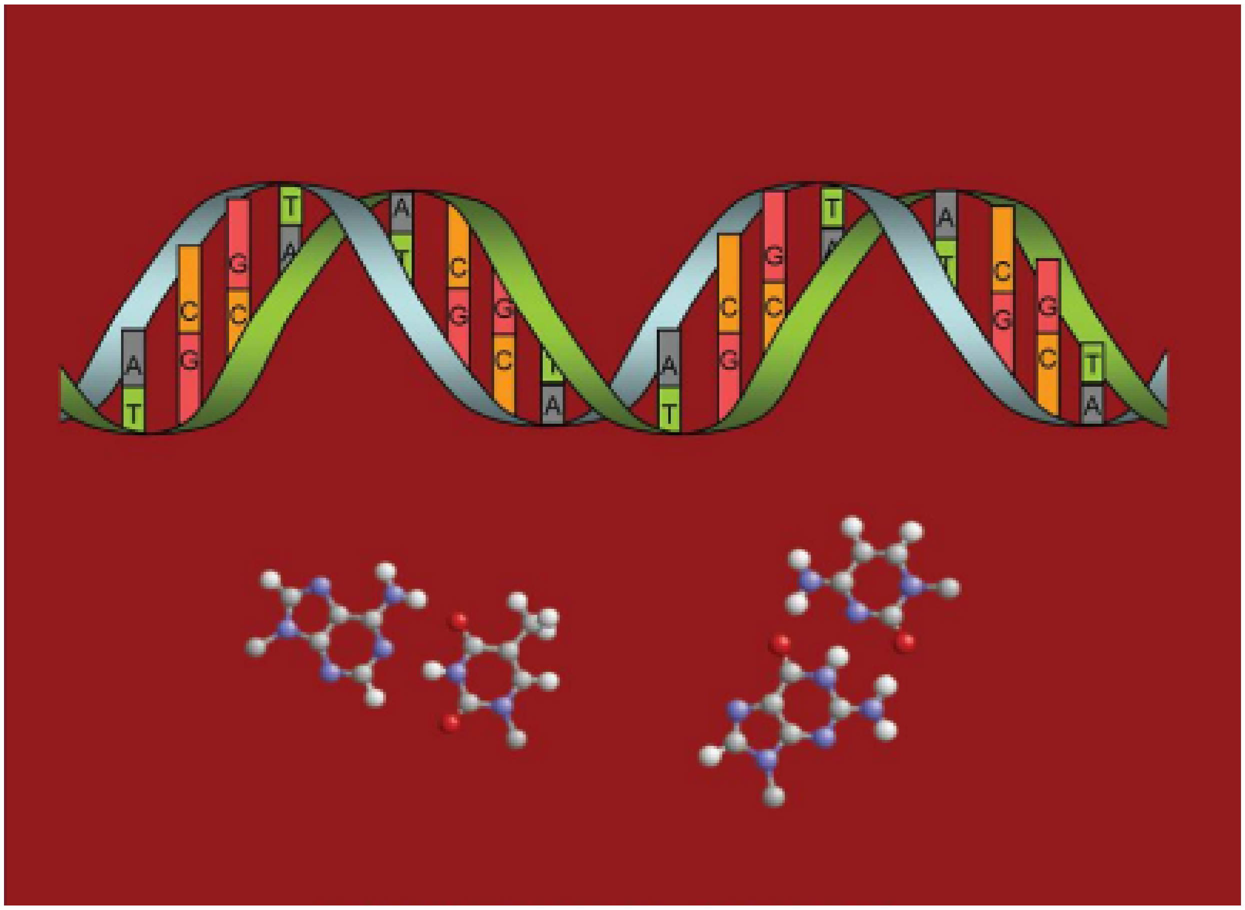


ALGORITHMS IN BIOINFORMATICS

2

Sequence Similarity

2.1 Introduction

There is a well-known conjecture in biology: If any *two protein (or DNA, or RNA) sequences are similar*, they will have *similar functions or 3D structures*. Bioinformaticians often compare the similarity between two biological sequences to understand their structures or functionalities. Below, we show some example applications.

- ***Predicting the biological function of a gene (or an RNA or a protein):*** If a gene is similar to some gene with known function, we can conjecture that this gene also has the same function.
- ***Finding the evolution distance:*** Species evolve through modifying their genomes. By measuring the similarity of the genomes, we could know their evolution distances.
- ***Helping genome assembly:*** Current technology can only decode short DNA sequences. Based on the overlapping information of a huge amount of short DNA pieces, the Human Genome Project reconstructs the whole human genome. The overlapping information is obtained by sequence comparison.
- ***Finding a common region in two genomes:*** If two relatively long strings from two genomes are similar or even identical, they probably represent genes that are homologous (i.e., genes evolved from the same ancestor and with common function).
- ***Finding repeats within a genome:*** The human genome is known to have many similar substrings which are known as repeats. Sequence comparison methods can help to identify them.

As a matter of fact, the reverse of the conjecture may not always be true. For example, the hemoglobin of V. stercoraria (bacterial) and that of P. marinus (eukaryotic) are *similar in structure and function*; *their protein sequences share just 8% sequence identity*.

2.2 Global Alignment Problem

Before describing the global alignment problem, we first revisit a well-known computer science problem, the ***string edit problem***. The string edit problem computes the *minimum number of operations to transform one string to another*, where the operations are:

1. Replace a letter with another letter.
2. Insert a letter into a sequence.
3. Delete a letter from a sequence.

For example, given two strings *S = interestingly* and *T = bioinformatics*, S can be transformed to T using six replacements, three insertions, and two deletions. The *minimum number of operations required to transform S to T is called the* ***edit distance***. For the above example, the edit distance is 11.

In general, we can associate a cost to every edit operation. Let Σ be the alphabet set and '_' be a special symbol representing a null symbol. The cost of each edit operation can be specified by a matrix σ where σ(x, y) equals the cost of replacing x by y, for x, y ∈ Σ∪{_}. Note that σ(_, x) and σ(x,_) denote the cost of insertion and deletion, respectively. Then, the cost of a set of edit operations E equals $\Sigma_{(x,y)\in E}$ σ(x,y). For example, if the costs of mismatch, insert, and delete are 1, 2, and 2, respectively, then the total cost to transform interestingly to bioinformatics is 16.

Unlike ***edit distance*** which measures the *difference between two sequences*, we sometimes want to know the *similarity of two sequences*. The ***global alignment problem*** is aimed at this purpose. First, let's give a formal definition of alignment.

DEFINITION 2.1 *An alignment of two sequences is formed by inserting spaces in arbitrary locations along the sequences so that they* ***end up with the same length*** *and there are* ***no two spaces at the same position*** *of the two augmented sequences.*

For example, a possible alignment of S and T is as follows:

```
-i--nterestingly
bioinformatics--
```

Two sequences are similar if their alignment contains many positions having the same symbols while minimizing the number of positions that are different. In general, we can associate a similarity score with every pair of aligned characters. Let Σ be the alphabet set and _ be a special symbol representing a null symbol. The similarity of a pair of aligned characters can be specified by a *matrix δ, where δ(x, y) equals the similarity of x and y for x, y ∈ Σ∪{_}*. Figure 2.1 shows an example of a *similarity matrix*.

	-	A	C	G	T
-		-1	-1	-1	-1
A	-1	2	-1	-1	-1
C	-1	-1	2	-1	-1
G	-1	-1	-1	2	-1
T	-1	-1	-1	-1	2

Figure 2.1 Example of a similarity score matrix.

The aim of the global alignment problem is to find an alignment A which ***maximizes*** $\boldsymbol{\Sigma_{(x,y)\in A}\, \delta(x, y)}$. Such alignment is called the *optimal alignment*.

There is a one-to-one correspondence between a pair of aligned characters and an edit operation. When a pair of aligned characters are the same, it is called a ***match***; otherwise it is called a ***mismatch***. When a space is introduced in the first sequence, it is called an ***insert*** while a space in the second sequence is called a ***delete***. For the above example, the alignment corresponds to five matches, six mismatches, three inserts, and two deletes.

The *global alignment problem* and the string *edit distance problem* are in fact a ***dual problem***. The lemma below shows that the solutions reported by them are in fact the same.

LEMMA 2.1 *Let σ be the cost matrix of the edit distance problem and δ be the score matrix of the global alignment problem. If δ(x, y) = −σ(x, y) for all x, y ∈ Σ∪{_}, the* ***solution to the edit distance problem*** *is equivalent to the* ***solution to the string alignment problem***.

PROOF Let (x, y), x, y ∈ Σ∪{_}, be an operation that transforms x to y. Recall that insertion of symbol y is (_ , y) and deletion of x is (x, _). The *cost of an operation (x, y) in the edit distance problem is σ(x, y)* and the *score of an operation (x, y) in the string alignment problem is δ(x, y)*.

Given any alignment of two sequences. For any operation (x, y), let n_{xy} *be the number of occurrences of the operation (x, y)*.

Then, the *edit distance* of the alignment is:

$$\sum_{x \in \Sigma \cup \{\_\}} \sum_{y \in \Sigma \cup \{\_\}} n_{x,y}\sigma(x,y)$$

The *alignment score* of the alignment is:

$$\sum_{x \in \Sigma \cup \{\_\}} \sum_{y \in \Sigma \cup \{\_\}} n_{x,y}\delta(x,y)$$

Because ***δ = −σ***, ***minimizing the edit distance*** is equivalent to ***maximizing the alignment score***.

Although *string edit* and *global alignment* are equivalent, people in computational biology prefer to use *global alignment* to measure the similarity of DNA, RNA, and protein sequences.

An example of global alignment. Consider the similarity matrix for {_ , A, C, G, T } in Figure 2.1, where δ(x, y) = 2, −1, −1, −1 for *match*, *mismatch*, *delete*, and *insert*, respectively. One possible alignment of two DNA sequences S = ACAATCC and T = AGCATGC is shown below.

$$\begin{aligned} \mathrm{S} &= \texttt{A-CAATCC} \\ \mathrm{T} &= \texttt{AGCA-TGC} \end{aligned}$$

The above alignment has five matches, one mismatch, one insert, and one delete. Thus, the similarity score of this alignment is 7 ($5 * 2 - 1 - 1 - 1 = 7$). We can check that this alignment has the maximum score. Hence, it is an optimal alignment.

Note that S and T may have more than one optimal alignment. For example, another optimal alignment is as follows.

$$\begin{aligned} \mathrm{S} &= \texttt{A-CAATCC} \\ \mathrm{T} &= \texttt{AGC-ATGC} \end{aligned}$$

2.2.1 Needleman-Wunsch Algorithm

Consider two strings S[1..n] and T[1..m]. To compute the ***optimal global alignment between S and T*** , *the* ***brute-force method*** *generates all possible alignments and reports the alignment with the maximum alignment score*. Such an approach, however, takes exponential time. This section introduces the ***Needleman-Wunsch algorithm*** *which applies* ***dynamic programming*** *to find the optimal global alignment between S and T in* ***O(nm)*** *time*.

	-	A	C	G	T
-		-1	-1	-1	-1
A	-1	2	-1	-1	-1
C	-1	-1	2	-1	-1
G	-1	-1	-1	2	-1
T	-1	-1	-1	-1	2

Figure 2.1 Example of a *similarity score matrix* δ.

Define ***V(i, j)*** to be the ***score of the optimal alignment*** *between* S[1..i] *and* T[1..j]. We devise the *recursive formula* for *V(i, j)* depending on two cases: (1) either i = 0 or j = 0 and (2) both i > 0 and j > 0.

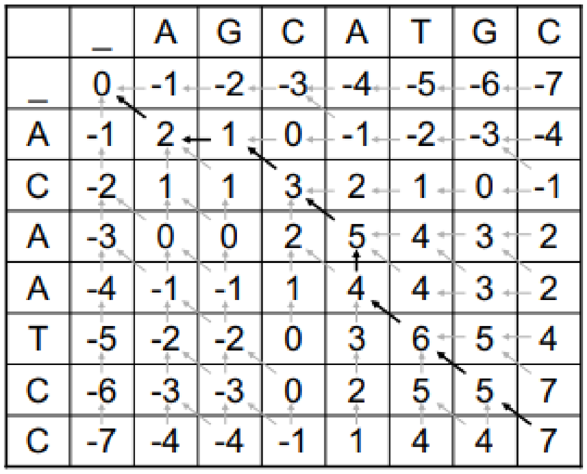
For case (1), when either i = 0 or j = 0, we will align a string with an empty string. In this case, we have either *insertions* or *deletions*. Hence, we have the following equations.

$$\begin{aligned}&V(0,0) = 0\\&V(0,j) = V(0,j-1) + \delta(\_,T[j]) \text{ Insert } j \text{ times}\\&V(i,0) = V(i-1,0) + \delta(S[i],\_) \text{ Delete } i \text{ times}\end{aligned}$$

For case (2), when both i > 0 and j > 0, we observe that, in the best alignment between S[1..i] and T[1..j], *the last pair of aligned characters* should be either *match/mismatch*, *delete*, or *insert*. *To get the optimal score, we choose the maximum value among these three cases*. Thus, we get the following *recurrence relation*:

$$V(i,j) = \max \begin{cases} V(i-1,j-1) + \delta(S[i],T[j]) & \mathtt{match/mismatch} \\ V(i-1,j) + \delta(S[i],\_) & \mathtt{delete} \\ V(i,j-1) + \delta(\_,T[j]) & \mathtt{insert} \end{cases}$$

The optimal alignment score of S[1..n] and T[1..m] is V(n, m). This score can be computed by filling in the table V(1..n, 1..m) row by row using the above recursive equations. Figure 2.2 shows the table V of the two strings *S = ACAATCC* and *T = AGCAT GC*. For example, the value in the entry V(1, 1) is obtained by choosing the maximum of {0+2, −1−1, −1−1}. The score of the optimal alignment is obtained at the bottom right corner of the table V (7, 7) which is 7.



	_	A	G	C	A	T	G	C
_	0	-1	-2	-3	-4	-5	-6	-7
A	-1	2	1	0	-1	-2	-3	-4
C	-2	1	1	3	2	1	0	-1
A	-3	0	0	2	5	4	3	2
A	-4	-1	-1	1	4	4	3	2
T	-5	-2	-2	0	3	6	5	4
C	-6	-3	-3	0	2	5	5	7
C	-7	-4	-4	-1	1	4	4	7

Figure 2.2 The *dynamic programming table* for running the *Needleman-Wunsch algorithm* for two strings *S = ACAATCC* and *T = AGCATGC*.

To recover the optimal alignment, for every entry, we draw arrows to indicate the ways to get the corresponding values. We draw a *diagonal arrow*, a *horizontal arrow*, or a *vertical arrow* for the entry V(i, j) if V (i, j) equals *V(i − 1, j − 1) + δ(S[i], T [j]), V(i, j − 1) + δ(_ , T [j])*, or *V(i − 1, j) + δ(S[i], _)*, respectively. Figure 2.2 shows an example. *The optimal alignment is obtained by back-tracing the arrows from V(7, 7) back to V (0, 0). If the arrow is diagonal, two characters will be aligned. If it is horizontal or vertical, a deletion or an insertion, respectively, will be present in the alignment*. In Figure 2.2, the optimal alignment score is V(7, 7) = 7. Through back-tracing from V(7, 7) to V(0, 0), the path is 0← 2 ← 1 ← 3 ← 5 ← 4 ← 6 ← 5 ← 7. *The corresponding optimal alignment is A−CAAT CC and AGCA−TGC. The optimal alignment is not unique*. We may get another optimal alignment: A − CAATCC and AGC − ATGC.

Finally, we analyze the *time* and *space* complexity of the *Needleman-Wunsch algorithm*. The Needleman-Wunsch algorithm fills in nm entries of the table V(1..n, 1..m). *Each entry requires O(1) word space* and can be *computed in O(1) time*. Hence, the Needleman-Wunsch algorithm computes the *optimal global alignment* of S[1..n] and T [1..m] using ***O(nm) time*** and ***O(nm) space***.

2.2.2 Running Time Issue

As shown in the previous section, *the Needleman-Wunsch algorithm takes O(nm) time to solve the global alignment problem*. In other words, using a 1 GHz computer, it takes *hundreds of years* to align the *mouse* and the *human* genomes. Can we improve the time complexity?

$$H \sim 3\times10^{9}\ bp$$

$$M \sim 2.5\times10^{9}\ bp$$

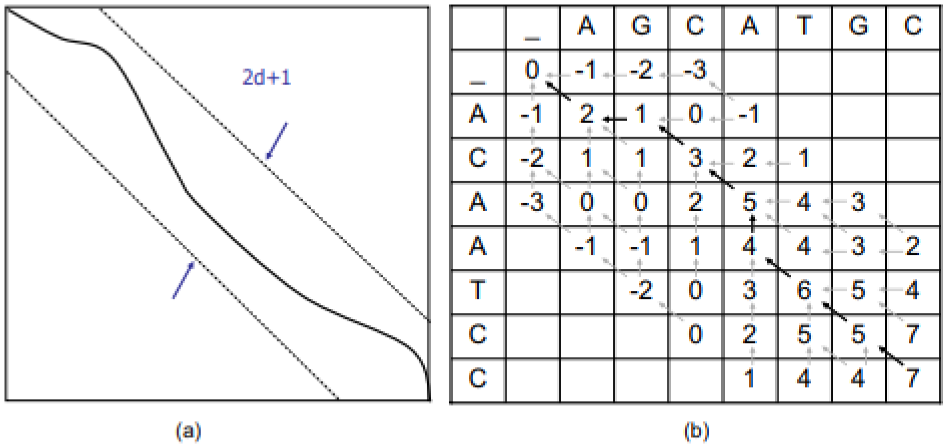
$$\Rightarrow \frac{3\times10^{9} \times 2.5\times10^{9}\times 1\ ns}{365 \times 24 \times 3600 \times 10^{9}\ ns} \approx 237\ years$$

The current best method runs in ***O(nm/log n)*** *time*. The method was proposed by *Masek* and *Paterson* in 1980.

One special case is that *we* ***restrict the maximum number of insertions or deletions*** *(by shorthand, we use indel to denote an insertion or a deletion) in the alignment to be* ***d***. Obviously, *0 < d ≤ n + m*. Recall that, *in table V , an insertion corresponds to a horizontal arrow and a deletion corresponds to a vertical arrow*. Hence, ***the alignment should be inside the (2d + 1) band if the number of indels is at most d*** (see Figure 2.3). *It is unnecessary to execute the Needleman-Wunsch algorithm to fill in the lower and upper triangles in the table V . An algorithm which fills in only the middle band is called the* ***banded Needleman-Wunsch*** *alignment*. *For time analysis, note that the area of the (2d + 1) band in the table V is nm − (n − d)(m − d) = md + nd − d^2*. Since the time for filling in every entry inside the band is O(1), the *running time of the banded Needleman-Wunsch algorithm is* ***O((n + m)d)***.

2.2.3 Space Efficiency Issue

Note that the Needleman-Wunsch algorithm requires O(mn) space as the table V has nm entries. When we compare two very long sequences, memory space will be a problem. For example, if we compare the human genome with the mouse genome, the memory space is at least 9×10^{18} words, which is not feasible in real life. Can we solve the global alignment problem using less space?



	_	A	G	C	A	T	G	C
_	0	-1	-2	-3				
A	-1	2	1	0	-1			
C	-2	1	1	3	2	1		
A	-3	0	0	2	5	4	3	
A		-1	-1	1	4	4	3	2
T			-2	0	3	6	5	4
C				0	2	5	5	7
C					1	4	4	7

Figure 2.3 (a) An illustration of a 2d + 1 band of a dynamic programming table. **(b)** A 2d + 1 band example for the dynamic programming table in Figure 2.2 with d = 3.

Recall that, in the Needleman-Wunsch algorithm, the value of an entry V(i, j) depends on three entries V(i − 1, j − 1), V(i − 1, j), and V(i, j − 1). Hence, *when we fill in the i-th row in table V , only the values in the (i − 1)-th row are needed*.

Therefore, *if we just want to compute the* ***optimal alignment score****, it is unnecessary to store all values in the table V* . Precisely, *when we fill in the i-th row of the table V , we only keep the values in the (i − 1)-th row. In this way, the space complexity becomes* ***O(m)***. This method is called the ***cost-only Needleman-Wunsch algorithm***.

Can we also reconstruct the *optimal alignment*? The answer is YES! Below, we present ***Hirschberg's algorithm*** which computes the *optimal alignment of two strings S[1..n] and T[1..m] in* ***O(n + m) space***.

The main observation is that the *optimal alignment score of (S[1..n], T[1..m])* is the sum of *(1) the optimal alignment score of (S[1..n/2], T [1..j])* and *(2) the optimal alignment score of (S[n/2+1..n], T [j+1..m]) for some j*. Equation 2.1 illustrates this idea. *Note that AS(A, B) denotes the optimal alignment score between sequences A and B.*

$$AS(S[1..n], T[1..m]) = \max_{1\le j\le m} \{AS(S[1..n/2], T[1..j]) + AS(S[n/2+1..n], T[j+1..m])\} \quad (2.1)$$

The *integer j, which maximizes the sum, is called the mid-point. The algorithm FindMid in Figure 2.4 describes how to compute the mid-point using the cost-only Needleman-Wunsch algorithm*.

```
Algorithm FindMid($S[1..n], T[1..m]$)
Require: Two sequences $S[1..n]$ and $T[1..m]$
Ensure: A mid-point $j \in 1..m$
 1: Perform a cost-only Needleman-Wunsch algorithm to compute
    $AS(S[1..n/2], T[1..j])$ for all $j$;
 2: Perform a cost-only Needleman-Wunsch algorithm to compute
    $AS(S[n/2+1..n], T[j+1..m])$ for all $j$;
 3: Determine the mid-point $j$ which maximizes the sum using Equation
    2.1;
```

Figure 2.4 The algorithm FindMid.

Figure 2.5 gives an example demonstrating how the algorithm FindMid computes the mid-point. *Step 1 fills in the first half of the table to compute AS(S[1..n/2], T [1..j]) for all j*. Then, *Step 2 fills in the second half of the table in reverse to compute AS(S[n/2+1..n], T [j+1..m]) for all j*. Finally, *Step 3 computes the score AS(S[1..n/2], T[1..j]) + AS(S[n/2+1..n], T [j + 1..m]), for j = 0, 1,...,m − 1, by getting the m sums of the m diagonal pairs from the two middle rows. Then, we obtain the mid-point j which corresponds to the maximum sum*. Here the maximum score is 7, which is the sum of 4 and 3 and the position j = 4 is determined to be the mid-point.

	_	A	G	C	A	T	G	C	_
_	0	-1	-2	-3	-4	-5	-6	-7	
A	-1	2	1	0	-1	-2	-3	-4	
C	-2	1	1	3	2	1	0	-1	
A	-3	0	0	2	5	4	3	2	
A	-4	-1	-1	1	4	4	3	2	
T		-1	0	1	2	3	0	0	-3
C		-2	-1	1	-1	0	1	1	-2
C		-4	-3	-2	-1	0	1	2	-1
_		-7	-6	-5	-4	-3	-2	-1	0

Figure 2.5 Mid-point example. We split the dynamic programming table into two halves. For the top half, we fill in the table row by row in *top-down order*. Then, we obtain AS(S[1..n/2], T [1..j]) for all j. For the bottom half, we fill in the table row by row in *bottom-up order*. Then, we obtain AS(S[n/2+1..n], T [j + 1..m] for all j. Finally, the mid-point is computed by Equation 2.1.

If we divide the problem into two halves based on the mid-point and recursively deduce the alignments for the two halves, we can compute the optimal alignment while using O(n+ m) word space. The detailed algorithm is shown in Figure 2.6 and the idea is illustrated in Figure 2.7.

```
Algorithm Alignment($S[i_1..i_2], T[j_1..j_2]$)
Require: Two sequences $S[i_1..i_2]$ and $T[j_1..j_2]$
Ensure: The optimal alignment between $S[i_1..i_2]$ and $T[j_1..j_2]$.
  1: If $i_1 = i_2$, compute the alignment using the Needleman-Wunsch algorithm and report it;
  2: Let $mid = (i_1 + i_2)/2$;
  3: $j$ = FindMid($S[i_1..i_2], T[j_1..j_2]$);
  4: Concatenate the alignments computed by Alignment($S[i_1..mid], T[j_1..j]$) and Alignment($S[mid+1..i_2], T[j+1..j_2]$) and report it;
```

Figure 2.6 The recursive algorithm for computing the global alignment of S[$i_1..i_2$] and T [$j_1..j_2$] using *linear space*.

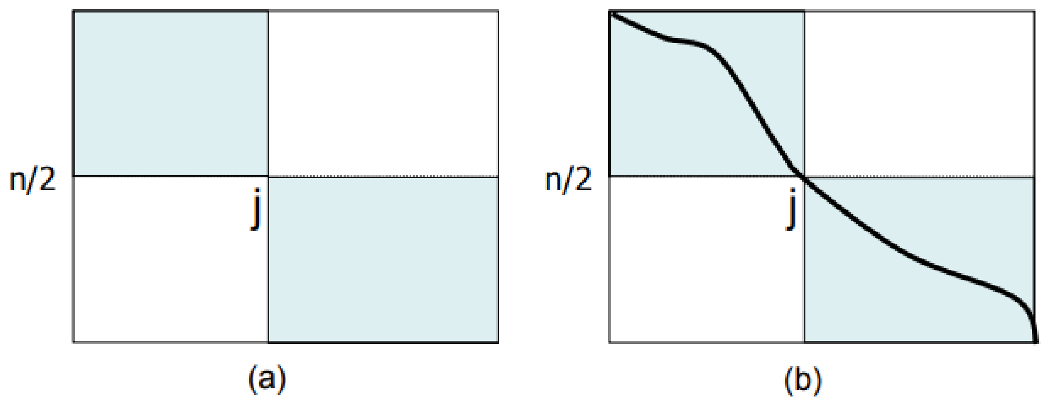


Figure 2.7 (a) When we call Alignment(S[1..n], T [1..m]), we first compute the mid-point j and generate two subproblems Alignment(S[1..n/2], T [1..j]) and Alignment(S[n/2+1..n], T [j + 1..m]). (b) By recursion, we obtain the alignments for the two subproblems. By concatenating the two alignments, we obtain the alignment of S[1..n] and T [1..m].

For algorithm FindMid, both steps 1 and 2 take O(nm/2) time. Step 3 finds the maximum of m sums, which requires O(m) time. In total, FindMid takes O(nm) time.

For algorithm Alignment, let the running time of Alignment(S[1..n], T [1..m]) be Time(n, m). *Time(n, m) = time for finding the mid-point + time for solving the two* *subproblems = O(nm) + Time(n/2, j) + Time(n/2, m − j). By solving the recursive equation, the time complexity is Time(n, m) = O(nm).*

For space complexity, the working memory for finding the mid-point takes O(m) space. Once we find the mid-point, we can free the working memory. ***In each recursive call, we only need to store the alignment path****. Therefore* ***the space complexity is O(m+n)***.

2.2.4 More on Global Alignment

There are two special cases of the global alignment problem. Both of them can be solved using a similar dynamic programming approach with *specific score function*:

1. *Longest Common Subsequence (LCS):* Given two sequences X and Y, a sequence Z is said to be a common subsequence of X and Y if Z is a subsequence of both X and Y. *The LCS problem aims to find a maximum-length common subsequence of X and Y.* For example, the LCS of "catpaplte" and "xapzpleg" is "apple," which is of length 5. *The LCS of two strings is equivalent to the optimal global alignment with the following scoring function*.

- score for match = 1
- score for mismatch = $-\infty$ (that is, we don't allow mismatch)
- score for insert/delete = 0

Using the same algorithm for the global alignment, *LCS can be computed in* ***O(nm) time***.

2. *Hamming Distance:* *The Hamming distance is the number of positions in two strings of equal length for which the corresponding symbols are different*. For example, the Hamming distance between two strings "toned" and "roses" is 3. It is equivalent to the optimal global alignment with the following scoring function.

- score for match = 1
- score for mismatch = 0
- score for indel = $-\infty$ (that is, we don't allow any indel)

Since the number of indels allowed is d = 0, the Hamming distance of two length-n strings can be computed in ***O((2d+1)n) = O(n) time and space***.

2.3 Local Alignment

Global alignment is used to align ***entire sequences***. Sometimes we are interested in *finding the* ***most similar substring*** *pair of the two input sequences*. Such a problem is called the *local alignment problem*. Precisely, given two strings S[1..n] and T[1..m], a local alignment is a pair of substrings *A* of S and *B* of T with the *highest alignment score*.

By ***brute force***, the local alignment can be computed using the following three steps.

1: *Find* ***all substrings*** *A of S and B of T*; *

2: *Compute the* ***global alignment*** *of A and B*;

3: *Return the substring pair (A, B) with the* ***highest score***.

Since there are $\frac{n}{2}$ choices for A and $\frac{m}{2}$ choices for B, the time complexity of this method is $O(\frac{n}{2}\frac{m}{2}nm) = O(n^3m^3)$. The algorithm is too slow.

* For example, Find all substrings *ACGT*

A, AC, ACG, ACGT

C, CG, CGT

G, GT

T

The total is n + (n-1) + (n-2) ...1 i.e. sum of n elements which is $\frac{n(n+1)}{2} = \frac{n}{2}$.

In 1981, ***Smith and Waterman*** proposed a better solution to the ***local alignment problem***. *Similar to the global alignment, the* ***Smith-Waterman algorithm*** *computes the optimal local alignment using* ***dynamic programming***.

Consider two strings S[1..n] and T[1..m]. We define *V(i, j) to be the maximum score of the global alignment of A and B over all substrings A of S end at i and all substrings B of T end at j, where $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant m$.* ***Note that we assume an empty string is also a substring of S ends at i and a substring of T ends at j***. *By definition, the score of the optimal local alignment is* $\max_{1\leqslant i\leqslant n, 1\leqslant j\leqslant m} V(i, j)$.

We devise the recursive formula for V(i, j) depending on two cases: (1) either i = 0 or j = 0 and (2) both i > 0 and j > 0.

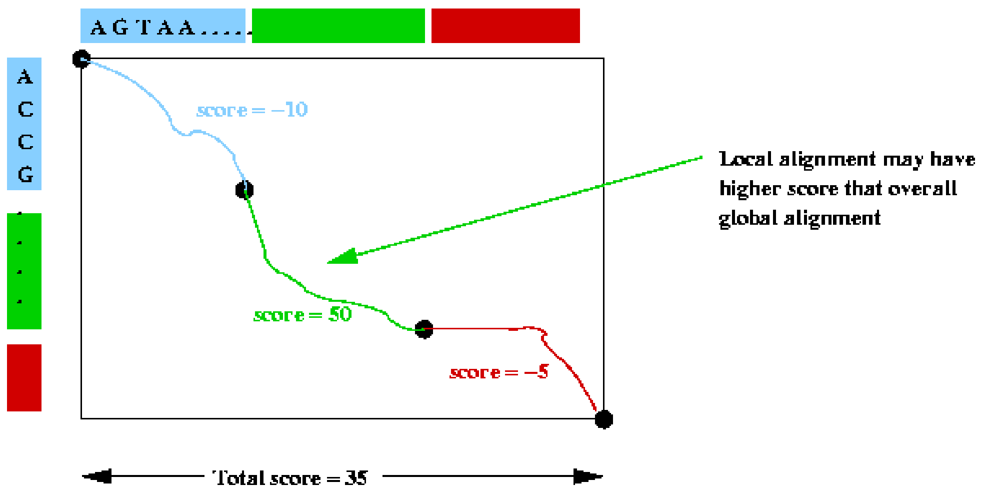
For case (1), when either i = 0 or j = 0, the best alignment aligns the empty strings of S and T . Hence, we have the following equations.

$$V(i,0) = 0 \text{ for } 0 \leq i \leq n$$
$$V(0,j) = 0 \text{ for } 0 \leq j \leq m$$

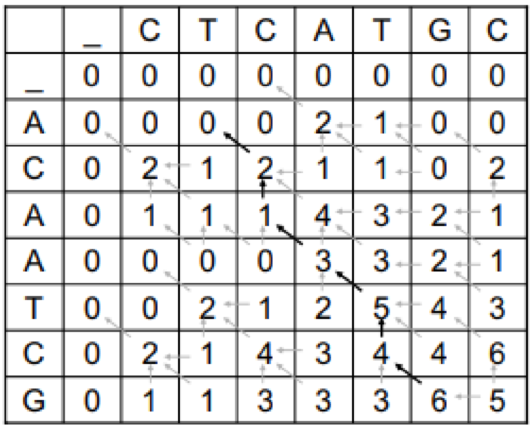
For case (2), when both i > 0 and j > 0, there are two scenarios. *The first scenario is that the best alignment aligns empty strings of S and T . In this case, V (i, j) = 0. For another scenario, within the best alignment between some substring of S ends at i and some substring of T ends at j, the last pair of aligned characters should be either match/mismatch, delete, or insert. To get the optimal score, we choose the maximum value among 0 and these three cases*. Thus, we get the following recurrence relation:

$$V(i,j) = \max \begin{cases} 0 & \mathit{align\ empty\ strings} \\ V(i-1,j-1) + \delta(S[i],T[j]) & \mathit{match/mismatch} \\ V(i-1,j) + \delta(S[i],\_) & \mathit{delete} \\ V(i,j-1) + \delta(\_,T[j]) & \mathit{insert} \end{cases}$$

The optimal local alignment score is max$_{i,j}$V(i, j). Smith and Waterman proposed that this score can be computed by filling in the table V row by row using the above recursive equations. Below is an example for further intuition of local alignment.


AGTAA.....
A
C
C
G
score = −10
score = 50
score = −5
Local alignment may have higher score that overall global alignment
Total score = 35

For example, consider *S = ACAATCG* and *T = CTCATGC*. Assume a match score is +2 and an insert/delete score is −1. Figure 2.8 shows the table V.



	_	C	T	C	A	T	G	C
_	0	0	0	0	0	0	0	0
A	0	0	0	0	2	1	0	0
C	0	2	1	2	1	1	0	2
A	0	1	1	1	4	3	2	1
A	0	0	0	0	3	3	2	1
T	0	0	2	1	2	5	4	3
C	0	2	1	4	3	4	4	6
G	0	1	1	3	3	3	6	5

Figure 2.8 The V table for local alignment between S = ACAATCG and T = CTCATGC.

The maximum score in table V is 6. Thus, the optimal local alignment score is 6. Through back-tracing from V(7, 6), we can recover the optimal alignment, which corresponds to the path in Figure 2.8:

```
CAATCG
C-AT-G
```

Below, we analyze the time and space complexities of the Smith-Waterman algorithm. The algorithm fills in nm entries in the table V (1..n, 1..m). Each entry can be computed in O(1) time. *The maximum local alignment score equals the entry with the maximum value*. So, ***both the time and space complexity are O(nm)***. Finally, *using additional O(n + m) time, we can recover the alignment by back-tracing*. *Similar to the optimal global alignment, the space complexity for computing the optimal local alignment can be reduced from O(mn) to O(n + m)*.

2.4 Semi-Global Alignment

Semi-global alignment is similar to global alignment, in the sense that it tries to align two sequences as a whole. The difference lies in the way it ***scores alignments****. Semi-global alignment ignores spaces at the beginning and/or the end of an alignment. In other words, semi-global alignment assigns no cost to spaces that appear before the first character and/or after the last character.*

Suppose we have two sequences S and T below:

```
S = ATCCGAACATCCAATCGAAGC
T = AGCATGCAAT
```

If we compute the *optimal global alignment*, we get:

```
ATCCGAACATCCAATCGAAGC
A---G--CATGCAAT------
```

Since the alignment has nine matches (score = 18), one mismatch (score = −1), and eleven deletions (score = −11), *the score of the alignment is* ***6****. However, such alignment might not be desired*.

One might wish to disregard flanking (i.e., starting or trailing) spaces. Such a feature is desirable, *for example, in aligning* ***an exon*** *to the* ***original gene sequence****. Spaces in front of the exon might be attributed to an untranslated region (UTR) or introns and should not be penalized*.

Coming back to our example, the *optimal semi-global alignment* for S and T would be:

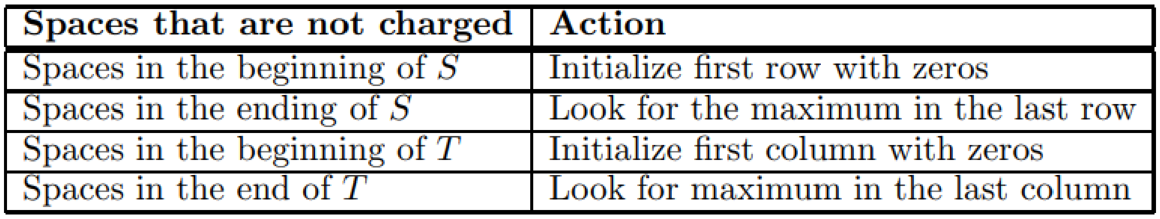
```
ATCCGAA-CATCCAATCGAAGC
------AGCATGCAAT------
```

The alignment has eight matches (score=16), one deletion (score=−1), and one mismatch(score=−1), which gives an alignment score of **14** instead of 6 in global alignment.

Another example of semi-global alignment is that *we may ignore the starting spaces of the first sequence and the trailing spaces of the second sequence*, as in the alignment below. *This type of alignment finds application in* ***sequence assembly****. Depending on the goodness of the alignment, we can deduce whether the two DNA fragments are overlapping or disjoint*.

```
---------ACCTCACGATCCGA
TCAACGATCACCGCA--------
```

Modifying the algorithm for global alignment to perform semi-global alignment is quite straightforward. The idea is that we give a ***zero score instead of a minus score*** *to those spaces in the beginning that are not charged. To ignore spaces at the end, we choose the maximum value in the last row or the last column*. Figure 2.9 summarizes the charging scheme.



Spaces that are not charged	Action
Spaces in the beginning of S	Initialize first row with zeros
Spaces in the ending of S	Look for the maximum in the last row
Spaces in the beginning of T	Initialize first column with zeros
Spaces in the end of T	Look for maximum in the last column

Figure 2.9 Charging scheme of spaces in semi-global alignment.

2.5 Gap Penalty

A ***gap*** *in an alignment is defined as a* ***maximal substring of contiguous spaces*** *in any sequence of alignment*. By definition, the following alignment has 2 gaps and 8 spaces.

```
A-CAACTCGCCTCC
AGCA-------TGC
```

Previous sections assumed the penalty for indel is proportional to the length of a gap. (*Such gap penalty model is known as the linear gap penalty model*.) *This assumption may not be valid in the biology domain. For example,* ***mutation*** *may cause* ***insertion/deletion of a substring*** *instead of a single base. This kind of mutation may be as likely as the insertion/deletion of a single base. Another example is related to mRNA. Recall that mRNA misses the introns during splicing. When aligning mRNA with its gene, the penalty should not be proportional to the length of the gaps*. Hence, it is natural not to impose a penalty that is strictly proportional to the length of the gap.

2.5.1 General Gap Penalty Model

If we define the ***penalty of a gap of length q as g(q)***, then we can align S[1..n] and T[1..m] using the following dynamic programming algorithm. Let V(i, j) be the global alignment score between S[1..i] and T[1..j]. We derive the recursive formula for V depending on two cases: (1) i = 0 or j = 0 and (2) i > 0 and j > 0.

When i = 0 or j = 0, we either insert or delete one gap. Hence, we have the following equations.

$$\begin{aligned}V(0,0) &= 0\\ V(0,j) &= -g(j)\\ V(i,0) &= -g(i)\end{aligned}$$

When i > 0 and j > 0, V(i, j) can be computed by the following recursive equation.

$$V(i,j) = \max\begin{cases} V(i-1,j-1) + \delta(S[i],T[j]) & \text{match/mismatch}\\ \max_{0\le k\le j-1}\{V(i,k) - g(j-k)\} & \text{Insert } T[k+1..j]\\ \max_{0\le k\le i-1}\{V(k,j) - g(i-k)\} & \text{Delete } S[k+1..i]\end{cases}$$

To compute the optimal global alignment score under the *general gap penalty*, we just fill in the table V row by row. *Then, the optimal alignment score is stored in V(n, m)*. The optimal alignment can be recovered by back-tracing the dynamic programming table V.

Below, we analyze the time and space complexities. *We need to fill in nm entries in the table V(1..n, 1..m)*. ***Each entry can be computed in O(n + m) time according to the recursive equation***. *The global alignment of two strings S[1..n] and T[1..m] can be computed in* ***O(nm(n+m)) time*** *and O(nm) space*.

Similarly, for computing *local or semi-global alignment*, we can also modify the algorithms to handle the *general gap penalty*.

2.5.2 Affine Gap Penalty Model

Although the general gap penalty model is flexible, computing alignment under the general gap penalty model is inefficient. Here, we study *a simple gap penalty model, called the* ***affine gap model, which is used extensively in the biology domain***.

Under the affine gap model, the penalty of a gap equals the sum of two parts: ***(1) a penalty h for initiating the gap*** *and* ***(2) a penalty s depending on the length of the gap***. Thus, the penalty for a *gap of length q* is:

$$g(q) = h + qs$$

Note that, when h = 0, the affine gap penalty is reduced to the uniform gap penalty.

Given two strings S[1..n] and T[1..m], their global alignment under the affine gap penalty can be computed as efficiently as that under the uniform gap penalty (that is, h = 0). *The idea is again to use dynamic programming*. *Moreover, **we require four dynamic programming tables instead of one***. *The first table is table V where V(i, j) is defined as the global alignment score between S[1..i] and T[1..j]. The other three tables are defined depending on the last pair of aligned characters.*

- ***G(i, j)***: the global alignment score between S[1...i] and T [1...j] with ***S[i] matches T[j]***
- ***F(i, j)***: the global alignment score between S[1...i] and T [1...j] with ***S[i] matches a space***
- ***E(i, j)***: the global alignment score between S[1...i] and T [1...j] with ***T[j] matches a space***

The global alignment score between S[1..n] and [1..m] is V(n, m). Below, we devise the recursive formula for the tables V, G, F, and E depending on whether *(1) i = 0 or j = 0* and *(2) i ≠ 0 and j ≠ 0*.

When i = 0 or j = 0, the basis for the recurrence equations are as follows:

$$\begin{aligned}
&V(0,0) = 0\\
&V(i,0) = F(i,0) = -g(i) = -h - is\\
&V(0,j) = E(0,j) = -g(j) = -h - js\\
&E(i,0) = F(0,j) = -\infty\\
&G(i,0) = G(0,j) = -\infty
\end{aligned}$$

When i > 0 and j > 0, the recurrence of V(i, j), G(i, j), E(i, j), and F(i, j) are defined as follows:

$$\begin{aligned}
&V(i,j) = \max\{G(i,j), F(i,j), E(i,j)\}\\
&G(i,j) = V(i-1,j-1) + \delta(S[i],T[j])\\
&F(i,j) = \max\{F(i-1,j) - s, V(i-1,j) - h - s\}\\
&E(i,j) = \max\{E(i,j-1) - s, V(i,j-1) - h - s\}
\end{aligned}$$

The recurrences for V and G are straightforward. *For F(i, j), it is the score of the global alignment of S[1..i] and T[1..j] with **S[i] matches a space***. There are two cases:

- ***S[i − 1] matches with a space***.

In this case, we need to add the ***space penalty****, that is, F(i, j) = F(i – 1, j)* ***– s****.*

- ***S[i – 1] matches with T[j]****.*

  In this case, we need to add the ***space penalty*** *and the* ***gap initiating penalty****, that is, F(i, j) = V(i – 1, j)* ***– h – s****.*

Then, by taking the maximum value of the two cases, we have F(i, j) = max{F(i−1, j)−s, V (i−1, j)−h−s}. Similarly, we can derive the recurrence for E.

To compute the optimal alignment of S[1..n] and T[1..m], we need to fill in the four tables according to the algorithm in Figure 2.10. ***The optimal alignment score is stored in V(n, m)****.* ***The optimal alignment can be recovered by back-tracing on the four tables****.*

```
Algorithm Affine_gap($S[1..n], T[1..m]$)
Require: Two sequences $S[1..n]$ and $T[1..m]$
Ensure: The optimal alignment score between $S[1..n]$ and $T[1..m]$.
 1: Compute $E(i,0), F(i,0), G(i,0), V(i,0)$ for $i = 1, \ldots, n$
 2: Compute $E(0,j), F(0,j), G(0,j), V(0,j)$ for $j = 1, \ldots, m$
 3: for $i = 1$ to $n$ do
 4:    for $j = 1$ to $m$ do
 5:       $G(i,j) = V(i-1, j-1) + \delta(S[i], T[j])$
 6:       $F(i,j) = \max\{F(i-1,j) - s, V(i-1,j) - h - s\}$
 7:       $E(i,j) = \max\{E(i,j-1) - s, V(i,j-1) - h - s\}$
 8:       $V(i,j) = \max\{G(i,j), F(i,j), E(i,j)\}$
 9:    end for
10: end for
11: Report $V(n,m)$
```

Below, we analyze the time complexity and the space complexity. *We maintain four n × m matrices E, F, G, and V . Hence, the* ***space complexity is O(nm)****. Since the 4 tables have 4nm entries where each entry can be computed in O(1) time, the* ***time complexity is O(nm)****.*

2.6 Scoring Function

To measure the similarity between two sequences, we need to define the scoring function δ(x, y) for every pair of bases x and y.

2.6.1 Scoring Function for DNA

For DNA, since there are only 4 nucleotides, the score function is simpler. *In BLAST, it simply gives a positive score for a match and a negative score for mismatch. The NCBI-BLASTN set the match score and the mismatch score to +2 and -1, respectively. The WU-BLASTN and FastA set the match score and the mismatch score to +5 and -4, respectively. The* ***score function for NCBI-BLASTN*** *is better for detecting* ***homologous alignment*** *with* ***95% identity****. For* ***WU-BLASTN and FastA****, their* ***scoring function*** *is more suitable to detect* ***homologous alignment*** *with* ***65% identity****.*

In the evolution of real sequences, ***transitions are observed more often than transversions****. Thus, some people use the* ***transition transversion matrix****.* In this model, the four nucleotides are separated into two groups: purines (A and G) and pyrimidines (C and T). *It gives a* ***mild penalty (-1)*** *for replacing* ***purine by purine*** *or* ***pyrimidine by pyrimidine****. For substitution between* ***purine and pyrimidine****, it gives a* ***bigger penalty (-5)****.* ***Match score is +1****.*

	A	**G**	**C**	**T**
A	+1	-1	-5	-5
G	-1	+1	-5	-5
C	-5	-5	+1	-1
T	-5	−5	-1	+1

2.6.2 Scoring Function for Protein

For protein, there are two approaches to assign a similarity score. *The first approach assigns a similarity score based on the* ***chemical or physical properties*** *of amino acids, including* ***hydrophobicity, charge, electronegative, and size****. The basic assumption is that an amino acid is more likely to be substituted by another if they share a similar property. For example, we give a higher score for substituting a non-polar amino acid by another non-polar amino acid.*

Another approach assigns a similarity score purely based on ***statistics****. The two popular scoring functions* ***PAM*** *and* ***BLOSUM*** *belong to this category. The idea is to count the* ***observed substitution frequency*** *and compare it with the* ***expected substitution frequency****. Two residues a and b are expected to be similar if they have a* ***big log-odds score***

$$\log \frac{O_{a,b}}{E_{a,b}}$$

where $O_{a,b}$ and $E_{a,b}$ are the observed substitution frequency and the expected substitution frequency, respectively, between amino acid residues a and b. ***With the assumption that the aligned residue pairs are statistically independent, the alignment score of an aligned sequence equals the sum of individual log-odds scores****.*

2.6.2.1 Point Accepted Mutation (PAM) Score Matrix

Dayhoff developed the PAM score matrix in 1978. Actually, it is a family of ***score matrices*** *which are suitable for aligning* ***protein sequences*** *of different levels of divergence. The* ***divergence*** *is measured in terms of* ***point accepted mutation (PAM)****. A point mutation refers to a substitution of one residue by another. A* ***point accepted mutation (PAM)*** *is a point mutation which* ***does not change the protein's function or is not fatal****. Two sequences S_1 and S_2 are said to be* ***1 PAM diverged*** *if S_1 can be converted to S_2 with* ***an average of 1 accepted point mutation per 100 residues****. For every i, Dayhoff constructed the* ***PAM-i matrix*** *which is suitable to compare sequences which are* ***PAM-i diverged****.*

To obtain a ***PAM-1 matrix****, Dayhoff collected a set of* ***ungapped alignments*** *which is constructed from high similarity amino acid sequences (usually > 85%). Then, a* ***phylogenetic tree*** *is constructed to identify the set of* ***mutations****.* Figure 2.15 shows an example of ungapped alignment and the corresponding phylogenetic tree.

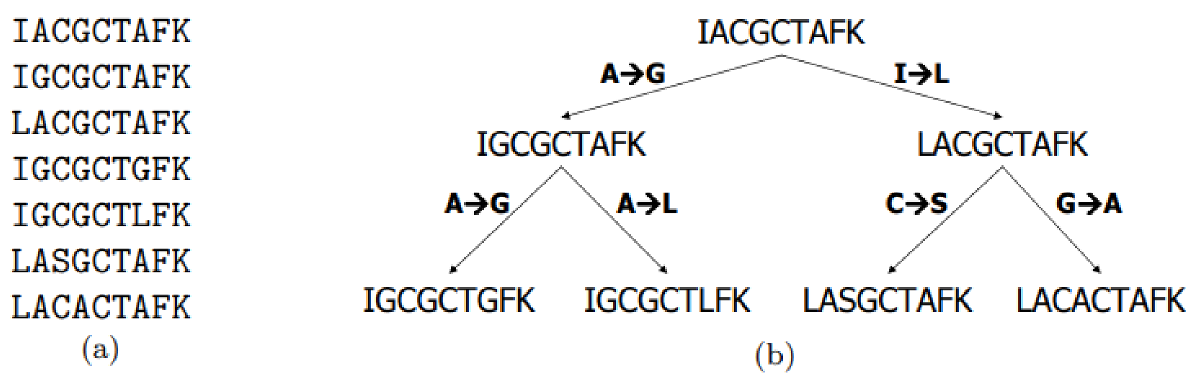


Figure 2.15 The left figure is an *ungapped alignment of seven amino acid sequences* and the right figure is the *corresponding phylogenetic tree of those seven sequences*.

Based on the ***phylogenetic tree****, $O_{a,b}$ and $E_{a,b}$ can be computed for any two amino acid residues a and b as follows. Since PAM-1 assumes 1 mutation per 100 residues, $O_{a,a}$ is set to be 99/100 for any residue a. For any residues a ≠ b, $O_{a,b}$ is set to be $F_{a,b}/(100 \sum_{x,y\in A} F_{x,y})$ where* ***A is the set of amino acids*** *and* ***$F_{x,y}$ is the frequency of substitution between x and y in the phylogenetic tree***. ***$E_{a,b}$ is set to be $f_a f_b$ where f_a is the number of residue a divided by the total number of residues***. *Given $O_{a,b}$ and $E_{a,b}$, the* ***similarity score δ(a, b)*** *equals* ***log($O_{a,b}/E_{a,b}$)***.

$$O_{a,a} = \frac{99}{100}$$

$$O_{a,b} = \frac{F_{a,b}}{100 \sum_{x,y\in A} F_{x,y}} \quad \Rightarrow \quad \delta(a,b) = log\frac{O_{a,b}}{E_{a,b}}$$

$$E_{a,b} = f_a \cdot f_b$$

For example, in Figure 2.15, the similarity score of **(A, G) = ?**

$$O_{A,G} = \frac{F_{A,G}}{100 \sum_{x,y\in A} F_{x,y}} = \frac{F_{A,G}}{100(F_{A,G}+F_{I,L}+F_{A,G}+F_{A,L}+F_{C,S}+F_{G,A})} = \frac{3}{100(6\times 2)} = 0.0025$$

$$E_{A,G} = f_A \cdot f_G = \frac{10}{63} \times \frac{10}{63} = 0.0252$$

$$\delta(A, G) = log\frac{O_{A,G}}{E_{A,G}} = log\frac{0.0025}{0.0252} = -1.0034$$

PAM-1 matrix for Figure 2.15:

	A	C	F	G	I	K	L	S	T
A	**1.5942**	**-∞**	**-∞**	**-1.0034**	**-∞**	**-∞**	**-1.0809**	**-∞**	**-∞**
C	**-∞**	**1.3672**	**-∞**	**-∞**	**-∞**	**-∞**	**-∞**	**-0.5860**	**-∞**
F	**-∞**	**-∞**	**1.9057**	**-∞**	**-∞**	**-∞**	**-∞**	**-∞**	**-∞**
G	**-1.0034**	**-∞**	**-∞**		**-∞**	**-∞**	**-∞**	**-∞**	**-∞**
I	**-∞**	**-∞**	**-∞**	**-∞**		**-∞**	**-0.6829**	**-∞**	**-∞**
K	**-∞**	**-∞**	**-∞**	**-∞**	**-∞**		**-∞**	**-∞**	**-∞**
L	**-1.0809**	**-∞**	**-∞**	**-∞**	**-0.6829**	**-∞**		**-∞**	**-∞**
S	**-∞**	**-0.5860**	**-∞**	**-∞**	**-∞**	**-∞**	**-∞**		**-∞**
T	**-∞**	**-∞**	**-∞**	**-∞**	**-∞**	**-∞**	**-∞**	**-∞**	

To obtain the PAM-i matrix, Dayhoff generated it by extrapolating the PAM-1 matrix. *Let $M_{a,b}$ be the probability that a is mutated to b, which equals $O_{a,b}/f_a$. Let M^i be the matrix formed by multiplying M i's times. Then, $M^i_{a,b}$ is the probability that a is mutated to b after i steps. Therefore, the (a, b) entry of the PAM-i matrix is*

$$log\frac{O_{a,b}}{E_{a,b}} = log\frac{f_a M^i_{a,b}}{f_a f_b} = log\frac{M^i_{a,b}}{f_b}$$

Which PAM matrix should we use when we align protein sequences? *In general, to align closely related protein sequences, we should use the PAM-i matrix where i is small. To align distant related protein sequences, we should use the PAM-i matrix where i is large.*

2.6.2.2 BLOSUM (BLOck SUbstitution Matrix)

PAM did not work well for aligning evolutionarily divergent sequences since the PAM matrix is generated by extrapolation. Henikoff proposed BLOSUM, which is a scoring matrix constructed directly from the observed alignments (instead of extrapolation).

*The **BLOSUM matrix** is constructed using **multiple alignments** of nonredundant groups of protein families. Using PROTOMAT, **blocks of ungapped local alignments** are derived. Each **block** represents an **ungapped conserved region** of a protein family. For any two amino acid residues a and b, the **expected frequency $E_{a,b}$ is set to be $f_a f_b$ where f_a is the frequency of a in the blocks**. The **observed frequency $O_{a,b}$ is set to be the (a, b) pairs among all aligned residue pairs in the block**. BLOSUM defines the **similarity score** as the **log-odds** score*

$$\delta(a,b) = \frac{1}{\lambda} \ln \frac{O_{a,b}}{f_a f_b}$$

*where λ is some **normalization constant**.*

For example, in Figure 2.15, the similarity score of **(A, G) = ?** (Suppose λ = 0.347)

$$O_{A,G} = \frac{23\ (A,G)\ pairs}{9\frac{7}{2}} = \frac{23}{189} = 0.1216$$

$$E_{A,G} = f_A \cdot f_G = \frac{9}{63} \times \frac{10}{63} = 0.0227$$

$$\delta(A,G) = \frac{1}{\lambda}\ln\frac{O_{A,G}}{E_{A,G}} = \frac{1}{0.347}\ln\frac{0.1216}{0.0227} = 4.84$$

To reduce the contribution of closely related protein sequences to the residue frequencies and the residue pair frequencies, *similar sequences are merged within blocks. The **BLOSUM p matrix** is created by **merging sequences with no less than p% similarity***. For example, consider the following set of sequences.

```
AVAAA
AVAAA
AVAAA
AVLAA
VVAAL
```

The first four sequences have at least 80% similarity. The similarity of the last sequence with the other four sequences is less than 62%. *For BLOSUM 62, we merge the first four sequences and we get sequences*

$$\begin{gathered} AV[A_{0.75}L_{0.25}]AA \\ VVAAL \end{gathered}$$

▯

AVAAA	***AVLAA***
VVAAL	***VVAAL***

The sequences consist of 10 residues and 5 residue pairs. We have $f_A = 5.75/10$, $O_{A,V} = 1/5$, and $O_{A,L} = (0.25 + 1)/5$.

$$f_A = \frac{0.75 \times 6}{10} + \frac{0.25 \times 5}{10} = \frac{5.75}{10}$$

$$O_{A,V} = \frac{0.75 \times 1}{5} + \frac{0.25 \times 1}{5} = \frac{1}{5}$$

$$O_{A,L} = \frac{0.75 \times 1}{5} + \frac{0.25 \times 2}{5} = \frac{1 + 0.25}{5}$$

To align distant related sequences, we use BLOSUM p for small p; otherwise, we use BLOSUM p for large p. The relationship between BLOSUM and PAM is as follows.

- BLOSUM 80 $\approx$ PAM 1
- BLOSUM 62 $\approx$ PAM 120
- BLOSUM 45 $\approx$ PAM 250

BLOSUM 62 is the default matrix for BLAST 2.0.
