring matrix is created of dimensions (N+1) x (M+1) with N being the length of one sequence and M being the length of the other. Along with the substitution matrix, we define a gap penalty d. Gaps in alignments are the result of insert or deletions made with either sequence during alignment. Though gaps play a role in creating more possible matches, they themselves are not a good indicator of proficient alignment. Thus, we aim to keep the number of gaps in the final sequence alignment to be minimal as possible. We do so with the gap penalty, usually associated with a negative score. The substitution matrix S will be needed for calculating each cell of the matrix.

The algorithm works in the following sequence: initialization, matrix fill, and traceback. During the initialization phase, the first row and column are filled entirely left to right and top to bottom in the following order respectively:$0, d, 2d, 3d \ldots Nd$, and $0, d, 2d, 3d \ldots Md$. Once this is done the next phase is to fill every cell starting from the $\delta_{1,1}$ cell. This is where the core of the algorithm is introduced, as shown below. Let the value for the current cell δ_{ij} be the maximum of: the value of the cell one indices diagonal to it, $\delta_{i-1,j-1}$, plus the score for the match/mismatch at the current cell (eg. S(G,G) = 5, S(A,C) = -3); the value of the cell directly to the left of it $\delta_{i,\,j-1}$ plus the gap penalty; and the value of the cell directly above it $\delta_{i-1,\,j}$ plus the gap penalty.

$$\delta_{ij} = \text{MAX}\left\{\begin{array}{ll} \delta_{i-1,\,j-1} + S(A_i,B_j) & \leftarrow \text{ value of top-left diaganol cell plus match/miss score} \\ \delta_{i,\,j-1} + d & \leftarrow \text{ value of left cell plus gap penalty} \\ \delta_{i-1,\,j} + d & \leftarrow \text{ value of top cell plus gap penalty} \end{array}\right.$$

As we perform the mathematical operations above, it is important to note that there are two comparisons that can be made amongst the three possible choices. First, we compute the maximum of any of the two: eg. $a = max(\delta_{i-1,\,j} + d,\ \delta_{i,\,j-1} + d)$ and then compare the result with the remaining one: $\delta_{ij} = max(a,\ \delta_{i-1,\ j-1} + S(A_j, B_j))$. While picking the maximum at each cell, we also keep track of the cell (to the left, top, or diagonal) where the maximum value was obtained from. This practice of keeping back pointers is essential for the trace back phase and is consistent with the practice of dynamic programming. We can think of back pointers as arrows pointing from the current cell to the cell where the maximum value was obtained from. Keeping track of back pointers requires the creation of a trace back matrix simply to store the pointers at each cell.

Finally, the trace back phase of the algorithm works in reverse order starting from the cell with the bottommost-right score and going towards topmost-left to find the final path. The trace back requires following the arrows to the top, to achieve the best possible alignment. Thus, the horizontal and vertical arrows indicate gaps in the alignment, and the diagonal arrows indicate match and mismatches. NW algorithm provides us with the best possible global alignment. But what about the best possible local alignment?

Smith-Waterman Algorithm

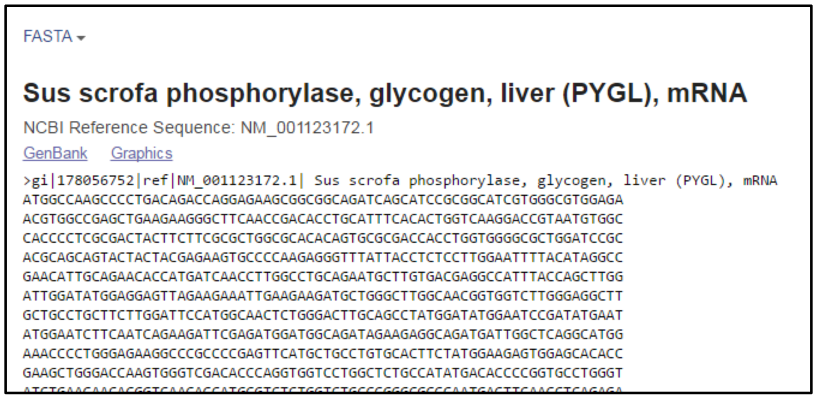
Smith-Waterman algorithm works very similar to the NW algorithm with minor adjustments made to the initialization, matrix-filling and trace back phase. For the initialization phase, the entire first row and column is set to zero. Shown below is the main algorithm that runs within each matrix cell $_{ij}$ of the scoring matrix after the first row and column.

$$\delta_{ij} = \mathrm{MAX} \begin{cases} \mathbf{0} & \leftarrow \text{added comparison, selected if rest are negative} \\ \delta_{i-1,\,j-1} + S(A_i,B_j) & \leftarrow \text{value of top-left diagonal cell plus match/miss score} \\ \delta_{i,\,j-1} + d & \leftarrow \text{value of left cell plus gap penalty/indels} \\ \delta_{i-1,\,j} + d & \leftarrow \text{value of top cell plus gap penalty/indels} \end{cases}$$

Once a maximum value is selected, a trace back matrix records the cell from which the maximum value of the current cell's score was obtained from. This use of back pointers, again like the NW algorithm, is used for the trace back phase. Along with matrix filling and recording back pointers for trace back, SW algorithm requires an additional step: keeping track of the maximum score of the entire matrix. The trace back phase itself however, is different from NW algorithm. Implementation may vary for how programmer wishes to keep track of the maximum. In our method, the matrix maximum score is checked within the fill matrix() (*see next section*) method when filling the scoring matrix. When a cell's value is obtained from the formula above, it is compared with the current maximum. If it is not greater, we move forward with the algorithm, if it is greater, we replace the current maximum score with the value obtained at the current matrix cell. For the trace back phase, instead of always starting from the bottom-most right corner, we begin from the highest score in the table, knowing the cell's indices, and work from there until we hit a cell with a score of zero. By doing so, we are not traversing the entire matrix, thus validating our method of local alignment. These differences between the two algorithms are crucial to remember as their performance will be illustrated in the *Results and Analysis* section.

Methodology and Experimentation and Database

Our implementation involved coding both the NW and SW algorithm in Java. NCBI, the National Center for Biotechnology Information, provides user with an expansive genetic sequence database containing all publicly available DNA sequences called GenBank **[7]**. Here, users can search for DNA sequences for various proteins and structures, and filter their result by sequence length, species, type of sequence (protein, nucleotide, etc.), and molecule type. Varying the sequence length was a primary factor in our testing phase as the time complexity was dependent on input sizes of the two sequences. Thus, random sequences of arbitrary species were used in the following approximate[3] lengths: 1000, 2500, 5000, 7500, 10000, 12500, 15000, 17500, 20000 base-pairs (or bp). The files were downloaded in a format called FASTA.


FASTA

Sus scrofa phosphorylase, glycogen, liver (PYGL), mRNA

NCBI Reference Sequence: NM_001123172.1

GenBank Graphics

```
>gi|178056752|ref|NM_001123172.1| Sus scrofa phosphorylase, glycogen, liver (PYGL), mRNA
ATGGCCAAGCCCCTGACAGACCAGGAGAAGCGGCGGCAGATCAGCATCCGCGGCATCGTGGGCGTGGAGA
ACGTGGCCGAGCTGAAGAAGGGCTTCAACCGACACCTGCATTTCACACTGGTCAAGGACCGTAATGTGGC
CACCCCTCGCGACTACTTCTTCGCGCTGGCGCACACAGTGCGCGACCACCTGGTGGGGCGCTGGATCCGC
ACGCAGCAGTACTACTACGAGAAGTGCCCCAAGAGGGTTTATTACCTCTCCTTGGAATTTTACATAGGCC
GAACATTGCAGAACACCATGATCAACCTTGGCCTGCAGAATGCTTGTGACGAGGCCATTTACCAGCTTGG
ATTGGATATGGAGGAGTTAGAAGAAATTGAAGAAGATGCTGGGCTTGGCAACGGTGGTCTTGGGAGGCTT
GCTGCCTGCTTCTTGGATTCCATGGCAACTCTGGGACTTGCAGCCTATGGATATGGAATCCGATATGAAT
ATGGAATCTTCAATCAGAAGATTCGAGATGGATGGCAGATAGAAGAGGCAGATGATTGGCTCAGGCATGG
AAACCCCTGGGAGAAGGCCCGCCCCGAGTTCATGCTGCCTGTGCACTTCTATGGAAGAGTGGAGCACACC
GAAGCTGGGACCAAGTGGGTCGACACCCAGGTGGTCCTGGCTCTGCCATATGACACCCCGGTGCCTGGGT
```

Figure 1.2: Sample FASTA format for a DNA sequence on GenBank

GenBank also provides its own GenBank format, but FASTA format worked well with the Java code. Since we had nine different input sizes, and needed two of each for aligning sequence A with sequence B, a total of 18 FASTA file were retrieved from the database. Our Java code carefully parsed the FASTA files as to only retrieve the sequence and input each of them into strings. The code creates an objects of the class taking in both of sequence string as inputs in order to generate the size of the trace back and scoring matrix.

Our primary method called fill matrix() contains the core of the algorithm, computing the value at each cell of the matrix of size M x N. The trace back phase and initialization phase are done elsewhere in the code, but do not play a vital role in affecting the overall time complexity as they are overshadowed by the fill matrix() method's higher-ordered time complexity. To test the running time and performance of each algorithm, we assigned a match/mismatch score and a gap penalty, and then recorded the max scores for SW algorithm and the score of the bottom-most right cell for NW algorithm. The numbers of comparisons were also counted in order to determine the pattern of running time as well as the elapsed time of the algorithms. It is

[3] Finding exact similar sizes from the database to compare is very difficult, and impractical. Thus, for an example, when testing for an input size of 10,000, one sequence being 10240 bp and the other being 10541 bp were "good enough" to compare in terms of similar sizes.

important to remember SW algorithm contains: an extra comparison per cell relative to NW algorithm, and also needs to keep track of the maximum score for the entire matrix. Thus, one can expect the running time (proportional to number of comparisons/elapsed time) to be *slightly* higher than the NW algorithm.

The number of comparisons, elapsed time, and score of alignment[4] were calculated for each trial for the respective input sequence size, and recorded into Microsoft Excel, where the data was plotted and studied to make comparisons between the two algorithms. The program itself was tested on a Linux machine.

Results and Analysis

In order to provide a complete overview of the performance and time efficiency of the algorithms, both the Needleman-Wunsch and Smith-Waterman algorithms need to be evaluated for each phase: initialization, matrix fill, and trace back. But before we get into that, it is important to bring back space complexity. Before doing anything with the matrix, the creation of the scoring and trace back matrix itself already has a space complexity of ~ *O(NM)*. This is because the core of dynamic programming relies on the algorithm and data structures to record and store information needed to invoke *memoization* (*see previous section on Recursion and Dynamic Programming*). Thus, evaluating each matrix cell and storing the data is pertinent to obtain the final alignment during the trace back phase.

For both algorithms, the initialization phase involved filling out the first row and the first column. Given there were N rows and M columns, and that we are only dealing with the first ones of each, the cost of this would just be ~ $O(N+M)$. During the NW algorithm, since we are performing arithmetics and comparisons in every cell of the matrix, and storing backpointers in the traceback matrix, the time complexity of the matrix fill() method will be ~ $O(NM)$. Finally, for the traceback phase, since we are only traversing the matrix from one corner towards the opposite corner, we traverse at most $max(N,M)$cells thus, the time complexity would of linear time $O(N)$ if $N > M$. The matrixfill() method in the SW algorithm is same as NW except that it involves an extra comparison against zero when finding maximum and also keeping track of the total maximum score. Nonetheless, the running time for this phase is still $O(NM)$. The traceback phase for SW algorithm only requires us to start at the indices of the maximum score and stop when the back pointers reach a zero-value cell. However, for approximation purposes, the time complexity is still of linear time.

Thus our total cost of the algorithm is **[8]**:

$$Total\ Cost = O(N+M) + O(NM) + O(N) \simeq O(NM)$$

[4] The score of the algorithms is not strongly related to the time efficiency. Regardless of good/poor alignment, the fillmatrix() method will always have a cost of O(NM) as every cell is guaranteed to be checked. Studying the scores was an additional observation made by the group in order to study any underlying patterns that may exist.

Thus we can expect a quadratic running time behavior from both algorithms when the sizes of both sequences are of similar lengths ($N \simeq M$). Shown below is a comparison plot of the calculated running time of both algorithms for increasing input size. The running time was evaluated by the number of comparisons and the time elapsed for the algorithm itself (in milliseconds).

Input Size (bp)	Time Elapsed Needleman-W (ms)	Time Elapsed for Smith-W (ms)	# of Comparisons Needleman-W	# of Comparisons Smith-W.
1000	42	39	3917965	8024000
2500	157	173	25157686	50160000
5000	567	553	92330431	200440224
7500	1265	1326	196995165	450360040
10000	2341	2147	339502612	800880224
12500	3866	3433	514522305	1250700080
15000	5558	5330	725343513	1800120000
17500	13793	16618	981340952	1843146960
20000	66091	193542	*	*

Table 1.1: Results of calculating time elapsed/number of comparisons for both algorithms

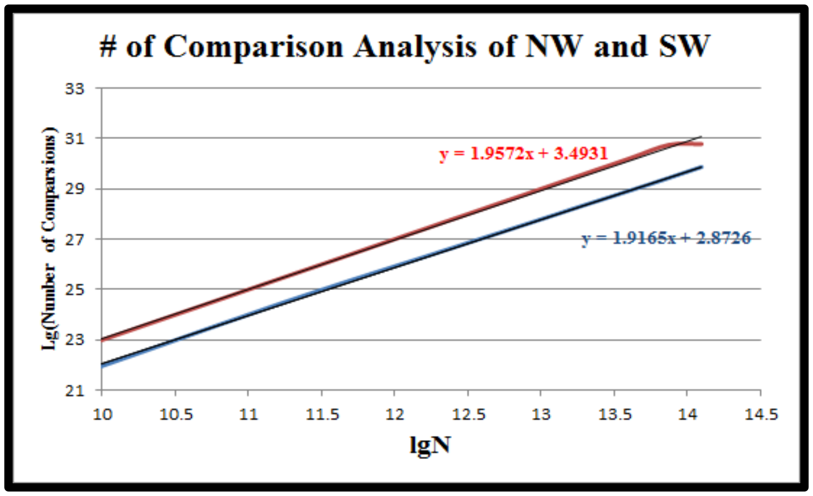


Figure 1.3: This figure compares the log-log plot of the number of comparisons made in the Smith-Waterman algorithm (RED) and Needleman-Wunsch(BLUE).

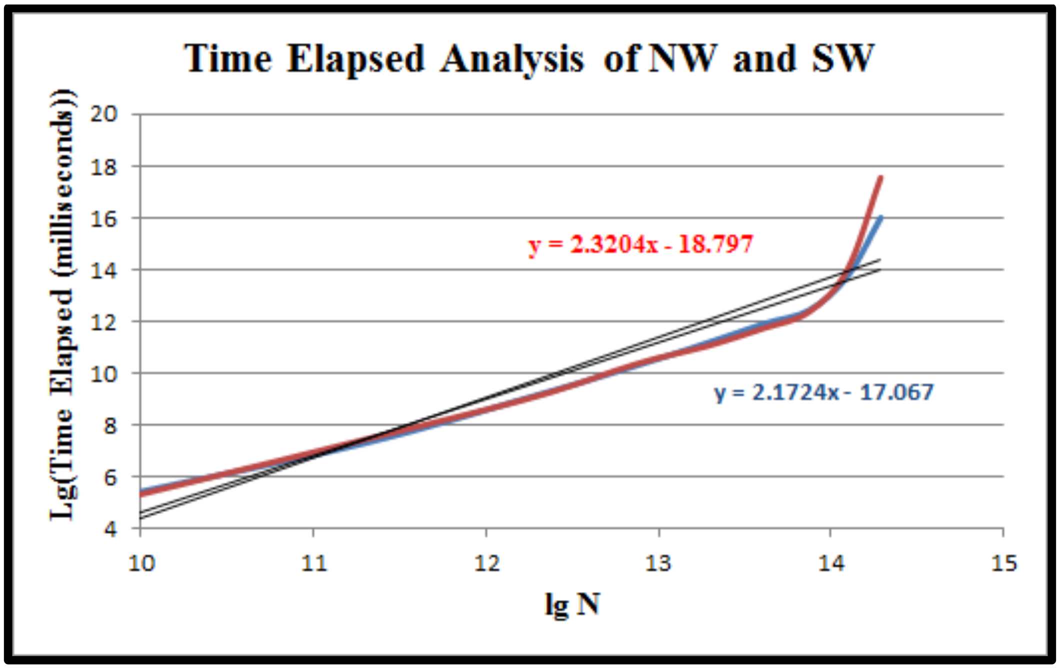


Figure 1.4: This figure compares the log-log plot of the elapsed time of the Smith-Waterman algorithm (RED) and Needleman-Wunsch(BLUE).

In Figure 1.2, the number of comparisons for increasing inputs were calculated for each algorithm and plotted as shown in the log-log plot ($Lg \sim \log base\ 2$). The red line indicates the data for the SW algorithm, whereas the blue represents the data for the NW algorithm. Using MS Excel to perform a best fit line aided us in determining the order of the running time by observing the slopes of the line. Remember, the lengths of both sequences for each data sets were nearly identical ($M \simeq N$).

Cost of Smith-Waterman/Needleman-Wunsch Algorithm By # of Comparison Analysis:

$$\textit{Smith-Waterman: } O(N^{1.957}) \simeq O(N^2) \quad \textit{Needleman-Wunsch: } O(N^{1.916}) \simeq O(N^2)$$

Performing the elapsed time analysis yielded similar log-log plots as shown in the next figure (Figure 1.3). Again the red line indicates the data for the SW algorithm and the blue line for the NW algorithm. Calculating time elapsed through built-in timers in the code can be tricky due to the inconsistent behaviour of the hardware running the algorithm. However, this doesn't affect the results greatly since we do indeed observe a convincing running time complexity.

Cost of Smith-Waterman/Needleman-Wunsch Algorithm By Elapsed Time Analysis:

$$\textit{Smith-Waterman: } O(N^{2.320}) \quad \textit{Needleman-Wunsch: } O(N^{2.172})$$

From the costs observed for both algorithms, we can confidently conclude an approximate quadratic[5] running time when both input sequences are nearly identical in length. When running the algorithms at an input size of roughly 20,000 base pairs, the algorithm took nearly 66 seconds and over 100 seconds for the NW and SW algorithms respectively. The slightly greater running time for the SW algorithm is most likely due to the extra comparison made against zero, as well as keeping track of the maximum score, as mentioned previously. Strangely enough, the time elapsed for input size greater than 12500 base pairs increased much faster than the previous input sizes. This could be due to a number of factors including how the hardware and RAM deals with large inputs such as 20,000 base pairs which create 20,000 by 20,000 matrices; thus, the $O(NM)$space complexity becomes a possible limiting factor. When testing on a Java-based IDE called Eclipse, input sizes above 15,000 base pairs would consistently give heap-space errors. However, testing on a Linux machine and terminal helped us evaluate sizes at 20,000 bp.

An Additional Observation

As stated in the experimentation section, the sequence alignment scores were recorded as well. There is no clear correlation between time complexity and scoring; however, we took it upon ourselves to look for any underlying patterns that may exists between sequence alignment scores with varying sizes for both algorithms. The scores recorded were divided by the maximum possible score in order to normalize the scoring. The maximum possible score assumes the ideal alignment: every nucleotide in the smaller sequence matches, thus if a match reward score is 5, then the maximum score for a sequence sized N base pairs against a sequence sized M base pairs would be $5N, if(N < M)$. Shown below is the score comparison of the SW and NW algorithm for increasing input size of just one sequence, A. The other sequence, B, was simply one DNA sequence string of 15,000 base-pairs, and remained constant in size.

[5] Do the algorithms always exhibit a quadratic time complexity? No; one cannot assign a time complexity of O(NM) to be always quadratic, cubic, linear etc. It will always depend on the size of both N AND M.

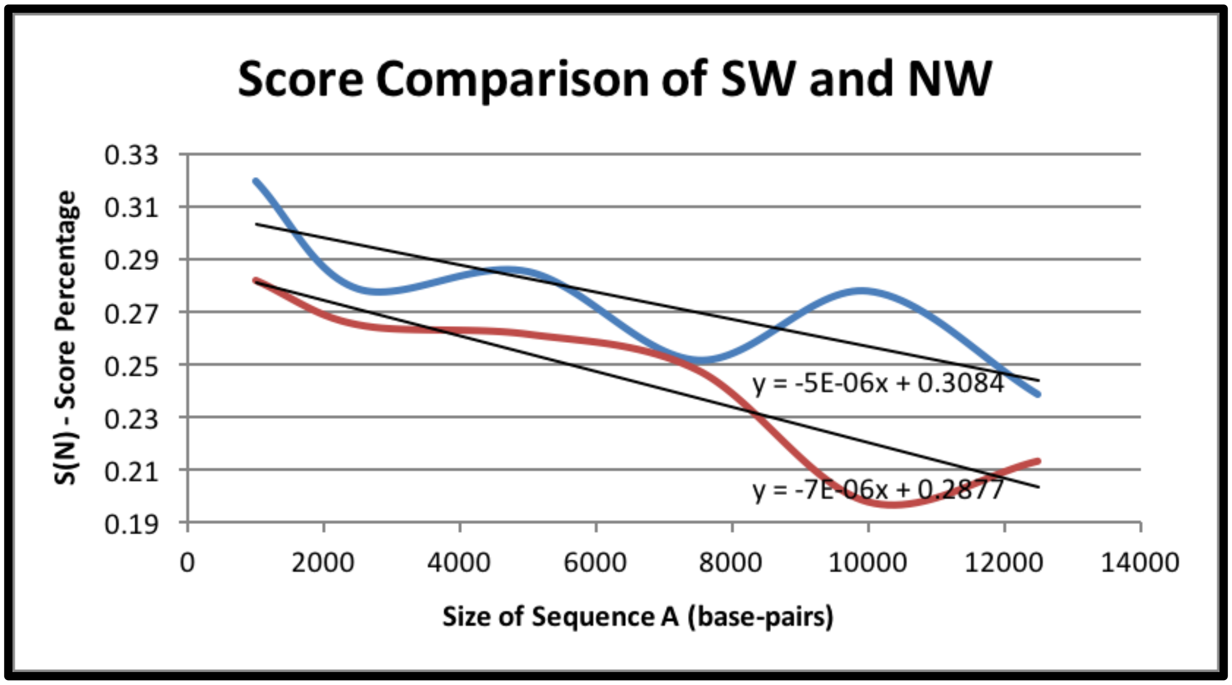


Figure 1.5: This figure compares score percentages of the Smith-Waterman algorithm (RED) and Needleman-Wunsch(BLUE) algorithm for varying the length of sequence A against a constant length of 15,000 bp for sequence B.

The trend shown above was confusing at first. The data sequence used for comparisons were chosen from GenBank based on sequence sizes for our experimentation and not on biological similarity or significance. This decrease in trend can possibly be explained by the fact that as we increase the input size of one sequence, more mismatches and gaps are being introduced. One may argue "But more matches are also being introduced? How come it is still decreasing". When creating the code for testing purposes, the scoring guideline was such that the gap penalty would receive a -4 score penalty and every mismatch resulted with a -3 score. Since the match reward score was set to +5, the combined penalty of gaps and mismatches, -7, outweighed the match reward score. This probably explains the general declining trend shown in the plot.

Discussion and Conclusion - Brief Introduction to BLAST

Observing that running the dynamic programming algorithms with input sizes of order ~ 10^4 caused them to run incredibly slow, a reader may ask the question: "Well, what about the DNA sequence in GenBank that are 11 million base-pairs in length; do we still use Smith-Waterman?" Indeed, one cannot begin to imagine how long that would take using the algorithms previously analyzed. The answer is a resounding 'NO'. The SW algorithm will provide us with the best local alignment possible, thereby resulting in the best possible score. However, time efficiency becomes a major issue, and it is obviously not a feasible solution. The algorithm has laid down the foundations for the local alignment in genetic research, however, it has become impractical because it is simply too slow **[8]**. Alternative methodologies and alignment packages have been made to resolve this issue. Currently, researchers use the tools such as BLAST to align genetic sequences **[1]**.

BLAST, Basic Local Alignment Search Tool, is a powerful family of algorithms that are used to achieve time efficient results with sequence alignment. This family of programs include: blastn, blastp, blastx, and other variations. For nucleotide sequence alignments, BLAST's *blastn* algorithm is used, and thus directly correlates to our discussion of the SW algorithm, as the current alternative solution to the poor time efficiency seen previously. It is important to note BLAST as a heuristic method: an algorithm which sacrifices optimality and accuracy for time efficiency. The sequence we want to find alignments for is called the query sequence, and the sequence we search against is called the subject sequence. A brief summary of blastn program is given below **[9]**.

Brief Summary of blastn Algorithm

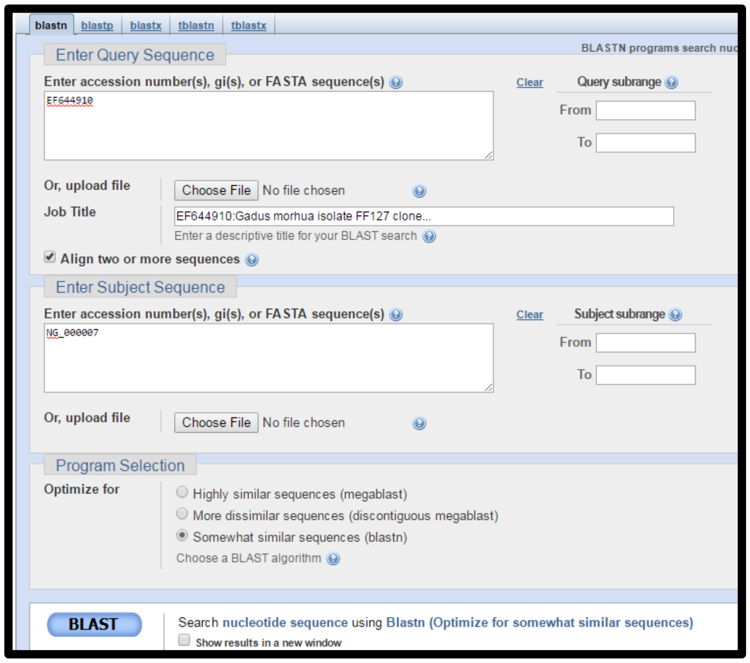
Break query sequence into words of sizes of roughly $6 < w < 11$ nucleotides, place them in a lookup table. Set a pairing threshold score T, helps discards irrelevant sequences for search hits. When a search hit is found, extend alignment from both sides until the score falls below $S = Max\ Score - X$. Discard any aligned segments below score S. Use the expectation value E to judge the alignment. The expectation value indicates the chance that the alignment between two sequences has occurred by chance. Thus, if two sequences show a clear biological correlation, then most likely, their alignment wasn't due to chance; hence we would expect a low expectation value.

$E(N, M, S) = KMNe^{-\lambda S}$: where N, M are sequence lengths and S is the score.

The factors $K$and λ are simply scaling factors depending on the substitution matrix used. We can clearly see that as the alignment score S increases, then the expectation value decreases.

Example of blastn: http://blast.ncbi.nlm.nih.gov/Blast.cgi?PAGE=Protein

Query Sequence: genes for the hemoglobin structure of Atlantic cod fish - *gadus morhua*
Subject Sequence: genes for the hemoglobin structure of humans - *homo sapiens*


blastn | blastp | blastx | tblastn | tblastx

BLASTN programs search nuc

Enter Query Sequence

Enter accession number(s), gi(s), or FASTA sequence(s): EF644910

Clear

Query subrange: From ___ To ___

Or, upload file: Choose File No file chosen

Job Title: EF644910:Gadus morhua isolate FF127 clone...

Enter a descriptive title for your BLAST search

[✓] Align two or more sequences

Enter Subject Sequence

Enter accession number(s), gi(s), or FASTA sequence(s): NG_000007

Clear

Subject subrange: From ___ To ___

Or, upload file: Choose File No file chosen

Program Selection

Optimize for:
- ○ Highly similar sequences (megablast)
- ○ More dissimilar sequences (discontiguous megablast)
- ● Somewhat similar sequences (blastn)

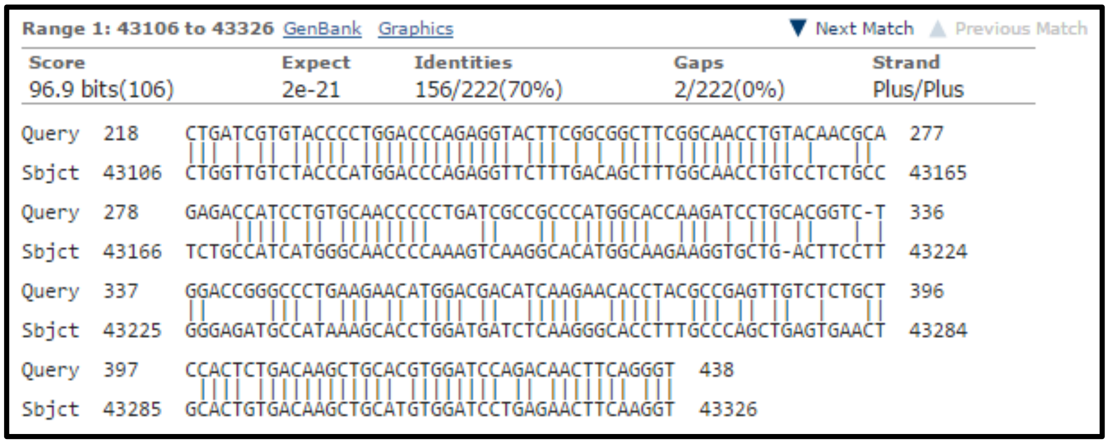
Choose a BLAST algorithm

BLAST

Search nucleotide sequence using Blastn (Optimize for somewhat similar sequences)

☐ Show results in a new window

Figure 1.6: The interface for the BLAST program running blastn for nucleotide alignment. The query and subject sequence accession number are pasted in the respective boxes.


```
Range 1: 43106 to 43326  GenBank  Graphics                    ▼ Next Match  ▲ Previous Match

Score              Expect    Identities       Gaps          Strand
96.9 bits(106)     2e-21     156/222(70%)     2/222(0%)     Plus/Plus

Query  218    CTGATCGTGTACCCCTGGACCCAGAGGTACTTCGGCGGCTTCGGCAACCTGTACAACGCA  277
              ||| | || ||||| ||||||||||||| ||| | | |||| |||||||||| |   ||
Sbjct  43106  CTGGTTGTCTACCCATGGACCCAGAGGTTCTTTGACAGCTTTGGCAACCTGTCCTCTGCC  43165

Query  278    GAGACCATCCTGTGCAACCCCCTGATCGCCGCCCATGGCACCAAGATCCTGCACGGTC-T  336
                  ||||| || ||||||||    ||   || |||||||  ||| | ||| ||   | |
Sbjct  43166  TCTGCCATCATGGGCAACCCCAAAGTCAAGGCACATGGCAAGAAGGTGCTG-ACTTCCTT  43224

Query  337    GGACCGGGCCCTGAAGAACATGGACGACATCAAGAACACCTACGCCGAGTTGTCTCTGCT  396
              ||     ||| | ||| || |||| ||  |||||  |||||  ||| || ||  |   ||
Sbjct  43225  GGGAGATGCCATAAAGCACCTGGATGATCTCAAGGGCACCTTTGCCCAGCTGAGTGAACT  43284

Query  397    CCACTCTGACAAGCTGCACGTGGATCCAGACAACTTCAGGGT  438
               |||| |||||||||||| |||||||| || ||||||| |||
Sbjct  43285  GCACTGTGACAAGCTGCATGTGGATCCTGAGAACTTCAAGGT  43326
```

Figure 1.7: One of the multiple alignment results from the sequence alignment search. There were 156/222 matches here with an expectation value of $2e^{-21}$.

As described by the blastn algorithm, many segments that have a low score are tossed away and do not return with the rest of the results. Thus BLAST's usefulness exists in the algorithms ability to overlook poor results for quicker results. The expectation value in the example above was very low. Since the expectation value indicates the probability that the sequence alignment occurred by chance according to a score S, we can clearly see that there are indeed similar genes involved in the hemoglobin structure of cod fish and humans. Starting from the Needleman-Wunsch algorithm, and then introducing local alignment with the Smith-Waterman algorithm, the foundations of sequence alignment were developed through the power of dynamic programming. However, with increasing sequence sizes being compared in bioinformatics, heuristic and powerful tools such as BLAST are needed.

References

[1] Altschul, Stephen F., et al. "Basic local alignment search tool." *Journal of molecular biology* 215.3 (1990): 403-410.

[2] Lemmen, Christian, and Thomas Lengauer. "Computational methods for the structural alignment of molecules." *Journal of Computer-Aided Molecular Design* 14.3 (2000): 215-232.

[3] Toth, Charles, and Richard Connelly. "A bioinformatics experience course." *Journal of Computing Sciences in Colleges* 21.6 (2006): 100-107.

[4] Du, Zhihua, and Feng Lin. "Improvement of the Needleman-Wunsch algorithm." *Rough Sets and Current Trends in Computing*. Springer Berlin Heidelberg, 2004.

[5] Mount, David W. "Comparison of the PAM and BLOSUM amino acid substitution matrices." *Cold Spring Harbor Protocols* 2008.6 (2008): pdb-ip59.

[6] Needleman, Saul B., and Christian D. Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins." *Journal of molecular biology* 48.3 (1970): 443-453.

[7] Benson, Dennis A., et al. "GenBank." *Nucleic acids research* 36.suppl 1 (2008): D25-D30.

[8] Chan, Alexander. "An Analysis of Pairwise Sequence Alignment Algorithm Complexities: Needleman-Wunsch, Smith-Waterman, FASTA, BLAST and Gapped BLAST." (2007).

[9] http://blast.ncbi.nlm.nih.gov/Blast.cgi