Chapter 1

Sequence Alignment

The goal of pairwise sequence alignment is to establish a correspondence between the elements in a pair of sequences that share a common property, such as common ancestry or a common structural or functional role. In computational biology, the sequences under consideration are typically nucleic acid or amino acid polymers. We will consider three variants of the pairwise sequence alignment problem: global alignment, semi-global alignment, and local alignment.

Global alignment is used in cases where we have reason to believe that the sequences are related along their entire length. If, for example, sequences s_1 and s_2 are two independent sequencing runs of the same PCR product, then they should differ only at those positions where there are sequencing errors. In order to find those sequencing errors, we align all of sequence s_1 with all of sequence s_2. Other applications of global alignment include finding mutations in closely related gene or protein sequences and identification of single nucleotide polymorphisms (SNPs).

Semi-global alignment is a variant of global alignment that allows for gaps at the beginning and/or the end of one of the sequences. Semi-global alignment is used in situations where we believe that s_1 and s_2 are related along the entire length of the region where they overlap. For example, if s_1 is the open reading frame of a eukaryotic gene with a single exon and s_2 is the mRNA transcript produced when s_1 is expressed, every base in s_1 should correspond to some base in s_2. Semi-global alignment "jumps" over the $5'$ untranslated region in s_2 without exacting a penalty, but forces an alignment along the entire length of s_1.

In contrast, local alignment addresses cases where we only expect to find isolated regions of similarity. One example is alignment of genomic DNA upstream from two co-expressed genes to find conserved regions that may correspond to transcription factor binding sites. Another application is identification of conserved domains[1] in two amino acid sequences

[1] A domain is a peptide sequence that encodes a protein module that will fold into its characteristic

that encode proteins that share one or more domains, but are otherwise unrelated.

Prior to introducing algorithms for these pairwise alignment problems, we introduce some notation in Box 1.

1.1 Global pairwise alignment

Now that we have some notation to work with, we will introduce a formal definition of a global alignment. Next we will consider how to assign a numerical score to an alignment. The score represents an assessment of the quality of the alignment. Finally, we will introduce an efficient algorithm to find the alignment that is optimal with respect to a scoring function.

Let $\Sigma' = \Sigma \cup \{\_\}$ be the alphabet expanded to include a character to represent gaps. Given sequence $s_1 \in \Sigma^*$ of length n_1 and sequence $s_2 \in \Sigma^*$ of length n_2, $\alpha^\kappa(s_1, s_2) = \{s^\kappa, s_2^\kappa\}$ is a global alignment of s_1 and s_2 if and only if

- $s_1^\kappa, s_2^\kappa \in (\Sigma')^*$,
- $|s_1^\kappa| = |s_2^\kappa| = l^\kappa$, where $\max(n_1, n_2) \leq l^\kappa \leq n_1 + n_2$,
- s_1 is the subsequence obtained by removing '_' from s_1^κ and s_2 is the subsequence obtained by removing '_' from s_2^κ,
- there is no value of i for which $s_1^\kappa[i] = s_2^\kappa[i] =$ '_'.

There are many alignments of s_1 and s_2. The superscript κ is an index to designate a specific alignment. In situations where only one alignment is under consideration or there is no ambiguity, we use the simpler notation $\alpha(s_1, s_2) = \{s_1', s_2'\}$, where the length of $\alpha(s_1, s_2)$ is simply denoted l.

Our goal is to find the global alignment that best captures the relationship between s_1 and s_2. Which alignment best reflects the relationship between s_1 and s_2 is fundamentally a biological question. From a practical perspective, we use a mathematical approach: We introduce an objective criterion that provides a measure of the quality of an alignment and then seek the alignment, $\alpha^*(s_1, s_2)$, that optimizes that criterion. There may be more than one.

1.1.1 Scoring an alignment

Given sequences s_1 and s_2 and an alignment $\alpha^\kappa(s_1, s_2) = \{s_1^\kappa, s_2^\kappa\}$, it is convenient to assign a score to alignment α^κ that quantifies how well α^κ captures the relationship between s_1 and s_2. This score may be a minimization or a maximization criterion.

---

shape independent of the surrounding amino acid context and that is found in many different proteins.

Box 1: Notation for pairwise alignment

Alphabet:

An *alphabet*, denoted by Σ, is a finite, unordered set of symbols; e.g.,

DNA: $\Sigma_D = \{A, C, G, T\}$
RNA: $\Sigma_R = \{A, C, G, U\}$
Amino acids: $\Sigma_{AA} = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$

Sequences or Strings:

A *sequence* or *string*, s, is a finite succession of the symbols in Σ.

Σ^* denotes the set of all sequences over alphabet Σ, including the empty sequence, $\varnothing$. For example, $\Sigma_R^* = \{\varnothing, A, C, G, U, AA, AC, AG, AU, CA, CC, CG, CU, \ldots\}$.

Given a sequence s of length n, we use $s[1]s[2]\cdots s[n]$ to denote the symbols in s.

Subsequences:

A *subsequence* of s is any sequence obtained by removing zero or more symbols from s. The sequences `CATA` and `CTG` are subsequences of `CATTAG`. `AATTCG` is not.

A *proper subsequence* is a subsequence obtained by removing one or more symbols from s.

Substrings:

A *substring* of s is a subsequence of s consisting of consecutive symbols in s. Given a sequence, s, of length n, the substring that begins with $s[i]$ and ends with $s[j]$ is denoted $s[i \ldots j]$, $1 \leq i \leq j \leq n$. The sequence `CAT` is a substring of `CATTAG`. `CATA` is not.

A *prefix* of s is denoted $s[1 \ldots j]$, $j \leq n$.

A *suffix* of s is denoted $s[i \ldots n]$, $1 \leq i$.

Distance scoring: An alignment can be scored using a distance-based metric. This is a minimization criterion: a lower distance indicates a better alignment. We define the distance score of an alignment $\alpha^\kappa(s_1, s_2) = \{s_1^\kappa, s_2^\kappa\}$ to be

$$\begin{aligned} D\big(\alpha^\kappa(s_1, s_2)\big) &= D(s_1^\kappa, s_2^\kappa) \\ &= \sum_{i=1}^{l^\kappa} d(s_1^\kappa[i], s_2^\kappa[i]), \end{aligned} \tag{1.1}$$

where $d(x, y)$ is the distance between a pair of symbols x and y in Σ' and l^κ is the length of the alignment. The optimal alignment, denoted α^*, is the alignment that minimizes the distance between s and t:

$$\alpha^*(s_1, s_2) = \operatorname*{argmin}_\kappa D\big(\alpha^\kappa(s_1, s_2)\big).$$

The function specifying the distance between pairs of symbols must satisfy the following properties, for all x, y, and z in Σ':

1. $d(x, x) = 0$
2. $d(x, y) > 0$
3. $d(x, y) = d(y, x)$
4. $d(x, y) \leq d(x, z) + d(z, y)$ (triangle inequality)

Several properties of the distance scoring function are worth noting. First, $D(s_1^\kappa, s_2^\kappa)$ is a metric; that is, it satisfies the triangle inequality. This means that the penalty for replacing x with y is never improved by first replacing x with z and they replacing z with y. When $z =$ '_', this says that deleting x and then inserting y is never an improvement on a direct substitution of x with y. One consequence of the triangle inequality is that the cost of a substitution can never be greater than twice the cost of an indel. Intuitively, this makes sense: Alignments obtained by minimizing a function where one substitution costs more than two indels would contain no substitutions. With such a function, an alignment in which x is aligned with y could always be replaced by a lower cost alignment in which x and y are both aligned with gaps.

Second, the symmetric property, $d(x, y) = d(y, x)$, implies that there is no directionality in the scoring system. This is because, given a column with symbols x and y in a pairwise alignment, we have no way of knowing whether the ancestral symbol was an x that was later replaced by a y, or vice versa. It is also possible that the ancestor was neither x nor y and some combination of substitutions gave rise to x in one sequence and to y in the other. Similarly, when a symbol, x, in sequence s_1 is aligned with a gap in sequence s_2 (or vice versa), there is no way to know whether x was inserted in s_1 or deleted from s_2. For this reason, gaps are also called "indels."

If $d(x, y) = 1$ and $d(x, \_) = 1$, $\forall x, y$, then $D\big(\alpha^*(s_1, s_2)\big)$ corresponds to the minimum number of operations required to transform s_1 into s_2, where the operations are substitution, insertion, and deletion. This is called the *edit distance*. If $d(x, y) > 1$ or $d(x, \_) > 1$ or both, then $D(\alpha^*(s_1, s_2))$ is called the *weighted edit distance*.

Similarity scoring: Alignments can also be scored with similarity measures. These are maximization criteria: a higher score indicates a better alignment. The similarity score of $\alpha^\kappa(s_1, s_2) = \{s_1^\kappa, s_2^\kappa\}$ is

$$S(\alpha_k(s_1, s_2)) = \sum_{i=1}^{l^\kappa} p(s_1^\kappa[i], s_2^\kappa[i]), \tag{1.2}$$

where $p(x, y)$ is a score that reflects the similarity of x and y and $p(x, \_)$ is the gap score. The optimal alignment is the alignment that maximizes the similarity between s_1 and s_2:

$$\alpha^*(s_1, s_2) = \underset{\kappa}{\operatorname{argmax}}\, S\big(\alpha^\kappa(s_1, s_2)\big).$$

In general, amino acid alignments are scored with substitution matrices that assign a different similarity score to each pair of amino acid residues. Typically, pairs of amino acids with similar properties have higher scores than pairs with divergent properties. Examples of substitution matrices used to score alignments include the PAM and BLOSUM matrices. We will discuss how such substitution matrices are derived later in the semester. For now, we consider a simple similarity scoring function that treats all symbols in Σ equally. This simple scoring function has just three values; a score for matching symbols (M), a score for a mismatch (m), and a gap score (g):

$$\begin{aligned} p(x, x) &= M, \\ p(x, y) &= m, \\ p(x, \_) &= g. \end{aligned} \tag{1.3}$$

In order to obtain alignments that make sense with similarity scoring, several constraints are imposed on the values of M, m, and g. First, we require that $M > m$, because matches are preferred over mismatches. We further require that a substitution be preferred over two gaps (i.e., $m > 2g$). A scoring function with $m < 2g$ would exclude the possibility of an alignment with substitutions, because the score of any substitution could be improved by replacing it with two gaps.

Note that for both distance and similarity scoring, the score of an alignment is defined to be the sum of the scores for the individual positions in the alignment (Equations 1.1 and 1.2), which implies that each position in the alignment is independent of neighboring positions. This assumption is unrealistic: In real biomolecular sequences, there can be interactions between neighboring, or even distant, residues in the sequence. However, scoring functions that assume positional independence are widely used because they greatly simplify the calculation of alignment scores and other mathematical analyses.

1.1.2 A dynamic programming algorithm to align a pair of sequences

We now have a formal definition of an alignment and a way of assigning a numerical score to any given alignment. How do we find the alignment with the optimal score? We could generate all possible alignments, score each one, and choose the alignment with the best score. However, the computational cost would be prohibitive, since the size of the space of all possible alignments of s_1 and s_2 is $O(2^{n_1+n_2})$. (Convince yourself this is the case.)

Dynamic programming can be used to find the optimal alignment efficiently. This strategy takes advantage of the fact that every prefix of an optimal pairwise alignment is the optimal alignment of a prefix of s_1 and a prefix of s_2. This means that the optimal alignment of pairs of progressively longer prefixes of s_1 and s_2 can be obtained by extending the optimal alignment of shorter prefixes of s_1 and s_2. It is not necessary to examine a suboptimal alignment of prefixes in order to find the optimal alignment of the full length strings.

The dynamic programs for all three sequence alignment problems compute a matrix $\mathcal{A}$, where $\mathcal{A}[i,j]$ is the score of the optimal alignment of the prefixes $s_1[1..i]$ and $s_2[1..j]$, that is, the prefixes of s_1 and s_2 that end at positions i and j, respectively.

Dynamic programming algorithms for sequence alignment have four components:

- Initialization of the first row and column of $\mathcal{A}$.
- A recurrence relation that specifies how to calculate the value of $\mathcal{A}[i,j], i > 0, j > 0$, from the values of neighboring cells.
- Determination of the score of the optimal alignment from the entries in matrix $\mathcal{A}$.
- A procedure to trace back through the matrix to obtain the optimal alignment.

The details of each of these steps are what differentiate global, semi-global, and local alignment. In all cases, the dynamic program proceeds from the upper left to the lower right corner of $\mathcal{A}$, calculating the cost of progressively longer optimal alignments of prefixes. The details of dynamic programming for global alignment are given below for both distance and similarity scoring functions.

Global alignment with distance scoring:

Input:

Sequences s_1 and s_2 of lengths n_1 and n_2, respectively.

Initialization:

$$\begin{aligned} \mathcal{A}[0,j] &= \mathcal{A}[0,j-1] + d(\_, s_2[j]) \qquad \text{(top row)} \\ \mathcal{A}[i,0] &= \mathcal{A}[i-1,0] + d(s_1[i], \_) \qquad \text{(left column)} \end{aligned}$$

Recurrence:

$$\mathcal{A}[i,j] = \min \begin{cases} \mathcal{A}[i-1,j] + d(s_1[i], \_) \\ \mathcal{A}[i-1,j-1] + d(s_1[i], s_2[j]) \\ \mathcal{A}[i,j-1] + d(\_, s_2[j]) \end{cases} \tag{1.4}$$

Store the indices of the entry (or entries) in $\mathcal{A}$ that minimize the right hand side of Equation 1.4 in an n_1 x n_2 matrix, $\mathcal{T}$, which we call the traceback matrix.

Trace back:

Follow the pointers from $\mathcal{T}[n_1, n_2]$ to $\mathcal{T}[0,0]$ to obtain the optimal alignment.

Output:

The optimal alignment score, $\mathcal{A}[n_1, n_2]$.

The optimal global alignment of s_1 and s_2 with respect to distance function, D.

The dynamic programming algorithm for global alignment with similarity scoring has the same general structure as the global alignment algorithm for distance scoring. However, the details of the initialization and recurrence differ.

Global alignment with similarity scoring:

Initialization:

$$\begin{aligned} \mathcal{A}[0,j] &= \mathcal{A}[0,j-1] + g \\ \mathcal{A}[i,0] &= \mathcal{A}[i-1,0] + g \end{aligned}$$

Recurrence relation:

$$\mathcal{A}[i,j] = \max \begin{cases} \mathcal{A}[i-1,j] + g \\ \mathcal{A}[i-1,j-1] + p(i,j) \\ \mathcal{A}[i,j-1] + g \end{cases} \tag{1.5}$$

Store the indices of the entry in $\mathcal{A}$ that maximize the right hand side of Equation 1.5 in a traceback matrix, $\mathcal{T}$.

Traceback:

From $\mathcal{T}[n_1, n_2]$ to $\mathcal{T}[0,0]$ to obtain the optimal alignment.

Output:

The optimal alignment score, $\mathcal{A}[n_1, n_2]$.

The optimal global alignment of s_1 and s_2 with respect to similarity function, s_1.

At each step, the algorithm computes the value of $\mathcal{A}[i, j]$ from the values of $\mathcal{A}[i-1, j], \mathcal{A}[i, j-1]$, and $\mathcal{A}[i-1, j-1]$. With distance scoring, all entries in $\mathcal{A}$ are non-negative, since $d(x, y) \geq 0, \forall\, x, y$. With a similarity scoring function, the entries in $\mathcal{A}$ may be positive or negative. The indices of the entry in $\mathcal{A}$ that optimize the right hand side of the recurrence (Equations 1.4 and 1.5) are stored in a matrix, $\mathcal{T}$. These pointers are used to reconstruct the alignment that gave the optimal score. The algorithm continues until all entries in the matrix $\mathcal{A}$ have been assigned values. This algorithm computes the scores of all pairs of prefixes in $O(n_1 \cdot n_2)$ time. The trace back through the alignment matrix to obtain the optimal alignment requires $O(n_1 + n_2)$ time. Note that there may be more than one optimal alignment.

1.2 Semi-global alignment

Global alignment seeks the best, full length alignment of a pair of sequences; that is, the best way to match up two sequences along their entire length. For some applications, it is desirable to relax this requirement and not penalize gaps at the beginning and/or end of an alignment. For example, for sequence assembly, we seek sequence fragments that overlap; that is, we expect to be able to align the end of one fragment with the beginning of another. Very occasionally, we may find sequence fragments that start and end at the same position, but, in general, we expect some gaps at the beginning and at the end of the alignment. Another example is aligning cDNA's with genomic DNA to identify gene structure. Because the cDNA corresponds to a small region in the genome, the cDNA fragment will be flanked by gaps at both ends when aligned with the genomic DNA.

Semi-global alignment is a modification of global alignment that allows the user to specify that gaps will be penalty-free at the beginning of one of the sequences and/or at the end of one of the sequences. Given sequences s_1 and s_2, there are eight possible cases to consider:

1. Gaps are penalty-free at the beginning of s_1; *e.g.*,

```
s_1:   _ _ D O
s_2:   R E D O
```

2. Gaps are penalty-free at the beginning of s_2; *e.g.*,

```
s_1:   R E D O
s_2:   _ _ D O
```


3. Gaps are penalty-free at the end of s_1; *e.g.*,

```
s_1:   D O _ _
s_2:   D O N E
```

4. Gaps are penalty-free at the end of s_2; *e.g.*,

```
s_1:   D O N E
s_2:   D O _ _
```

5. Gaps are penalty-free at the beginning and end of s_1; *e.g.*,

```
s_1:   _ _ D O _ _
s_2:   R E D O N E
```

6. Gaps are penalty-free at the beginning and end of s_2; *e.g.*,

```
s_1:   R E D O N E
s_2:   _ _ D O _ _
```

7. Gaps are penalty-free at the beginning of s_1 and at the end of s_2; *e.g.*,

```
s_1:   _ _ D O N E
s_2:   R E D O _ _
```

8. Gaps are penalty-free at the beginning of s_2 and at the end of s_1; *e.g.*,

```
s_1:   R E D O _ _
s_2:   _ _ D O N E
```

In semi-global alignment, we do not allow gaps at the beginning of s_1 and the beginning of s_2 in the same alignment. Nor do we not allow gaps at the end of s_1 and the end of s_2. Why not?

Like global alignment, the optimal semi-global alignment can be found using dynamic programming with either distance or similarity scoring. Below, we describe the modifications that are required to adapt the dynamic program for global pairwise alignment to the semi-global alignment problem. These modifications are described in terms of alignment with similarity scoring. Similar modifications can be made to obtain a semi-global alignment algorithm that uses distances.

Semi-global alignment with similarity scoring:

Initialization:

To allow gaps at the beginning of s_1 (Case 1), set $\mathcal{A}[0, j] = 0, \forall_j$; i.e., the first row is zero. The first column is initialized as in global alignment.

To allow gaps at the beginning of s_2 (Case 2), set $\mathcal{A}[i, 0] = 0, \forall_i$; i.e., the first column is zero. The first row is initialized as in global alignment.

Recurrence relation:

Same as global.

Optimal alignment score and trace back:

To avoid trailing gap penalties at the end of s_1 (Case 3), we define the optimal score to be $\max_j \mathcal{A}[n_1, j]$, the optimal score in the bottom row. Trace back from $\mathcal{T}[n_1, j^*]$, where $j^* = \text{argmax}_j \mathcal{A}[n_1, j]$. In other words, trace back from the cell(s) in the last row with optimal score.

To avoid trailing gap penalties at the end of s_2 (Case 4), we define the optimal score to be $\max_i \mathcal{A}[i, n_2]$, the optimal score in the last column. Trace back from $\mathcal{T}[i^*, n_2]$, where $i^* = \text{argmax}_i \mathcal{A}[i, n_2]$. In other words, trace back from the cell(s) in the last column with optimal score.

Note that when the first row (or column) of the matrix is initialized to zero, the traceback will end in the first row (or column), but not necessarily in the cell $\mathcal{A}[0, 0]$.

Like global alignment, either distance or similarity scoring can be used for semiglobal alignment. There may be more than one optimal semiglobal alignment.

1.3 Local pairwise alignment

Global and semiglobal alignment are used in cases where we expect that s_1 and s_2 are related from end to end. Semi-global allows for some gaps at the beginning or end of one sequence, but the underlying assumption is the same: s_1 and s_2 share a relationship within the entire aligned region. In contrast, local alignment is used in cases where s_1 and s_2 share one or more local regions that are related, but are not related from end to end.

The alignment of any substring $s_1[h \ldots i]$ of s_1 and any substring $s_2[j \ldots k]$ of s_2 is a local alignment of s_1 and s_2. The optimal alignment of $s_1[h \ldots i]$ and $s_2[j \ldots k]$ is the highest scoring global alignment of those substrings. where $1 \leq h \leq i \leq n_1$, and $1 \leq j \leq k \leq n_2$ and S is the similarity scoring function defined in Equation 1.2. Note that there may be

more than one The optimal local alignment of s_1 and s_2 is the highest scoring optimal alignment of all possible substrings of s_1 and s_2, that is,

$$\alpha^*(s_1, s_2) = \underset{h,i,j,k}{\text{argmax}}\, S(\alpha^*(s_1[h \ldots i], s_2[j \ldots k])),$$

where $1 \leq h \leq i \leq m$, and $1 \leq j \leq k \leq n$ and S is the similarity scoring function defined in Equation 1.2. Note that there may be more than one optimal local alignment. High scoring sub-optimal alignments may also be of interest.

For local alignment, the pairwise alignment dynamic programming algorithm must be modified to allow the alignment to start and stop anywhere in s_1 and s_2. Unlike the dynamic programs for global alignment, the local alignment recurrence (Equation 1.6) has a fourth term that sets the score $\mathcal{A}[i, j]$ to zero whenever adding a substitution or a gap to the alignment results in a negative score. This is what allows the local alignment algorithm to consider all possible starting positions in s_1 and in s_2.

Local alignment with similarity scoring:

Initialization:

Set the first row and column to zero: $\mathcal{A}[i, 0] = 0$ and $\mathcal{A}[0, j] = 0$, for all i and j.

Recurrence:

$$\mathcal{A}[i, j] = max \begin{cases} \mathcal{A}[i, j-1] + g \\ \mathcal{A}[i-1, j-1] + p(s_1[i],\ s_2[j]) \\ \mathcal{A}[i, j-1] + g \\ 0 \end{cases} \tag{1.6}$$

For non-zero entries in $\mathcal{A}$, store the indices of the entry in $\mathcal{A}$ that maximize the right hand side of Equation 1.6 in a traceback matrix, $\mathcal{T}$.

Optimal alignment score:

The score of the optimal alignment is $\max_{i,j} \mathcal{A}[i, j]$, where the maximum is taken over all i and all j.

Trace back:

Trace back starting at $\mathcal{T}[i^*, j^*]$, where $(i^*, j^*) = \text{argmax}_{i,j}\, \mathcal{A}[i, j]$ is the cell corresponding to the maximum score. End the trace back at the first cell with value zero encountered.

The point at which $\mathcal{A}[i, j]$ drops below zero depends on the scoring function and critically determines what the resulting alignment will look like. For this reason, scoring functions for local alignment are subject to more stringent constraints than scoring functions for global and semi-global alignment.

In order to find biologically meaningful conserved regions, a scoring function for local pairwise alignment must satisfy the following requirements:

- $M > m > 2g$.
- The scoring function must be a similarity function. The local alignment that minimizes the edit distance (weighted or unweighted) is the empty alignment, which tells us nothing.
- There must be at least one pair of residues, x and y, for which the similarity score $p(x,y)$ is positive. Otherwise, the optimal alignment is always the empty alignment.
- The expected alignment score of a pair of randomly generated sequences (i.e., sequences sampled from a background distribution) must be negative.

These rules apply to general scoring functions, where the score for each pair of residues can have a different numerical value. For our simple similarity function, we require a positive score for matches ($M > 0$) and a negative score for mismatches ($m < 0$) and gaps ($g < 0$).

1.4 Multiple Sequence Alignment

In multiple sequence alignment, the goal is to align k sequences, so that residues in each column share a property of interest, typically descent from a common ancestor or a shared structural or functional role. Applications of multiple sequence alignment include identifying functionally important mutations, predicting RNA secondary structure, and constructing phylogenetic trees.

Given sequences $s_1,...,s_k$ of lengths $n_1,...,n_k$, $\alpha = \{s'_1,...,s'_k\}$ is an alignment of $s_1,...,s_k$ if and only if

- $s'_a \in (\Sigma')^*$, for $1 \leq a \leq k$
- $|s'_a| = l$, for $1 \leq a \leq k$, where $l \geq \max(n_1, \ldots, n_k)$
- s_a is the sequence obtained by removing gaps from s'_a
- No column contains all gaps

1.4.1 Scoring a multiple alignment

As with pairwise alignment, multiple sequence alignments (MSAs) are typically scored by assigning a score to each column and summing over the columns. The most common approach to scoring individual columns in a multiple alignment is to calculate a score for each pair of symbols in the column, and then sum over the pair scores. This is called sum-of-pairs or SP-scoring. For global multiple sequence alignment, SP-scoring can be used with either a distance metric or a similarity scoring function. The sum-of-pairs similarity score of an alignment of k sequences is

$$S_{sp}(s'_1, \ldots, s'_k) = \Sigma_{i=1}^{l} \Sigma_{a=1}^{k} \Sigma_{b>a}\ p(s'_a[i], s'_b[i]), \tag{1.7}$$

where l is the length of the alignment. As before, $p(x, y)$ is a numerical score that represents the similarity of x and y and $p(x, \_)$ is the gap score. Further, we define $p(\_, \_)$ to be zero. In pairwise alignment there is no need to assign a value to $p(\_, \_)$, because the definition of a pairwise alignment specifies that no column may contain two gaps. However, in a multiple alignment, two aligned sequences can have a gap in the same column (i.e., $s'_a[i] = s'_b[i] = \_$), as long as there exists at least one sequence in the MSA that does not have a gap in that column.

As an example, let us calculate the SP-score for the alignment of three sequences shown below:

```
s1   A_TT
s2   A_T_
s3   ACAT
```

We can calculate the SP-score for each column separately:

```
              A_TT
              A_T_
              ACAT
           ---------
s1, s2     M0Mg
s1, s3     MgmM
s2, s3     Mgmg
```

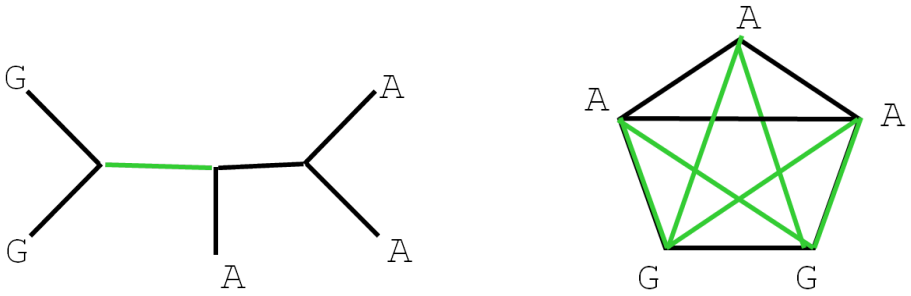
Note that the second column contains two gaps and that these are assigned a score of zero. The total SP-score is $5M + 2m + 4g$. (Is this alignment optimal? If not, how could you improve it?)

We can also use sum-of-pairs with distance scoring for global multiple alignment. This is how we would score the same alignment using unweighted edit distance:

	A_TT A_T_ ACAT
s_1, s_2	0001
s_1, s_3	0110
s_2, s_3	0111

The sum-of-pairs edit distance for this alignment is 6.

Sum-of-pairs scoring tends to overestimate the number of mutations required to explain the data. For example, a single mutation is required to explain the column (A, A, A, G, G), when scored on the tree in Fig. 1.1a. In contrast, SP-scoring assigns this column a score of six (Fig 1.1b), because SP-scoring is based on the implicit assumption that each pair of symbols is independent of all other pairs.



(a) (b)

Figure 1.1: Two ways of scoring the column (A, A, A, G, G) in a multiple alignment. Green edges represent mismatches. **(a)** Scoring mutations on a tree. **(b)** Sum-of-pairs scoring

Scoring an alignment on a tree, also known as tree alignment, is based on the assumption that the residues in the columns of the multiple sequence alignment share an evolutionary history and that this history can be expressed as a single tree for all columns.

Given a known tree topology as input, the k extant sequences are associated with the k leaves of the tree. Sequences for the internal nodes are selected such that the sum of edge costs, i.e. the total number of mutations required along the branches of the tree, is minimized. Under this model, the cost of an edge (X_i, X_j) in the tree is the minimum number of mutations required to transform sequence X_i into sequence X_j.

In order to use this approach, several issues must be resolved. First, a tree topology is needed. In general, the underlying tree is not known. In fact, multiple sequence alignments are generally used to estimate evolutionary trees and not vice versa. Second, tree alignment methods are often based on the assumption that every column in the alignment has the same underlying tree topology. For many sequences, such as those that that have undergone

domain shuffling, this is not the case. Third, in order to compute the branch costs of the tree, we must infer the ancestral sequences associated with the internal nodes. Tree alignment has historically been based exclusively upon the parsimony criterion; that is, on the assumption that mutations are rare and the minimum number of evolutionary steps required to explain the data is the best evolutionary hypothesis. Data that does not happen to be parsimonious can favor the wrong tree model. In addition, column-oriented optimization approaches to MSA usually assume that sequence positions are independent and identically distributed. These assumptions frequently do not hold for biological sequence data.

Finally, for some data sets, a tree may not be a suitable model for describing the relationship between residues in each column, for example, when property of interest is functional or structural. When alignment is used to study function or structure, residues in a column do not always share a common ancestor. Although residues that share a functional or structural role often also share an evolutionary history, this is not the case when functional or structural roles migrate to neighboring residues. For all of these reasons, tree alignment is rarely used in practice.

Given two sequences s_a and s_b in a multiple alignment, the pairwise alignment of s_a and s_b induced by the MSA is the alignment obtained by deleting the other sequences in the MSA and then removing any column that contains two gaps. For example, in the multiple alignment below,

```
AC_T_G
A_GT_G
ACGTAG
```

the induced alignment of the first two sequences is

```
AC_TG
A_GTG.
```

Further, the pairwise alignment induced by the optimal multiple alignment is not necessarily the optimal pairwise alignment. In this example, the optimal pairwise alignment is

```
ACTG
AGTG.
```

Although the optimal pairwise alignment may have a better score, the induced pairwise alignment may be a biologically more realistic alignment because it reflects properties of the family as a whole.

1.4.2 A dynamic programming algorithm for multiple alignment

The dynamic programming algorithm used for finding the optimal global alignment of two sequences can be extended to the problem of global alignment of k sequences. First, let us consider a dynamic program to align three sequences using a sum-of-pairs similarity score.

Global alignment of three sequences with similarity scoring:

Input:

Sequences s_1, s_2, and s_3 of lengths n_1, n_2, and n_3, respectively.

Initialization:

$$\begin{aligned}\mathcal{A}[i_1, 0, 0] &= \mathcal{A}[i_1 - 1, 0, 0] + 2g \\ \mathcal{A}[0, i_2, 0] &= \mathcal{A}[0, i_2 - 1, 0] + 2g \\ \mathcal{A}[0, 0, i_3] &= \mathcal{A}[0, 0, i_3 - 1] + 2g\end{aligned}$$

Recurrence:

$$\mathcal{A}[i_1, i_2, i_3] = \max \begin{cases} \mathcal{A}[i_1 - 1, i_2, i_3] + 2g \\ \mathcal{A}[i_1, i_2 - 1, i_3] + 2g \\ \mathcal{A}[i_1, i_2, i_3 - 1] + 2g \\ \mathcal{A}[i_1 - 1, i_2 - 1, i_3] + 2g + p(s_1[i_1], s_2[i_2]) \\ \mathcal{A}[i_1 - 1, i_2, i_3 - 1] + 2g + p(s_1[i_1], s_3[i_3]) \\ \mathcal{A}[i_1, i_2 - 1, i_3 - 1] + 2g + p(s_2[i_2], s_3[i_3]) \\ \mathcal{A}[i_1 - 1, i_2 - 1, i_3 - 1] + p(s_1[i_1], s_2[i_2]) + p(s_1[i_1], s_3[i_3]) + p(s_2[i_2], s_3[i_3]) \end{cases}$$

Store the indices of the entry in $\mathcal{A}$ that maximize the right hand side of the recurrence in a trace-back matrix, $\mathcal{T}$.

Trace back:

From $\mathcal{T}[n_1, n_2, n_3]$ to $\mathcal{T}[0, 0, 0]$ to obtain the optimal alignment.

Output:

The optimal alignment score, $\mathcal{A}[n_1, n_2, n_3]$.

The optimal alignment of s_1, s_2, and s_3 with respect to similarity function, $\mathcal{S}$.

The dynamic program for multiple sequence alignment has the same structure as the algorithms for pairwise sequence alignment, but the initiation and recurrence steps are more complex. Since the alignment matrix, $\mathcal{A}$, is a 3-dimensional matrix, the first row in each of the three dimensions must be initialized. The algorithm calculates the entries in $\mathcal{A}$ according to the recurrence proceeding diagonally from $\mathcal{A}[0,0,0]$ to $\mathcal{A}[n_1, n_2, n_3]$. As in the pairwise case, a trace-back matrix, $\mathcal{T}$, is used to record the indices that gave the optimal score for each i_1, i_2, i_3 prefix. Once the entire matrix has been filled in, the optimal score is found in $\mathcal{A}[n_1, n_2, n_3]$.

It is straightforward, if messy, to generalize the dynamic program for three sequences to a dynamic program for k sequences. To convince yourself that you understand how this works, try writing down the algorithm for four sequences. For three sequences, the recurrence has seven entries. How many entries will there be in the recurrence when $k = 4$? How many entries will there be for arbitrary k?

The computational complexity of the dynamic programming algorithm to align k sequences is $O(n^k 2^k k^2)$. To see this, note that for k sequences, the alignment matrix has $O(n^k)$ entries. For each entry in $\mathcal{A}$, the recurrence relation considers $O(2^k)$ neighboring cells. Calculating the SP-score for each of those neighbors requires $O(k^2)$ time. (Why?) Thus, the time complexity of the dynamic program for multiple sequence alignment is exponential in the number of sequences. Given 10 sequences of length at most 500, it is possible to calculate the optimal alignment using dynamic programming. For larger problem instances, a heuristic is typically used.

1.4.3 Heuristics for global multiple alignment

The dynamic programming approach to global multiple sequence alignment is framed as an optimization problem. In this approach, we design an optimization criterion over the set of all possible MSAs and then seek the MSA that optimizes this criterion using dynamic programming. The advantage of this approach is that the optimization criterion makes explicit the assumptions upon which the optimization is based. Because they are explicit, these assumptions are open to scrutiny and falsification.

However, this formal optimization approach has disadvantages as well. One, as we have already seen, is that the computational complexity is exponential in the number of sequences. A second problem is the selection of an optimization criterion. In computational biology, the optimization criterion must follow a specific biological model relating the data to the evolutionary, structural, or functional question at hand. If the optimization criterion is not directly linked to a biological model, then the optimal solution may not reflect biological relationships. As we have seen, both sum-of-pairs and tree alignment have limitations in how well they capture the underlying biology.

In practice, the most widely used multiple alignment programs are based on heuristic methods, not only because of the exponential running time of the exact algorithm, but

also because heuristics often give MSAs that are more convincing biologically, even though they do not guarantee mathematically optimal alignments. A survey of multiple alignment software based on heuristic methods, *Protein multiple sequence alignment* by Do and Katoh, 2008, is posted in the "Optional reading" section of the course syllabus.

The performance of MSA programs is typically evaluated empirically using curated or automated structural alignments. BALiBase (http://www.lbgi.fr/balibase/), a collection of "high quality, manually refined, reference alignments based on 3D structural superpositions"[2], is one of the most widely used benchmarks. The BALiBase reference data sets are designed to mimic properties of different types of data sets encountered in practice, especially those that are challenging to align. Examples of challenging data sets include highly divergent sequences that are variable in length and have less than 50% identity, related sequences combined with several outlier, or "orphan", sequences, and related sequences that differ due to large insertions, deletions or terminal extensions.

One of the most commonly used heuristic strategies is *progressive alignment*. This approach is used in a number of programs, including the widely-used CLUSTAL family of multiple alignment programs. Given k sequences, $s_1, \ldots, s_k$, of lengths $n_1, \ldots, n_k$, progressive methods construct an alignment as follows:

1. Construct pairwise alignments for all pairs of sequences.
2. Compute $\mathcal{D}$, the matrix of pairwise distances, where $\mathcal{D}[a, b]$ is the distance between sequences s_a and s_b. Note that $\mathcal{D}$ is a symmetric matrix with zeros on the diagonal.
3. Construct a "guide tree", T, from $\mathcal{D}$. T is a rooted tree with k leaves corresponding to the k sequences.
4. Construct an MSA by repeatedly merging intermediate multiple alignments to obtain progressively larger alignments, until all k sequences have been incorporated in the alignment. The order of merging is determined by the guide tree, T.

The merge operation in Step 4 takes as input two multiple alignments of k_1 sequences and k_2 sequences and returns a multiple alignment of $k_1 + k_2$ sequences. This is repeated until all k sequences are incorporated into the alignment. The order in which sequences are merged is determined by a bottom up traversal of the guide tree. For example, if the tree in Fig. 1.2 were the guide tree, then the pairwise alignment of s_1 and s_2 would be merged with the pairwise alignment of s_3 and s_4, yielding an intermediate MSA of four sequences. A similar merging of two pairwise alignments would result in the MSA of s_5, s_6, s_7 and s_8. Finally, these two alignments, of four sequences each, would be merged to obtain a full alignment of eight sequences.

The actual merge operation is carried out using the pairwise global alignment algorithm to align the two alignments, where the input alignments are treated as sequences over an

[2] "BAliBASE 3.0: latest developments of the multiple sequence alignment benchmark." Thompson JD, Koehl P, Ripp R, Poch O., Proteins. 2005 Oct 1;61(1):127-36.

expanded alphabet. Durbin calls this a "profile." Two aligned sequences can be viewed as a sequence over the alphabet $\Sigma' \times \Sigma' \setminus \{(\_\,\_)\}$. For example, when $\Sigma = \{A, C, G, T\}$, this alphabet contains 24 symbols ($\{AA, AC, AG, AT, A\_, CA, \ldots \_T\}$).

We illustrate profile alignment with the case where a multiple alignment of three sequences is obtained by aligning a profile t with a single sequence s. The elements in $s[i]$ are of the form $\binom{x}{y}$, $\binom{x}{\_}$, or $\binom{\_}{y}$, where x and y are symbols in Σ. The score for aligning $t[i]$ with $s[j]$ is the sum of the scores for aligning the first character in $t[i]$ with $s[j]$ and the second character in $t[i]$ with $s[j]$. For example, when similarity scoring is used and $s[i]$ is of the form, $\binom{x}{y}$, the recurrence relation for calculating the alignment matrix $\mathcal{A}[i, j]$ is

$$\mathcal{A}[i,j] = \max \left\{ \begin{array}{l} \mathcal{A}[i-1, j-1] + p(x, s[j]) + p(y, s[j]), \\ \mathcal{A}[i, j-1] + 2g, \\ \mathcal{A}[i-1, j] + 2g \end{array} \right\}. \tag{1.8}$$

When t[i] contains an indel (i.e., $t[i]$ is of the form $\binom{x}{\_}$), the recurrence relation for calculating $\mathcal{A}[i, j]$ is

$$\mathcal{A}[i,j] = \max \left\{ \begin{array}{l} \mathcal{A}[i-1, j-1] + p(x, s[j]) + g, \\ \mathcal{A}[i, j-1] + 2g, \\ \mathcal{A}[i-1, j] + g \end{array} \right\}. \tag{1.9}$$

Note that $p(x, y)$, the similarity of x and y, does not appear in the right hand side of Equation 1.8. Similarly, the gap score for the alignment of $\binom{x}{\_}$ does not appear in the recurrence in Equation 1.9. This is because the two symbols in $s[i]$ were compared and scored during the pairwise alignment in a previous step in the progressive alignment procedure.

A key aspect of the merging operation is that we are not allowed to modify the profiles being aligned. For the example above, that means we cannot change juxtaposition of the two symbols in $t[i]$, even if modifying the alignment in t would result in a better alignment with s. This is called the "once a gap, always a gap" rule (although it also applies to mismatches). A consequence of this rule is that if a bad decision is made with regard to the placement of gaps early in the procedure, then that bad decision will propagate through subsequent iterations and cannot be corrected. It is this rule that makes progressive alignment a fast heuristic; that is, this rule underlies both the improvement in running time and the possibility that the result may be suboptimal.

The complexity of progressive alignment is $O(k^2n^2)$, where $n = \max\{n_a\}, 1 \leq a \leq k$. Calculating the distance matrix in Step 2 requires $O(k^2)$ pairwise alignments. The cost of each pairwise alignment is $O(n^2)$. The merging process also requires $O(k^2n^2)$ time. The computational complexity of merging depends on the number of profile alignments required, the number of cells in the alignment matrix for each profile alignment, and number of comparisons required for each cell. The size of alignment matrix is $O(n^2)$ in all profile alignments. The number of comparisons for a single cell in an alignment of two profiles with k_1 and k_2 sequences, respectively, is $O(k_1 \cdot k_2)$.

The number of profile alignments and the values of k_1 and k_2 depend on the shape of the guide tree. At one extreme, we have a completely unbalanced guide tree with k sequences (e.g., Fig. 1.3, where $k = 8$). In this case, each merge represents an alignment of a single sequence ($k_1 = 1$) with a profile of size $k_2 = h$, $1 \leq h \leq k - 1$, resulting in a complexity of

$$\sum_{h=1}^{k-1} n^2 h,$$

which is $O(k^2 n^2)$.

At the other extreme, we have a balanced guide tree with k leaves (e.g., Fig. 1.2). We consider the case when k is a power of 2. We leave the case where k is not a power of two to the masochistic reader. In a balanced guide tree, there are $k/2^h$ merges at height, h. Each of these represents an alignment of two profiles, each comprising 2^{h-1} sequences. The computational complexity is the sum of the complexity of the merges at height h, $1 \leq h \leq \log_2 k$, or

$$\sum_{h=1}^{log_2 k} n^2 \cdot (2^{h-1})^2 \cdot \frac{k}{2^h}.$$

This reduces to

$$kn^2 \cdot \sum_{h=1}^{log_2 k} 2^{h-2}. \tag{1.10}$$

It is easy to verify that the series

$$\sum_{i=1}^{N} 2^{(i-1)} = 2^N - 1. \tag{1.11}$$

Substituting the right hand side of Equation 1.11 into Equation 1.10, where $N = \log_2 k$, we obtain

$$\frac{1}{2}\, kn^2(2^{(\log_2 k)} - 1).$$

This reduces to $(k-1)kn^2/2$, so we again obtain a computational complexity of $O(k^2 n^2)$.

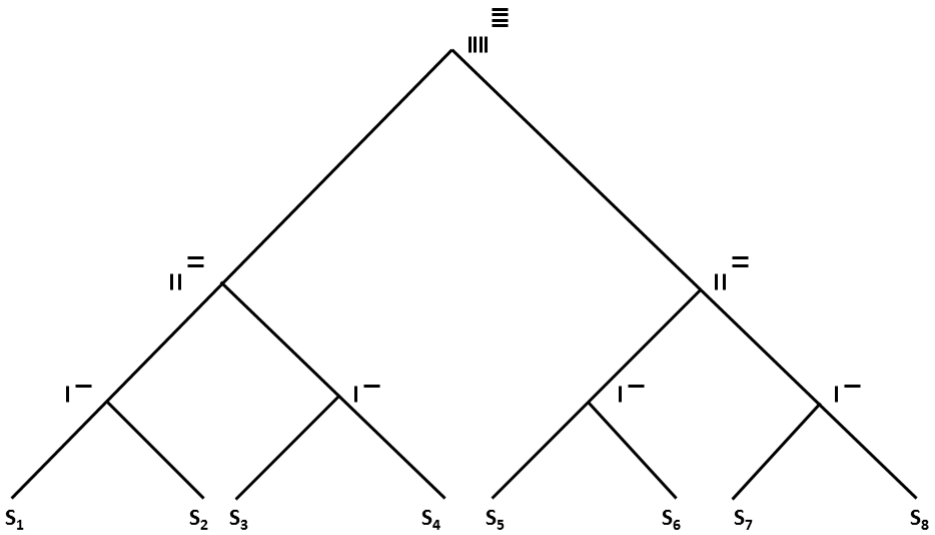


Figure 1.2: A balanced guide tree for 8 sequences.

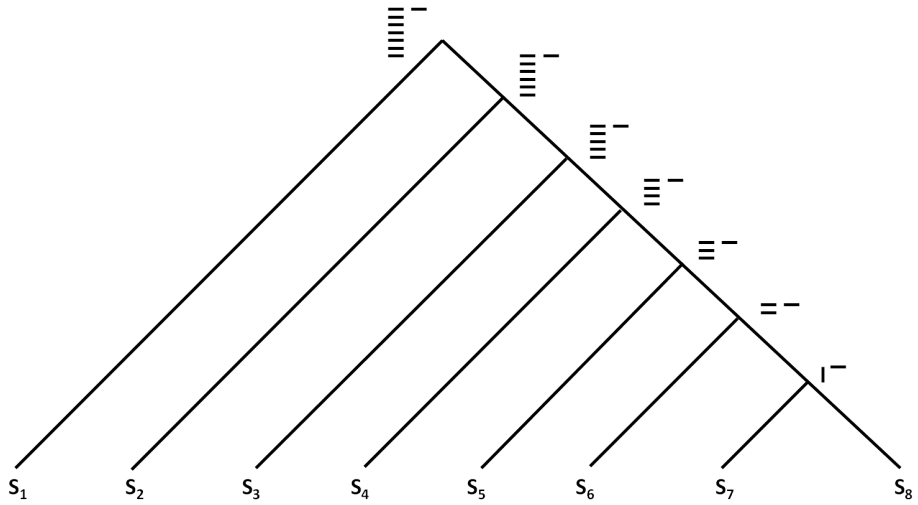


Figure 1.3: An unbalanced guide tree for 8 sequences.

