1. INTRODUCTION

Sequence alignment is an area in Bioinformatics using which the sequences of RNA, DNA and protein are arranged, in order to identify regions of similarity and their structural, functional or evolutionary relationships. Sequence alignment is of two types: Pair-wise Sequence Alignment and Multiple Sequence Alignment. Pair-wise Sequence Alignment is an alignment of two biological sequences whereas Multiple Sequence Alignment is an extension of Pair-wise Sequence Alignment where multiple biological sequences are aligned.

The fundamental building blocks of the cells of any organism are proteins, RNA and DNA. Proteins have a three dimensional structure required to perform a wide variety of chemical reactions necessary for life. A protein is constituted of 20 different types of amino acids. The linear sequence of amino acids arranged in a particular fashion is the reason why a protein assumes a three dimensional structure. The protein size is measured in terms of the number of amino acids that comprise it. On an average a protein is 340 amino acids in length.

Every protein that an organism produces is encoded in a piece of DNA called gene. The DNA has a one dimensional structure required to pass on critical storage information from one generation to another. RNA is another cellular molecule which is an intermediary between DNA and protein and provides some of the functionalities of DNA and protein.

A DNA sequence is a string of characters a_1, a_2, a_3, a_4…….a_x, each of which is drawn from the alphabet $a_i \in \sum = \{C, A, T, G\}$. The DNA sequences are "double stranded" meaning that each sequence a is paired with a complementary sequence a', with $a'_i \in \sum = \{C, A, T, G\}$ but where an A in a is paired with T in a', a T with A, a G with C and a C with G.

RNA is chemically similar to a DNA. RNA uses the base Uracil (U) instead of Thymine (T). U is chemically similar to T, and in particular is also complementary to A.

2. MOTIVATIONS FOR MULTIPLE SEQUENCE ALIGNMENT

Following are some of the many reasons why Multiple Sequence Alignment is an important field in the area of Bioinformatics:

1. To detect regions of variability or conservation in a family of proteins
2. To provide stronger evidence than pair-wise similarity for structural and functional inferences
3. To serve as the first step in phylogenetic reconstruction, in RNA secondary structure prediction, and in building profiles (probabilistic models) for protein families or DNA signals.

3. SEQUENCE PAIRWISE ALIGNMENT PROBLEM

When two (or more) sequences are being aligned, we come across two situations: one is when a character in one of the sequences is being matched with the same character at the corresponding position in the other sequence. This is nothing but a "match". When a character in one of the sequences is being matched with a different character (at the corresponding position) in the other sequence, this results in a "mismatch" and is called a point mutation. The other situation is when a character in one of the sequences is aligned/ matched with a "gap character" at the corresponding position in the other sequence. This again is a "mismatch" and can be interpreted as an "insertion" or a "deletion".

A string is a sequence of characters from some alphabet. Consider two strings ***acbcdb*** and ***cadcd***.

The similarity between these two strings can be witnessed by finding a good alignment between them. Following is one possible way of aligning the strings:

Sequence #1 ac--bcdb
Sequence #2 -cadc-d–

Corresponding to the first character "a" in Sequence #1, there is a deletion in Sequence #2. Similarly, corresponding to the third and fourth characters "a" and "d" in Sequence #2, there are deletions in Sequence #1. Hence, a deletion in one sequence corresponds to an insertion in the other sequence with which it is paired. Corresponding to the fifth character "b" in Sequence #1, there is a point mutation in Sequence #2 in the form of character "c".

3.1. Types of Pair-wise Alignments

3.1.1. Global Alignment

Global alignment assumes that the two sequences are basically similar over the entire length of one another. This alignment attempts to match them from end to end, even though parts of the alignment may not look convincing.

Given below is an example for the same:

```
LGPSTKDFGKISESREFDN
|      ||| |    |
LNQLERSFGKINMRLEDA
```

3.1.2. Local Alignment

Local alignment searches for segments of the two sequences that match well. There is no attempt to force the entire sequences into the alignment. Only the parts that appear to have a good similarity are aligned according some criterion. Consider the following sequences: LGPSTKDFGKISESREFDN and LNQLERSFGKINMRLEDA

A local alignment on the above sequences will have the following structure:

```
----------FGKI----------
          ||||
----------FGKI----------
```

3.1.3.Optimal Alignment

The best alignment, given a defined set of rules and parameter values for comparing different sequences, is called an Optimal Alignment.

Mathematically Optimal Pair-wise Alignment can be explained in the following way:
Given a pair of sequences A = $a_1a_2….a_m$ and B = $b_1b_2…b_n$ of length |A| = m and |B| = n from the finite alphabet, a pair-wise sequence alignment is obtained by inserting gap characters "-" into A and B. The aligned sequences A′ and B′ from the extended alphabet are of equal length such that |A′| = |B′|.

An alignment score provides a metric to assess the quality of an alignment and represents a measure of similarity between sequences. For pair-wise alignment, it is the sum of similarity values for each pair of aligned characters. Characters that match have a positive value while those that mismatch have a lower or negative value. Any character aligned with a gap also contributes a negative value to the alignment score. Let the function s: $\sum$ x $\sum$ ->Z determine the similarity of two characters and let *l* denote the length of an alignment. The pair-wise scoring function is then given by:

$$FPW\ (A', B') = \sum_{1 \le i \le l} s\ (a_i', b_i')$$

This metric as shown in the equation above is called the *sum of pairs scoring function.*

The goal is to find an optimal pair-wise alignment of A and B such that for all possible alignments, the score is maximal. Hence, optimality is always defined with respect to a particular score function.

4. Pair-wise Sequence Alignment – A naïve approach

A simple and naïve approach would be to try all possible alignments and choose the one which yields the maximum score value. A subsequence of a string S means a string of characters that need not be consecutive in S but retain the order in which they appear in S. For example, *acd* is a subsequence of *acbcdb*.

Assume that we are given two strings S and T and |S| = |T| = n. Also, suppose we have a scoring function σ(x, y), such that aligning spaces is not allowed, i.e., σ (-,-) <= 0.
The algorithm for a pair-wise alignment is given below:

```
for all i, 0 <= i <= n, do
        for all subsequences A of S with |A| = i do
```

```
            for all subsequences B of T with |B| = i do
                Form an alignment that matches A[k] with B[k], 1 <= k <= i,
                and matches all other characters with spaces;
                Determine the value of this alignment;
                Retain the alignment with maximum score value;
        end ;
    end ;
end;
```

RUNNING TIME ANALYSIS:

A string of length n has ${}^nC_i$ [1] subsequences of length i. Thus there are $({}^nC_i)^2$ pairs (A, B) of subsequences each of length i. Consider one such pair. Since |S| = n, only i of which are matched with characters in T, there will be (n-i) characters in S unmatched to characters in T. Thus, the alignment has length n + (n-i) = 2n – i. The score of each pair in the alignment must be added; therefore the total number of basic operations is at least:

$$\sum_{i=0}^{n} ({}^nC_i)^2 (2n - i) \geq n \sum_{i=0}^{n} ({}^nC_i)^2 \geq n\,({}^{2n}C_n) > 2^{2n}, \text{ for } n>3$$

The last inequality follows from Stirling's approximation. Hence, for n=20, the algorithm requires more than 2^{40} basic operations.

Thus, the above algorithm is too slow to be practical.

5. PAIRWISE SEQUENCE ALIGNMENTS USING DYNAMIC PROGRAMMING

Dynamic programming is a method for solving complex problems by breaking them down into simpler sub problems. Hence, dynamic programming is applicable to problems exhibiting the properties of overlapping sub problems which are only slightly smaller and have an optimal substructure. As seen above the naïve Sequence Alignment approach takes exponential running time (O (2^{2n})) which, doesn't seem to be a pragmatically feasible approach. The dynamic programming approach reduces the running time from exponential to polynomial which is far less time than the naive approach.

5.1. COMPUTING AN OPTIMAL ALIGNMENT BY DYNAMIC PROGRAMMING

5.1.1.Needleman – Wunsch Algorithm

The Needleman-Wunsch algorithm, which is an example of dynamic programming, was the first application of dynamic programming to biological sequence comparison. This algorithm performs a global alignment on two sequences. The algorithm was published in 1970 by Saul B. Needleman and Christian D. Wunsch.

[1] The notation ${}^nC_i$ denotes the number of combinations of *n* distinguishable objects taken *i* at a time.

Consider two strings S and T, with length of string S being |S| = n and length of string T being |T| = m. In order to compute an optimal alignment of S and T, we define V(i, j) as the value of an optimal alignment of the strings S[1], S[2], …….S[i] and T[1], T[2], ……T[j].

The value of an optimal alignment of S and T is V (n, m). The dynamic programming approach first solves the more general problems of computing all values V (i, j) with $0 \leq i \leq n$ and $0 \leq j \leq m$, in increasing order of i and j. Each of these values is computed by using the already computed values of V (i, j) (for smaller values of i and j) with the help of a recurrence relation. The algorithm memorizes the solutions of optimal sub-problems in an organized, tabular form (a dynamic programming matrix), so that each sub-problem is solved just once.
The basis for starting the Dynamic programming approach is to set i=0 and/or j=0. S[i] and T[j] correspond to the i^{th} character in string S and the j^{th} character in string T respectively. σ represents the scoring function.

BASIS:

$$V(0, 0) = 0$$
$$V(i, 0) = V(i-1, 0) + \sigma(S[i], -), \text{ for } i > 0$$
$$V(0, j) = V(0, j-1) + \sigma(-, T[j]), \text{ for } j > 0$$

According to the basis V (i, 0), if i characters of S are to be aligned with 0 characters of T, then they must all be matched with spaces. Similarly, for V (0, j), if j characters of T are to be aligned with 0 characters of S, then they must be matched with spaces.

RECURRENCE:

For i > 0 and j > 0,

$$V(i, j) = \max \begin{pmatrix} V(i-1, j-1) + \sigma(S[i], T[j]), \\ V(i-1, j) + \sigma(S[i], -), \\ V(i, j-1) + \sigma(-, T[j]) \end{pmatrix}$$

The above formula can be understood by considering an optimal alignment of the first i characters from S and the first j characters from T. In particular, consider the last aligned pair of characters in such an alignment. The last pair must be one of the following:

1. (S[i], T[j]), in which case the remaining alignment excluding this pair must be an optimal alignment of S[1]…..S[i-1] and T[1]…..T[j-1] (i.e must have the value V(i-1, j-1)), or
2. (S[i], -), in which case the remaining alignment excluding this pair must have the value V(i-1, j), or
3. (-, T[j]), in which case the remaining alignment excluding this pair must have the value V(i, j-1)

In Case 1, aligning S[i] and T[j] will result in a match or a mismatch which corresponds to a point mutation (as explained above). Case 2 corresponds to an insertion of a character at the i^{th} position in string S and a deletion in string T. According to Case 3, when a j^{th} character in string

T is being matched with a "gap character" in string S, it corresponds to an insertion in string T with respect to string S.

The optimal alignment chooses whichever among the above three possibilities has the greatest value.

5.1.1.1. Example:

Consider the two sequences: Sequence S = ***acbcdb*** and Sequence T = ***cadbd***. The dynamic programming algorithm will fill in the following values as indicated in the table below, by applying the basis and recurrence formulae.

Assumptions: In case of a match, σ (S_i, T_j) = +2 and in case of a mismatch, σ (S_i, T_j) = -1. Gap penalty for σ (S_i, -) or σ (-, T_j) is equal to -1.

The dynamic programming approach starts a pointer at the heads of each of the two strings, S and T, and then progressively move these pointers to the right, computing the best alignment under the three possible operations (as indicated in the recurrence relation). The result of moving these two pointers from 1 to n (or m) is an n x m matrix that contains the results of the sub problems.

Consider the entry in row 1 and column 2. V (1, 2) should be an entry corresponding to a "match". Hence the value corresponding to V (1, 2) will be **max (V (0, 1) + 2, V (0, 2) - 1, V (1, 1) - 1).**

Hence the value is max (-1 + 2, -2 - 1, -1 - 1) = 1.

Consider the entry in row 4 and column 5. V (4, 5) should be an entry corresponding to a "mismatch". Hence the value corresponding to V (4, 5) will be **max (V (3, 4) - 1, V (3, 5) - 1, V (4, 4) - 1).**

Hence the value is max (2 - 1, 1 - 1, 1 - 1) = 1.

The value of the optimal alignment is V (n, m) and so can be read from the entry in the last row and last column. Thus, there is an alignment of ***acbcdb*** and ***cadbd*** that has value 2.

j		0	1	2	3	4	5
i			**c**	**a**	**d**	**b**	**d**
0		0	-1	-2	-3	-4	-5
1	**a**	-1	-1	1	0	-1	-2
2	**c**	-2	1	0	0	-1	-2
3	**b**	-3	0	0	-1	2	1
4	**c**	-4	-1	-1	-1	1	1
5	**d**	-5	-2	-2	1	0	3
6	**b**	-6	-3	-3	0	3	2

Pseudo-code for computing the above matrix using Needleman Wunsch algorithm:

```
Needleman_Wunsch()
{
        for i=0 to length(S)      //Initialize first column according to the basis function
                V (i,0) ← d*i     //d=gap penalty
        for j=0 to length(T)      //Initialize first row according to the basis function
                V (0,j) ← d*j
        for i=1 to length(S)      //Traverse the rows from 1 to length(S)
        for j=1 to length(T)      //For each row, traverse the columns from 1 to length(T)
        {
                Match ← V (i-1,j-1) + $\sigma(S_i, T_j)$        //Calculate Match score; σ=scoring function
                Delete ← V (i-1, j) + d                          //Calculate Delete score
                Insert ← V (i, j-1) + d                          //Calculate Insert score
                V (i, j) ← max(Match, Insert, Delete) //Take the max of Match, Insert or Delete
        }
}
```

RECOVERING THE ALIGNMENTS:

After having obtained the optimal alignment value, the next step is to obtain the optimal alignment. This can be done by retracing the Dynamic programming steps back from V (n, m) entry, determining which preceding entries were responsible for the current one. For instance, in the table below, the (4, 2) entry could have followed from either the (3, 1) or (3, 2) entry; this is denoted by the two arrows pointing to those entries. We can then follow any of these paths from (n, m) to (0, 0), tracing an optimal alignment. Hence, an optimal alignment value denoted by (n, m) can have more than 1 possible optimal alignment.

i \ j		0	1	2	3	4	5
			c	**a**	**d**	**b**	**d**
0		0	-1 ←	-2	-3	-4	-5
1	**a**	-1 ↑	-1	1 ↖	0	-1	-2
2	**c**	-2	1 ↖	0	0 ↖	-1	-2
3	**b**	-3	0 ↑	↖ 0	-1	2 ↖	1
4	**c**	-4	-1	↖↑ -1	-1	1 ↑	1
5	**d**	-5	-2	-2	1 ↖	0	↖ 3
6	**b**	-6	-3	-3	0	3 ↖	←↑ 2

Pseudo-code for computing the optimal alignments using the Needleman Wunsch algorithm:

```
Retrace ()
{
    AlignmentS ← ""        //Set AlignmentS and AlignmentT to NULL, initially
    AlignmentT ← ""
    i ← length(S)
    j ← length(T)
    While (i > 0 and j > 0) //Loop until the end of one of the strings is encountered
    {
        Score ← V (i,j)
        ScoreDiag ← V (i - 1, j - 1)
        ScoreUp ← V (i, j - 1)
        ScoreLeft ← V (i - 1, j)

        //If Score at i,j is obtained from the Score at the diagonal element
        If (Score == ScoreDiag + $\sigma$ ($S_i$, $T_j$))
        {
            AlignmentS ← $S_i$ + AlignmentS        //Append $S_i$ to AlignmentS
            AlignmentT ← $T_j$ + AlignmentT        //Append $T_j$ to AlignmentT
            i ← i - 1
             j ← j - 1
        }

        //If Score at i,j is obtained from the Score at the Left element
        else if (Score == ScoreLeft + d)
        {
            AlignmentS ← $S_i$ + AlignmentS        //Append $S_i$ to AlignmentS
            AlignmentT ← "-" + AlignmentT        //Append '-' to AlignmentT
            i ← i - 1
        }

        //If Score at i,j is obtained from the Score at the Upper element
        otherwise (Score == ScoreUp + d)
        {
            AlignmentS ← "-" + AlignmentS        //Append '-' to AlignmentS
            AlignmentT ← $T_j$ + AlignmentT        //Append $T_j$ to AlignmentT
            j ← j - 1
        }
    }
    while (i > 0)    //If end of T is encountered
```

```
        {
            AlignmentS ← $S_i$ + AlignmentS        //Append remaining $S_i$ to AlignmentS
            AlignmentT ← "-" + AlignmentT        //Append '-' to AlignmentT
            i ← i - 1
        }
        while (j > 0)    //If end of S is encountered
        {
            AlignmentS ← "-" + AlignmentS    //Append '-'  to AlignmentS
            AlignmentT← $T_j$ + AlignmentT     //Append remaining $T_j$ to AlignmentT
            j ← j - 1
        }
}
```

For the example shown above, the optimal alignments corresponding to the three paths (and corresponding to the optimal value (n, m) = 2 are as follows:

a c b c d b - - c - a d b d	,	a c b c d b - - c a - d b d	and	- a c b c d b c a d b - d -

Each of the above sequences has three matches, one mismatch and three spaces, for a value of

3*(+2) + 4 *(-1) = 2, which is the optimal alignment value.

RUNNING TIME ANALYSIS:

The dynamic programming based Needleman-Wunsch algorithm requires an (n+1)*(m+1) table to be computed. Hence the storage complexity of the algorithm is O(nm) and the computational complexity of the algorithm is also, at most c * (n+1) * (m+1) = O(nm). (Where c is a constant that represents the number of atomic operations to compute one element)
Reconstructing a single alignment can be done in O (n + m) time.

6. Multiple Sequence Alignment

The previous section discussed the problem of determining the similarity between two strings. This section deals with the problem of determining the similarity among multiple strings. Given three strings – ***VSNS***,***SNA*** and ***AS***, an optimal alignment for them would be:

```
VSN-S
-SNA-
---AS
```

The problem is defined more precisely below:

Given strings S_1, S_2, … ,S_k a multiple global alignment maps them to strings S_1', S_2', …, S_k' that may contain spaces, where

$|S_1|' = |S_2|' = \ldots = |S_k|'$, and the removal of spaces from S_i' leaves Si, for $1 \le i \le k$.

To determine the quality of a MSA, a more complex scoring function than the one for pair-wise alignment is needed. Various assumptions about the relationship between multiple sequences lead to several possible scoring methods. The weighted sum-of-pairs (WSP) method is popular among MSA programs. It assumes that sequences are related by an evolutionary tree and that sequence weights are derived from this tree. The WSP method calculates a total score from the weighted pair-wise score of all sequences. Let F_{WSP} : A -> Z be a WSP scoring function for an MSA **A** such that

$$F_{WSP}(A) = \sum_{1 \le i < j \le n} w_{i,j} \sum_{1 \le l \le k} s(a_i[l], a_j[l])$$

where *n* is the number of sequences, *k* is the length of aligned sequences, $w_{i,j}$ is the weight given to a pair of sequences, and the function s : ∑ x ∑ -> Z determines the similarity of the symbol $a_i[l]$ with $a_j[l]$. The algorithm for Multiple Sequence Alignment maximizes the score and thereby produces an optimal alignment.

6.1. Optimal Multiple Sequence Alignment using Dynamic Programming

Given k strings of length n, we can generalize the dynamic programming algorithm used for pair-wise alignment, for aligning k strings. Instead of a 2-dimensional table, the algorithm now fills a k-dimensional table. This table has dimensions:

$$(n+1) \text{ x } (n+1) \text{ x } \ldots \text{ x } (n+1) \text{ (k times)},$$

that is, $(n+1)^k$ entries. Each entry depends on $2^k - 1$ adjacent entries (excluding the empty subset) and represents the alignment score for k subsequences. The alignment score at a given position is calculated by the following recurrence relation:

For $i_1 > 0, i_2 > 0, \ldots, i_k > 0$

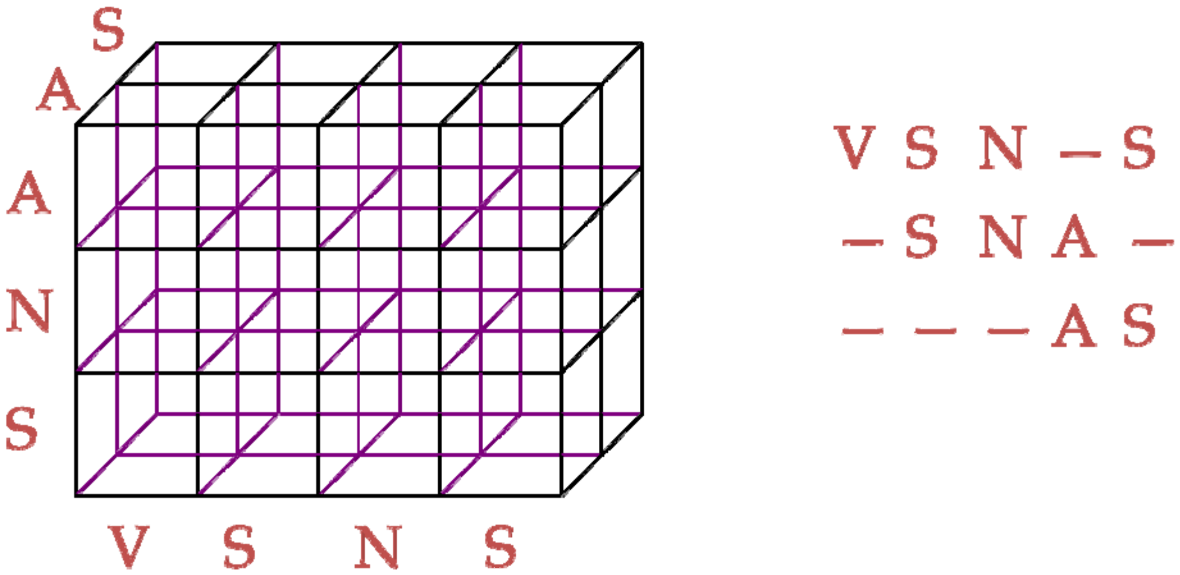
$$\begin{aligned} V(i_1, i_2, \ldots, i_k) = \max(&V(i_1 - 1, i_2 - 1, \ldots, i_k\text{-}1) + \sigma(S_1[i_1], S_2[i_2], \ldots, S_k[i_k]), \\ &V(i_1, i_2 - 1, \ldots, i_k\text{-}1) + \sigma(\text{-}, S_2[i_2\text{-}1], \ldots, S_k[i_k\text{-}1]), \\ &V(i_1\text{-}1, i_2, \ldots, i_k\text{-}1) + \sigma(S_1[i_1 - 1], \text{-}, \ldots, S_k[i_k\text{-}1]), \\ &\quad . \\ &. \\ &. \\ &V(i_1, i_2, \ldots, i_k\text{-}1) + \sigma(\text{-}, \text{-}, \ldots, S_k[i_k\text{-}1]) \\ &. \\ &. \\ &.\ ) \end{aligned}$$

RUNNING TIME ANALYSIS:

Each of the $(n+1)^K$ entries can be calculated in time proportional to $2^{k,}$ therefore the running time of the algorithm is $O((2n)^k)$. If n ≈ 350 (as is typical for the length of proteins), the algorithm would be practical for very small values of k, say 3 or 5. However typical protein families have hundreds of members, so the running time of the algorithm grows very quickly for such great values of k. Hence this algorithm is not practical for aligning large number of sequences. To find the global optimum for k sequences this way has been shown to be an NP-complete problem. Since time complexity of the dynamic programming approach is exponential in the number of sequences, heuristic methods are usually used.

Dynamic programming methods are now used only when an extremely high quality alignment of a small number of sequences is needed, and as a benchmarking standard in evaluating new or refined heuristic techniques.

The figure below shows a multiple sequence alignment using three sequences. The dynamic programming array looks like a cube. Visually it is a bit more confusing to look at and the complexity increases to $O(n^3)$.



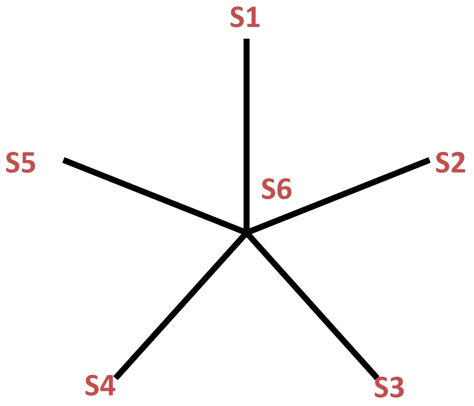
In the next section, we will look at an approximation approach to find a solution to the problem of multiple sequence alignment

7. An approximation approach to Multiple Sequence Alignment

In this section, we present an approximation algorithm for calculating the optimal multiple alignment using the Sum-of-pairs metric. Given a multiple alignment M, let $d(S_i, S_j)$ be the score of the pair-wise alignment it induces on S_i, S_j . Our target function is the Sum of Pairs value of M which is $d(M) = \sum_{i<j} d(S_i, S_j)$ (sum over all pairs of the score of the induced alignment).

We define the center star, T_c to be a star tree of k nodes, with the center node labeled S_c and with each of the k - 1 remaining nodes labeled by a distinct sequence in $S – \{S_c\}$. The multiple alignment M_c of S is the multiple alignment induced by the center star, i.e., for each $v \neq c$, the alignment M_c induces an optimal pair-wise alignment between S_c and S_v.

The figure below shows a generic center star for six strings, where the center string S_c is S_6.



The Center Star Algorithm:

1. Find $S_t \in S$ minimizing $\sum_{i\neq t} D(S_i, S_t)$ and let $M = \{S_t\}$. (Use the dynamic programming algorithm for pair-wise sequence alignment).

2. Add the sequences in $S - \{S_t\}$ to M one by one so that the alignment of every newly added sequence with S_t is optimal. Add spaces, when needed, to all pre-aligned sequences.

RUNNING TIME ANALYSIS:

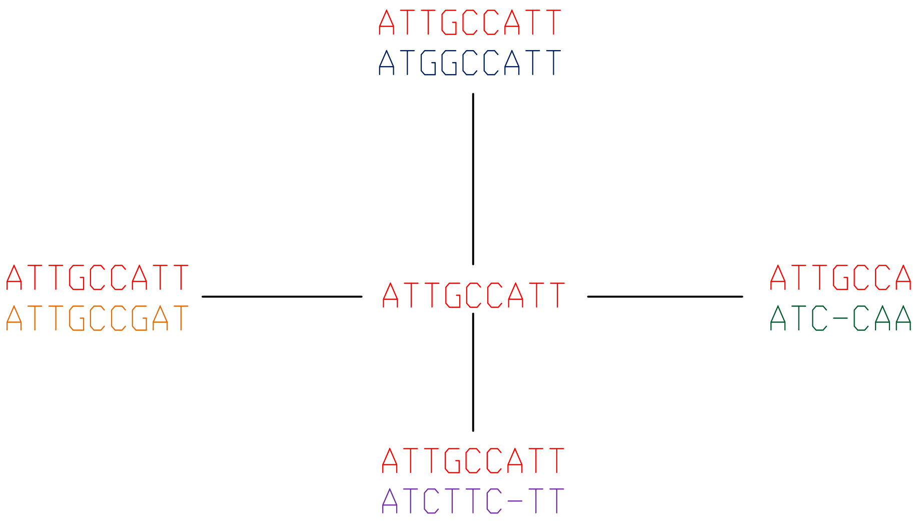
1. ${}^{k}C_{2}O(n^2)$ for step 1.
2. $\sum_{i=1}^{k-1} O((i.n).n) = O(k^2.n^2)$ for step 2 (Since the worst-case length of S'_c after the addition of i strings is (i + 1).n)

EXAMPLE:

Given the following sequences:

ATTGCCATT
ATGGCCATT
ATCCAATTTT
ATCTTCTT
ATTGCCGATT

1. The first step would be to choose the center star and align it with all other sequences in the input set. Assume that **ATTGCCATT** is chosen as the center star by the algorithm for this example. The following figure shows the alignment of the "center star" with all the other sequences in the input set.



2. The next step would be to merge the pair-wise alignments obtained in step 1. The resultant aggregate multiple alignments would be returned.

```
ATTGCCATT--
ATGGCCATT--
ATC-CAATTTT
```

```
ATTGCCATT--
ATGGCCATT--
ATC-CAATTTT
ATCTTC-TT--
```

```
ATTGCC-ATT--
ATGGCC-ATT--
ATC-CA-ATTTT
ATCTTC--TT--
ATTGCCGATT--
```

8. Other Approaches

One approach to multiple sequence alignments uses a heuristic search known as progressive technique (also known as the hierarchical or tree method), that builds up a final Multiple Sequence Alignment by combining pair-wise alignments beginning with the most similar pair and progressing to the most distantly related. All progressive alignment methods require two stages: a first stage in which the relationships between the sequences are represented as a tree, called a *guide tree*, and a second step in which the MSA is built by adding the sequences sequentially to the growing Multiple Sequence Alignment according to the guide tree.

A set of methods to produce Multiple Sequence Alignments while reducing the errors inherent in progressive methods are classified as "iterative" because they work similarly to progressive methods but repeatedly realign the initial sequences as well as adding new sequences to the growing Multiple Sequence Alignment.

References

1. Eddy, S. R. 2004. What is Dynamic Programming? *Nature Biotechnology* 22, 909–910. DOI=10.1038/nbt0704-909
2. Do, C. B. and Katoh, K. 2008. Protein Multiple Sequence Alignment. In Thompson, J.D. et al. *Methods in Molecular Biology: Functional Proteomics: Methods and Protocols 484,* 379-413. DOI: 10.1007/978-1-59745-398-1
3. Dynamic Programming Algorithms. Retrieved March 24, 2012, from http://bix.ucsd.edu/bioalgorithms/book/excerpt-ch6.pdf
4. Araki, S. et al. (N.D.). Application of Parallelized DP and A* Algorithm to Multiple Sequence Alignment. Retrieved March 24, 2012 from http://www.mtl.t.u-tokyo.ac.jp/~goshima/shiho.pdf.
5. Jean-Michel Richer, Vincent Derrien, and Jin-Kao Hao. 2007. A new dynamic programming algorithm for multiple sequence alignment. In *Proceedings of the 1st international conference on Combinatorial optimization and applications* (COCOA'07), Andreas Dress, Yinfeng Xu, and Binhai Zhu (Eds.). Springer-Verlag, Berlin, Heidelberg, 52-61.
6. Lloyd, G. S. (2010). Parallel Multiple Sequence Alignment: An Overview. Retrieved March 24, 2012 from http://dna.cs.byu.edu/msa/overview.pdf
7. Choudhury, R. (N.D.) Application of Evolutionary Algorithm for Multiple Sequence Alignment. Retrieved March 24, 2012 from http://biochem218.stanford.edu/Projects%202003/choudhury.pdf
8. Multiple Sequence Alignment. Retrieved March 24, 2012 from http://www.incogen.com/bioinfo_tutorials/Bioinfo-Lecture_4-multiple-sequence-align.html
9. Sheneman, L. and Foster, J. A. (N.D.). Eliminating Dynamic Programming Bias in Multiple Sequence Alignment Algorithms. Retrieved March 24, 2012 from http://people.ibest.uidaho.edu/~foster/Papers/8.pdf
10. Dewey, C. 2011. Multiple Sequence Alignment. Lecture Notes. Retrieved March 24, 2012 from http://www.biostat.wisc.edu/bmi576/lectures/multiple-alignment.pdf
11. Multiple Sequence Alignment. Lecture Notes. Retrieved March 24, 2012 from http://www.bioinformatics.wsu.edu/bioinfo_course/notes/lecture7.pdf
12. Introduction to Dynamic programming. Lecture Notes. Retrieved March 24, 2012 from http://www.webmath.iitkgp.ernet.in/breadth/ma23011/lectnotes07/Dynamic_Programming-4.pdf

13. Agarwal, P. K. (2003). Multiple Sequence Alignment. Lecture Notes. Retrieved March 24, 2012 from http://www.cs.duke.edu/courses/fall03/cps260/notes/lecture15.pdf
14. Reiners, P. D. (N.D.). Dynamic programming and sequence alignment. Retrieved March 24, 2012 from http://www.ibm.com/developerworks/java/library/j-seqalign/index.html
15. Tompa, M. (2000). Lecture Notes on Biological Sequence Analysis. Technical Report. University of Washington. Retrieved March 24, 2012 from http://www.cs.washington.edu/education/courses/cse527/00wi/lectures/roottr.pdf
16. Hochreiter, S. (2008). Bioinformatics I - Sequence Analysis and Phylogenetics**.** Lecture Notes. Retrieved March 24, 2012 from http://www.master-bioinformatik.at/curriculum/BioInf_I_Notes.pdf
17. Sequence Alignment. 2012. Retrieved March 24, 2012 from http://en.wikipedia.org/wiki/Sequence_alignment
18. Multiple Sequence Alignment. Retrieved March 24, 2012 from http://www.google.com/url?sa=t&rct=j&q=&esrc=s&frm=1&source=web&cd=1&sqi=2&ved=0CCwQFjAA&url=http%3A%2F%2Fwww.cs.iastate.edu%2F~cs544%2FLectures%2FMultiple_Sequence_Alignment.ppt&ei=4M17T-DnHqPi2QXLl4XJDA&usg=AFQjCNGAP2eeHhkQo_FFXBHCvRM9Wzvv2w&sig2=0axRhkVFGY0Wc7ge6EY62A