

Lecture 2: Sequence Comparison - August 29, 2003

Lecturer: Wing-Kin Sung

Scribe: Li Haixia, Eddie Loh Yong Hwee, Yap Choon Kong, Wang Jiren

2.1 The goal of sequence similarity

The importance of sequence comparison can be traced back to a few earliest researches. The first research on sequence comparison starting in 1983 where Doolittle et al[1] tried to search for all platelet-derived growth factor (PDGF) to built up a database for it. After accumulating some sequences, they discovered that some of the sequences, v-sis oncogene, are very similar to the PDGF. At that time the function of v-sis oncogene is still unknown. Based on the similarity of two sequences, they suggested that the function of both v-sis oncogene and PDGF are the same. Indeed, scientist later found that these two gene share similar function. Another research done by Riordan et al[2] also results in the same conclusion. They made use of multiple sequence alignment to understand the cystic fibrosis gene. Since all the compared sequences are similar, they share similar function too.

So, there is a well-known conjecture in the industry of biology. Given two sequences, either in DNAs, RNAs or Proteins, if they are highly similar, we can infer that they share similar function or similar 3D structure. Nevertheless, the reverse does not hold. One such example is tRNAs where their structure are the same but do not exhibit much similarity in their sequences. Consequently, researchers of bioinformatics often need to compare the similarity between two biological sequences. A wide variety of applications in sequence comparison have been probed for quite a long time. Some typical applications are given below:

- Inferring biological functions of gene.
When a gene looks similar to some gene with known function, we can conjecture that both genes have similar function.
- Finding the evolution distance between two species.
We know that evolution modifies the genomes of different species due to mutation. By measuring the similarity of their genome, we can know their evolution distance. Such knowledge is especially important for biology scientists.

- Helping genome assembly.
For instance, human genome project can reconstruct the whole genome on the basis of the overlapping information of large quantities of short DNA pieces. The overlapping information is extracted using sequence comparison.
- There are many other applications of sequence similarity.
Examples include comparing DNA sequences in Databases, and comparing two or more sequences for similarities and searching databases for related sequences and subsequences.

2.2 Global alignment

2.2.1 String edit and string alignment

Even before the computational biology, something similar to sequence alignment called string edit has already been studied extensively. String edit deals with problems on transforming a string to another string with the minimal number of operations. The allowed operations are (I) Replace a letter with another letter, (II) Insert a letter, and (III) Delete a letter. So given two strings, $A = \textit{interestings}$ and $B = \textit{bioinformatics}$, the minimal number of edit operations needed to transform A to B is 9 and is shown in Figure 2.1.

```

A = interestings      _i__nterestings
B = bioinformatics    bioinformatic_s
                      101101101100110

```

Figure 2.1: Example on converting string

Instead of minimizing the number of edit operations, we can associate a cost function to the operations and minimize the total cost. Such cost is called edit distance. For instance we can assign a cost of 2 for replacement and a cost of 1 for either insertion or deletion. In the example of Figure 2.1, we assume all operations to have cost 1 and thus result in edit distance of 9.

In computational biology, the concept of string alignment is somewhat similar to string edit. While the string edit problem measures the minimum *cost* of transforming one string to another, the string alignment problem measures the maximum *similarity* or *goodness* when one string is compared to another. When we negate the values in the cost function, it becomes a similarity function, therefore string edit and string alignment are in fact dual problems. Only string

alignment will be studied here.

Given two strings, $S = ACAATCC$ and $T = AGCATGC$, one way to compare them is to compute their alignment. Below is a possible alignment of the two strings. The special symbol “-” is called a space. More precisely, the alignment of the two strings is obtained by introducing some spaces, either into or at the ends of S and T , so the length of the sequences will be the same, and then placing the two resulting sequences one above the other so that every character or space in one of the sequences is paired to a unique character or a unique space in the other sequence.

The comparison done is pretty much similar to those operations in string edit. If the paired characters are the same, we call it a match, otherwise, we call a mismatch. If a space in the first sequence is paired to a character of the second sequence, it is an insert. If a character in the first sequence is paired to a space in the second sequence, it is called a delete. For the following example, the alignment contains 8 pairs. 5 pairs are match. 1 pair is mismatch. 1 pair is delete. 1 pair is insert.

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

The goodness of an alignment is defined by $\sum_i \delta(S[i], T[i])$, where $\delta(x, y)$ is a similarity function between x and y , each is a single character or a single space. Figure 2.2 shows a similarity function, where $\delta(x, y) = 2, -1, -1, -1$ for match, mismatch, delete and insert respectively.

For the previous two sequences. The similarity score of their alignment is $7(2*5 - 1 - 1 - 1 = 7)$. We can check that this alignment has the maximum score. Such alignment is called optimal alignment. Other alignments of the two strings get lower scores. **String alignment problem** tries to find the alignment with the maximum similarity score. This problem is also called **global alignment problem**.

	-	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 2.2: Example of similarity function

2.2.2 Needleman-Wunsch algorithm

This section discusses how to compute the optimal alignment of two strings. Consider two strings: $S[1..n]$ and $T[1..m]$. Define $V(i, j)$ be the score of the optimal alignment between $S[1..i]$ and $T[1..j]$.

When $i = j = 0$, the alignment problem is trivial and $V(i, j)$ is assigned to be 0. When $i = 0$ (or $j = 0$), the resultant alignment would consist of all gaps for the empty string. The basis equations can thus be obtained.

Basis:

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

When both $i > 0$ and $j > 0$, for the alignment between $S[1..i]$ and $T[1..j]$, the last pair of alignment should be match, mismatch, delete or insert, as shown below.

xxx...xx	xxx ...xx	xxx ...x_
xxx...yy	yyy ...y_	yyy...yy
match/mismatch	delete	insert

To get the optimal score, we choose the maximum value among these three cases. Thus, we have the following recurrence.

Recurrence: (for $i > 0$ and $j > 0$)

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

Figure 2.3 shows the V table of the two strings $S = AGCATGC$ and $T = ACAATCC$. We fill in the table row by row based on the above recursive equations. Let's look at row 2, column 2. The value 2 is obtained by the following: $\max(0 + 2, -1 - 1, -1 - 1) = \max(2, -2, -2) = 2$. Since this value of 2 is obtained from (1,1) (ie. row 1, column 1), we will draw an arrow from the current position (2,2) to (1,1). Let's calculate another one, e.g., row 2 column 3. We choose the maximum value to be 1 where $\max(-1 + 2, 2 - 1, -2 - 1) = \max(1, 1, -3) = 1$. Note that in this case, there are two ways to get the maximum value (from (1,2) or (2,2)). Thus we draw arrows to these positions too from our current position

		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 2.3: The dynamic programming table for $S = AGCATGC$ and $T = ACAATCC$

of (2,3).

In Figure 2.3, we draw arrows to indicate all the ways to get the maximum values. A diagonal arrow would mean a match/mismatch, while a horizontal arrow would mean an addition, and a vertical arrow a deletion. The max alignment for global alignment is always starting from the right bottom corner, which is entry for last row and last column. The score that we obtained is 7. Then, we trace back along the arrows to the value for the first pair of the two sequences, thus finding the optimal alignment. Sometimes there are more than one path to go from last entry (last row last column) to first entry (first row first column). In this case, all pathways are taken as optimal alignment and hence it is not necessary to have a single optimal alignment only between two sequences.

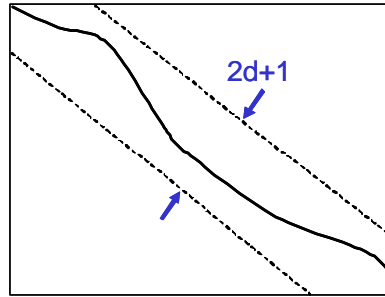
Analysis

- **Space:** We need to fill in all entries in the $n \times m$ table, so space complexity = $O(nm)$
- **Time:** Each entries can be computed in $O(1)$ time. Time complexity = $O(nm)$

2.2.3 Problem on speed

- Aho, Hirschberg, Ullman (1976): If we can only compare whether two symbols are equal or not, the string alignment problem can be solved in $\Omega(nm)$ time.
- Hirschberg (1978): If symbols are ordered and can be compared, the string alignment problem can be solved in $\Omega(n \log n)$ time.
- Masek and Paterson (1980)—Based on Four-Russian's paradigm, the string alignment problem can be solved in $O(nm/\log n)$ time. Today, this is the best algorithm that we can get.

Let d be the total number of inserts and deletes. Note that $0 \leq d \leq n + m$. If d is smaller than $n + m$, we can get a better solution. So if the total number of gaps d for both sequences is known, we claim that we only need only to fill in the region inside the $(2d + 1)$ band only, (see Figure 2.4).

Figure 2.4: $2d + 1$ Band

The reason being is like this. In order to create gaps, we must either move horizontally or vertically. Thus knowing the total number of gaps d in advance, we can limit our move, either horizontally to left and right, or vertically up and down, no more than d steps. The lower and upper triangle beside the $(2d + 1)$ band in the V table require more than d 's deletes or inserts. Thus, we don't need to fill in the lower and upper triangle in the V table. The area of the $(2d + 1)$ band in the V table is $(nm - (n - d)(m - d) = md + nd - d^2)$. The time for filling in every entry inside the band is $O(1)$. So the total Time is $O((n + m)d)$. The V table of Figure 2.3 is redo using this method and the result is shown in Figure 2.5.

		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 2.5: $2d+1$ Band Example

2.2.4 Problem on space

Note that the dynamic programming requires $O(mn)$ space. When we compare two very long sequences, space would be a problem. Can we solve the string alignment problem in linear space?

2.2.4.1 Finding optimal alignment score of two strings in $O(\min(n, m))$ space

If we just want to get the **alignment score** in the previous example, observe that the table can be filled in row by row. When we fill in a row, we only need the V -values of the current row and the previous row.

	_	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 2.6: V table using $O(\min(n, m))$ space

As shown in Figure 2.6, row 4 is depending on row 3 only in filling its entries. So rows 1 and 2 are not needed and the space can be freed. In general, we only need to keep two rows. If n is smaller than m , we could fill the table row by row. If m is smaller, we could fill column by column. Thus, if we did not need to backtrack, space complexity = $O(\min(n, m))$.

2.2.4.2 Finding optimal alignment of two strings in $O(n + m)$ space

In fact, we can deduce the optimal alignment of two strings in $O(n + m)$ space[3]. based on the cost-only algorithm in Section 2.2.4.1.

The main idea is as follow, see Figure 2.7. Based on the cost-only algorithm, find the mid-point of the alignment. Then the problem can be divided into two halves. Recursively solving the problem in the two halves will result in the final optimal alignment.

The central idea behind this is that the optimal alignment of $(S[1..n], T[1..m])$ is the union of (1) the optimal alignment of $(S[1..\frac{n}{2}], T[1..j])$ and (2) the optimal alignment of $(S[\frac{n}{2} + 1..n], T[j + 1..m])$ for some j . It can be written into the equation in this form.

$$V(S[1..n], T[1..m]) = \max_{1 \leq j \leq m} \{V(S[1..\frac{n}{2}], T[1..j]) + V(S[\frac{n}{2} + 1..n], T[j + 1..m])\}$$

The entry $(\frac{n}{2}, j)$ is called the mid-point. The following algorithm tells us how to compute the mid-point using the cost-only algorithm in Section 2.2.4.1.

The mid-point algorithm is as follows:

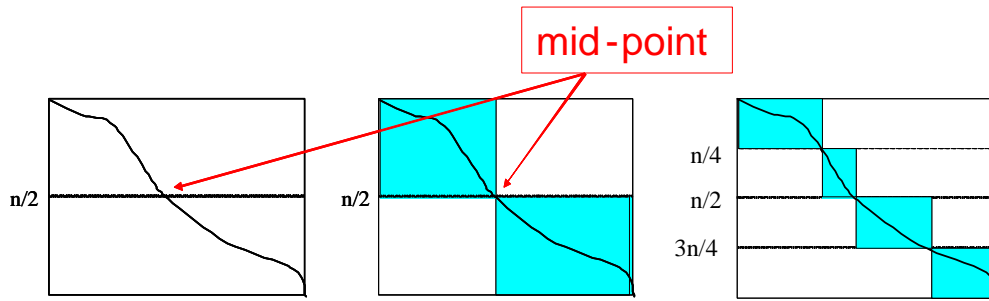


Figure 2.7: Mid-point Example

1. Do cost-only dynamic programming for the first half. Then, we find $V(S[1..\frac{n}{2}], T[1..j])$ for all j .
2. Do cost-only dynamic programming for the reverse of the second half. Then, we find $V(S[\frac{n}{2} + 1..n], T[j + 1..m])$ for all j
3. Determine j which maximizes the sum!

	_	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 2.8: Mid-point Example

Figure 2.8 shows an example of how to do this. We first filling up the V table as shown, which cover the first two steps of mid-point algorithm. Then we sum up the score of the middle two row diagonally and get the maximum score. Here the maximum score that we can get is 7, which is by adding 4 to 3 and j can thus be determined.

If we divide the problem into two halves based on the mid-point and recursively deduce the alignments for the two halves, we can reduce the space complexity to $O(n + m)$. The overall algorithm is as follows.

Algorithm: Alignment of $(S[i_1..i_2], T[j_1..j_2])$

1. Let $mid = (i_1 + i_2)/2$

2. Find the mid-point (mid, j) using the mid-point algorithm
3. Deduce the alignment based on $\text{Alignment}(S[i_1..mid], T[j_1..j])$ and $\text{Alignment}(S[mid+1..i_2], T[j+1..j_2])$

Analysis

- **Time:** Time for step 1 is $O(\frac{n}{2}m)$. Time for step 2 is also $O(\frac{n}{2}m)$. Time for step 3 is m . So the time complexity of the first cycle is the sum of the three steps, $O(nm)$. Let's define $T(m, n)$ as the time for the whole alignment.
 $T(n, m) = \text{time for finding mid-point} + \text{time for solving the two subproblems} = O(nm) + T(\frac{n}{2}, j) + T(\frac{n}{2}, m - j)$. By solving the recursive equation, the time complexity $= T(n, m) = O(nm)$.
- **Space:** Working memory for finding mid-point takes $O(m)$ space. However we can free the working memory once we find the mid-point. Thus, in each recursive call, we only need to store the alignment path. Observe that the alignment subpaths are disjoint, the total space required is thus $O(m + n)$.

2.2.5 More on string alignment problem

Two special Cases:

- **Longest common subsequence (LCS):** Score for mismatch is negative infinity; score for insertion or deletion is 0; score for match is 1.
- **Hamming distance:** score for insert or delete is negative infinity Score for match is 1; score for mismatch is 0.

2.3 Local alignment

While global alignment is used to align the entire sequence, **local alignment** searches for regions of local similarity. A **local alignment** between s and t is an alignment between a substring of s and a substring of t . This section presents the algorithm to find the highest scoring local alignment between two sequences. In practice, local alignment method is very useful for scanning databases when we do not know that the sequences are similar over their entire lengths.

2.3.1 Brute-force algorithm

The mechanism of local alignment is a variation of the basic algorithm. Let $S[1..n]$ and $T[1..m]$ be two strings. Local alignment proceeds with the following steps.

- Find substrings(i.e., contiguous subsequences) A of S and B of T.

- Compute the similarity score of substring pair A and B.
- Select a pair of substring A' and B' , whose optimal (global) alignment has the maximum score over all pairs of such substrings A and B.

We can see that there are $\frac{n(n-1)}{2} + 1$ choices of substrings from S and $\frac{m(m-1)}{2} + 1$ choices of substring from T. The process of global alignment of A and B costs $O(nm)$. Hence, the total time complexity equals to $O(n^3m^3)$.

2.3.2 Smith-Waterman algorithm

Obviously, the above algorithm is too slow. In 1981, Smith and Waterman proposed a better solution toward the problem of local alignment. Before describing the algorithm, we give some definitions first.

Prefix X is a prefix of $S[1..n]$ if $X = S[1..k]$, where $1 \leq k \leq n$.

Suffix X is a suffix of $S[1..n]$ if $X = S[k..n]$, where $1 \leq k \leq n$.

For example, let $S=ATCCGGT$, then ATCC is a prefix of S and GGT is a suffix of S. Note that empty string is both prefix and suffix of S.

V(i,j) Given two strings S and T with $|S| = n$, $|T| = m$. $V(i,j)$ is defined as the maximum value of an optimal(global) alignment of A and B over all suffixes A of $S[1..i]$ and all suffixes B of $T[1..j]$, where $1 \leq i \leq n$ and $1 \leq j \leq m$.

Since the definition of local alignment allows us choose the best alignment score among any substring of S and T (without being restricted to having to take only the alignment of the full string), the value of any i, j which gives the maximum alignment score is the optimal local alignment for S and T. Smith-Waterman dynamic programming algorithm for optimal local alignment is quite similar to that of global alignment.

When i and/or $j = 0$, the optimal alignment would include an empty string (and suffix substring) and thus we assign the alignment score to be 0. This forms the basis of the recurrence equations and is represented below:

Basis:

$$V(0,0) = 0$$

$$V(i,0) = 0$$

$$V(0,j) = 0$$

	_	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	0	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 2.9: The V table for local alignment between $S = CTCATGC$ and $T = ACAATCG$

for any entry (i, j) , there is always the alignment between empty suffixes of $S[1..i]$ and $T[1..j]$, which has score 0. Therefore in local alignment, we add a fourth consideration (as compared to global alignment) when calculating $V(i, j)$. The resulting recurrence equations are represented below.

Recurrence: (for $i > 0$ and $j > 0$)

$$V(i, j) = \max \begin{cases} 0 & \text{Align empty strings} \\ V(i-1, j-1) + \delta(S[i], T[j]) & \text{Match/mismatch} \\ V(i-1, j) + \delta(S[i], \square) & \text{Delete} \\ V(i, j-1) + \delta(\square, T[j]) & \text{Insert} \end{cases}$$

2.3.3 Example

For example, let $S = CTCATGC$ and $T = ACAATCG$, score for match = +2, score for insert, delete and mismatch = -1. Smith-Waterman dynamic programming algorithm fills in the V table with values from top to bottom and left to right. Figure 2.9 shows the V table. The maximum score 6 is the score of the optimal local alignment of S and T . In this case, there are 2 possible optimal alignments.

The path in Figure 2.9 corresponds to the optimal alignment, which is:

C_ AT_ G
CAATCG

2.3.4 Time and Space Analysis

For the time analysis of Smith-Waterman algorithm, note that we need to fill in all entries in the $n \times m$ table, where each entry can be filled in $O(1)$ time. The

next step is to find the maximum value among all $n \times m$ entries. Totally, the time needed is $O(nm)$. For the space analysis, since we store the $n \times m$ table, the space required is $O(nm)$.

2.4 Semi-global alignment

In earlier sections, we have discussed two kinds of sequence alignments: global alignment and local alignment. There is another type of alignment known as *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 penalizes spaces at the beginning and end of the alignment. While global alignment does not differentiate between spaces that are sandwiched between two residues and spaces that precede or succeed a sequence, semi-global alignment imposes no penalty to the latter spaces. To put it more formally, semi-global alignment assigns no cost to spaces that appear before the first residue or after the last residue. One application for this alignment is detecting overlaps in short-gun sequencing. As endgaps do not correspond to insertion or deletions, they should not be penalized.

Let us begin by defining precisely what we mean by end spaces and why it might be better to let them be included for free in certain situations. End spaces are those that appear before the first or after the last character in a sequence. For instance, all the spaces in the second sequence in the alignment below are end spaces, while the single space in the first sequence is not an end space.

```
S = CAGCA-CTTGGATTCTCGG
T = ---CAGCGTGG-----
```

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

```
CAGCACTTGGATTCTCGG
CAGC-----G-T----GG
```

However, this is not the best alignment between these two sequences. In some alignments one might wish to disregard flanking (i.e. starting or trailing) spaces. Recall that one of the goals of sequence alignment is to deduce evolutionary relationship. Given two sequences, we can't say with 100% certainty that they stemmed from exactly the same DNA region, but rather coming around the same region. One of the sequences might have been padded (at the front and/or back) with residues that has nothing to do with the region of interest. In this case, having spaces flanking the other sequences should not be considered as a bad thing, and thus should not be penalized.

Such feature is desirable, for example in aligning an exon to the gene's original DNA sequence. Spaces in front of the exon might be attributed to 5'-UTR (Untranslated Region)¹ or introns and should not be penalized. This method is also used in locating genes in a prokaryotic genome.

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

```
CAGCA-CTTGGATTCTCGG
---CAGCGTGG-----
```

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

```
-----ACCTCACGATCCGA
TCAACGATCACCGA-----
```

Modifying the algorithm for global alignment to perform semi-global alignment is quite straightforward. Table 2.1 summarizes the necessary changes.

Spaces that are not charged for	Action
Spaces in the beginning of $S[1..n]$	Initialize first row with zeros
Spaces in the ending of $S[1..n]$	Look for the maximum in the last column
Spaces in the beginning of $T[1..m]$	Initialize first column with zeros
Spaces in the end of $T[1..m]$	Look for maximum in the last row

Table 2.1: Summary of end space charging in semi-global alignment.

2.5 Gap penalty models

In this notes, we shall define a gap as follows:

Definition 2.1 *A gap in an alignment is a maximal substring of contiguous spaces in either sequence of the alignment.*

By this definition, there are 2 gaps in the following alignment:

¹For more information on Untranslated Region, refer to <http://bighost.area.ba.cnr.it/BIG/UTRHome/>

```
A-CAACTCGCCTCC
AGCA-----CCTGC
```

In previous discussions, we had assumed that the penalty for an insertion or deletion is directly proportional to the length of a gap. This assumption may not be valid in certain applications. For example, large substrings of nucleotides may be inserted or deleted in a single instance, and they may be as likely to occur as insertions or deletions of a single nucleotide. Another example would be in the case of sequence alignment of mRNA with its genome, where we would definitely expect strings of gaps corresponding to the introns which have been deleted in the mRNA. Thus it may not be appropriate to simply assign penalties to these gaps that are directly proportional to its length. The following sections describe various methods for assigning gap penalties.

2.5.1 General gap penalty model

In general, if we define the penalty of a gap of length q as $g(q)$, then we can align $S[1..n]$ and $T[1..m]$ under the general gap penalty model using the following dynamic programming.

Let $V(i, j)$ be the global alignment score between $S[1..i]$ and $T[1..j]$.

When $i = j = 0$, we are aligning 2 empty strings. The global alignment score of $V(0, 0)$ of this trivial case is assigned as 0.

When $i = 0, i \neq j$, we are aligning $T[i..j]$ with an empty string S . The aligned sequence of S would thus be a gap of length j . Thus $V(0, j)$ is the penalty of a gap of length j

Similarly, when $j = 0, i \neq j$, we are aligning $S[i..j]$ with an empty string T . Thus $V(i, 0)$ is the penalty of a gap of length i .

The base cases can thus be represented with the following equations:

Base cases:

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

When $i > 0, j > 0$, we can apply the method of recurrence to obtain $V(i, j)$ by considering the 3 cases of Match/Mismatch, Insert or Delete.

1. $S : xx...xx\mathbf{i}$
 $T : yy...yy\mathbf{j}$

Alignment of $S[1..i]$ and $T[1..j]$ where characters $S[i]$ and $T[j]$ are aligned opposite each other. This is the case of Match/Mismatch.

2. $S : x...k.....\mathbf{i}$
 $T : y.....\mathbf{j}$

Optimal alignment of $S[1..k]$ and $T[1..j]$, where $0 \leq k \leq i - 1$, followed by $[i - (k + 1)]$ gaps in T . This is the case of Deletion.

3. $S : x.....\mathbf{i}$
 $T : y...k.....\mathbf{j}$

Optimal alignment of $S[1..i]$ and $T[1..k]$, where $0 \leq k \leq j - 1$, followed by $[j - (k + 1)]$ gaps in S . This is the case of Insertion.

The recurrence equations can therefore be represented by the following.

Recurrence:

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

In the general gap penalty model, to compute the alignment, we need to fill in all entries in the $n \times m$ dynamic programming matrix V . Each entry can be computed in $O(\max\{n, m\})$ time. In total, we would need $O(nm \max\{n, m\})$ time. We also need to allocate $O(nm)$ memory for the dynamic programming matrix.

2.5.2 Affine gap model

Computing an alignment in a general gap penalty model has a high time complexity, now we consider some gap penalty model which is biologically valid and has a better time complexity. In the Affine gap model, the penalty for a gap is divided into two parts. (1) A penalty (h) for initiating the gap, and (2) a penalty (s) depending on the length of the gap. The total penalty for a gap of length x is:

$$g(x) = h + xs \tag{2.1}$$

The model is called “affine” after its affine formula above. Note that the constant gap weight model is simply the affine model with $s = 0$, thus the algorithm described below can be used for the **constant gap penalty model** as well.

Affine Gap Penalty Algorithm

To align sequences S, T, consider the prefixes $S[1..i]$ of S and $T[1..j]$ of T. Any alignment of these two prefixes is one of the following three types:

1. $S : x..x\mathbf{i}$
 $T : y..y\mathbf{j}$

Alignment of $S[1..i]$ and $T[1..j]$ where characters $S[i]$ and $T[j]$ are aligned opposite each other. This includes both the case that $S[i] = T[j]$ and that $S[i] \neq T[j]$.

2. $S : xx..x\mathbf{i}$
 $T : yy.....y\mathbf{j}$

Alignment of $S[1..i]$ and $T[1..j]$ where character $S[i]$ is aligned to a character strictly to the left of character $T[j]$. Therefore, the alignment ends with a gap in S.

3. $S : xx.....x\mathbf{i}$
 $T : yy..y\mathbf{j}$

Alignment of $S[1..i]$ and $T[1..j]$ where character $S[i]$ is aligned to a character strictly to the right of character $T[j]$. Therefore, the alignment ends with a gap in T. We assume that

- G (i,j) is the score of the optimal global alignment between $S[1..i]$ and $T[1..j]$ of type 1, and
- E (i,j) is the score of the optimal global alignment between $S[1..i]$ and $T[1..j]$ of type 2, and
- F (i,j) is the score of the optimal global alignment between $S[1..i]$ and $T[1..j]$ of type 3, and
- V (i,j) is the maximum value of an alignment between $S[1..i]$ and $T[1..j]$, which is the maximum of E(i,j), F(i,j), and G(i,j).

In the base conditions ($i=0$ or $j=0$), we need to look at the insert and delete operations and the correct value is based not only on the weight of the space (xs), but also based on the weight of "opening the gap" (h).

When both $i, j > 0$, we define 3 recurrence relations, one for each of G(i, j), E(i, j) and F(i, j). Each will be calculated based on previously computed values. Take E(i,j) as an example. We are looking at alignments in which S ends to the left of T:

$S : atgci$
 $T : atgcatgcatgcj$

We identify two cases for the above alignment:

1. $T[j-1]$ maps with a space. In this case, we only need to add another “extension weight” to the value, forming the new weight $E(i, j-1) + s$
2. $T[j-1]$ maps with $S[i]$. In this case, we need to add both the gap “opening weight” and the gap “extension weight”, forming the new weight $V(i, j-1) + h + s$.

Taking the maximum of the two yields the value for $E(i, j)$. We could also calculate $F(i, j)$ and $G(i, j)$ using the similar argument. $V(i, j)$ is then calculated by simply taking the maximum of the three. The score for the optimal alignment is $V(n, m)$, the optimal alignment can be recovered by backtracking on the 4 tables.

Dynamic programming solution:

$$\begin{aligned}
 \text{Basis:} \quad & V(0, 0) = 0 \\
 & V(i, 0) = h + is \\
 & V(0, j) = h + js \\
 & E(i, 0) = -\infty \\
 & F(0, j) = -\infty
 \end{aligned} \tag{2.2}$$

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

Analysis of the space and time complexity

The time complexity is as before $O(nm)$. There is a need to save four matrices (for E , F , G and V respectively) during the computation. Hence, the space complexity is $O(nm)$.

2.5.3 Concave gap model

Many people (especially the biologists) feel that affine gap penalty does not truly represent the underlying biological mechanism. It may not be appropriate to assign a fixed cost for opening a gap and a fixed cost for extending a gap. The concave gap penalty function was proposed to describe the biological behavior better.

Concave Function

Definition: A function $w(x)$ is said to be concave if

$$w(x) - w(x + \Delta) > w(x') - w(x' + \Delta) \quad \forall x < x' \tag{2.3}$$

[4][5][6]

Figure 2.10 shows an example of concave function.

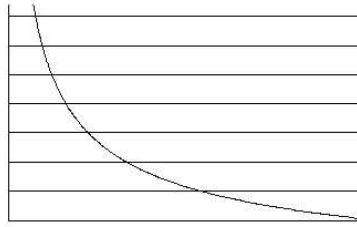


Figure 2.10: Concave function

Alignment with concave gap model

Suppose the gap function is concave, the penalty that we assign for a space in a gap is less than the penalty we assign to the space that precedes it. In other words, additional penalty incurred by additional space in a gap decreases as the gap gets longer while the penalty for opening a gap (i.e. the first space in the gap) is constant.

Let's recall the equation for computing $V(i, j)$ introduced in Section 2.5.1

Base cases:

$$V(0, 0) = 0$$

Recurrence: (for $i > 0$ and $j > 0$)

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

where

$$A(i, j) = \max_{0 \leq k \leq j-1} \{V(i, k) + g(j-k)\}$$

$$B(i, j) = \max_{0 \leq k \leq j-1} \{V(k, j) + g(i-k)\}$$

The bottleneck is the computation of $A(i, j)$ and $B(i, j)$ as it will take $O(n)$ time to fill in each element $A(i, j)$ and $B(i, j)$. Fortunately, we could design a method to speed them up[7].

Assume gap penalty is a concave function, we will show how to compute $A(i, 1), \dots, A(i, m)$ in $O(m \log m)$ time for a fixed i , and then $A(i, j)$ can be computed in $O(nm \log m)$ time for all i and j .

By using the same approach, $B(1, j), \dots, B(n, j)$ can be computed in $O(n \log n)$ time for a fixed j , and then $B(i, j)$ can be computed in $O(nm \log n)$ time for all i and j .

In total, all entries $V(i, j)$ can be filled in $O(nm \log(nm))$ time.

Now, we will prove that we can achieve such speed up.

Let's use the following simplifying substitution functions for a fixed i :

$$\begin{aligned} E(j) &= A(i, j) \\ D(k) &= V(i, k) \end{aligned}$$

Hence, the recurrence function of $A(i, j)$ can now be written as

$$E(j) = \max_{0 \leq k \leq j-1} \{D(k) + g(j - k)\}$$

By using dynamic programming, we can fill $E(i), \dots, E(m)$ in $O(m^2)$ time. Here we show that $E(i), \dots, E(m)$ can be computed in $O(m \log m)$ time. Let

$$C(k, j) = D(k) + g(j - k)$$

and thus,

$$E(j) = \max_{0 \leq k \leq j-1} C(k, j) \tag{2.4}$$

Lemma 2.2 $\forall k_1 < k_2,$

$$C(k_1, j) \geq C(k_2, j) \rightarrow C(k_1, j + 1) \geq C(k_2, j + 1)$$

Proof: Assume that $C(k_1, j) \geq C(k_2, j)$

$$\begin{aligned} C(k_1, j + 1) &= C(k_1, j) + g(j + 1 - k_1) - g(j - k_1) \\ &\qquad\qquad\qquad \text{since } C(k_1, j) \geq C(k_2, j) \\ &\geq C(k_2, j) + g(j + 1 - k_1) - g(j - k_1) \\ &\qquad\qquad\qquad \text{since } g(q) \text{ is concave} \\ &\geq C(k_2, j) + g(j + 1 - k_2) - g(j - k_2) \\ &= C(k_2, j + 1) \end{aligned}$$

■

How does Lemma 2.2 impact the filling of the dynamic programming matrix? Once we know that for a particular j , $C(k_1, j) \geq C(k_2, j)$, then for any $j' > j$ we no longer need to consider or calculate $C(k_2, j')$ as $C(k_2, j')$ must not be the maximum value.

Lemma 2.3 $\forall k_1 < k_2$,

let $h(k_1, k_2) = \arg \min_j \{C(k_1, j) \geq C(k_2, j)\}$, where $k_2 < j \leq n$.

Then we have:

$$C(k_1, j) < C(k_2, j) \text{ for } k_2 < j < h(k_1, k_2) \quad (2.5)$$

and

$$C(k_1, j) \geq C(k_2, j) \text{ for } h(k_1, k_2) \leq j \leq m \quad (2.6)$$

Proof: It's trivial to prove Equation 2.5 as it follows from the definition of $h(k_1, k_2)$. For Equation 2.6, we can readily apply the result of Lemma 2.2 to prove it. ■

What is Lemma 2.3 trying to say? It simply says that when any two curves $C(k_1, j')$ and $C(k_2, j')$ cross, there can be only **one** intersection, and that happens at $h(k_1, k_2)$. Figure 2.11 explains this phenomena graphically.

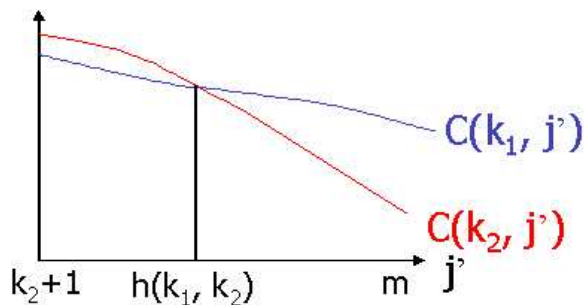


Figure 2.11: Graphical interpretation of lemma 2.3

Please take note that $C(k, j')$ is a decreasing function for a fixed k . When 2 curves cross, they can only intersect at one point. After the intersection, k of the top curve is less than k of the bottom curve. Thus, $h(k_1, k_2)$ can be found in $O(\log m)$ time by binary search.

Frontier of all curves

For a fixed j , consider curves $C(k, j')$ for all $k < j$ as shown in Figure 2.12(a), the black curve represents $\max_{k < j} C(k, j')$, which covers all the curves $C(k, j)$ for $k \leq j$, where $j = 5$. It is essential to trace the frontier curve to fill in $E(x) = A(i, x)$.

According to Lemma 2, any two curves can only overlap at one point, thus for a fixed j , the black curve can be represented by as shown in Figure 2.12(b), where $k_0 < k_1 < \dots < k_{top} < j < h_{top} < \dots < h_0$ (by default, $h_0 = m + 1$). In

this algorithm, (k_x, h_x) are stored in a stack with (k_{top}, h_{top}) at the top of the stack.

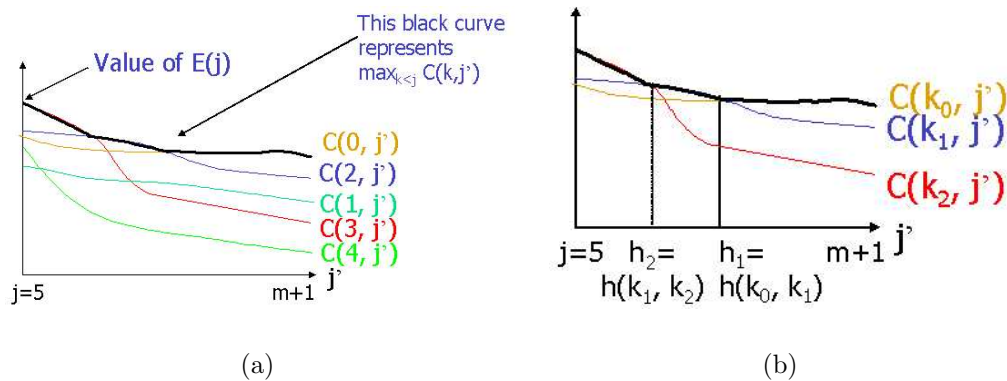


Figure 2.12: Frontier curves

Let's consider how to calculate $\max_{k < j} C(k, j')$:

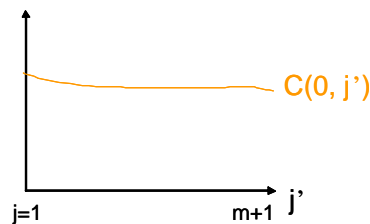


Figure 2.13:

(1) For $j = 1$, the set of curves $\{C(k, j') | k < j\}$ contains only one curve $C(0, j')$ as shown in figure 2.13. Thus, $\max_{k < j} C(k, j') = C(0, j')$ and $\max_{k < j} C(k, j')$ can be represented by $(k_0 = 0, h_0 = m + 1)$.

(2) For a particular $j > 1$, suppose the curve $\max_{k < j} C(k, j')$ is represented by $(k_0, h_0), \dots, (k_{top}, h_{top})$. Let's consider how to get the curve $\max_{k < j+1} C(k, j')$. There are two cases for overlaying of $C(j, j')$ with the curve $\max_{k < j} C(k, j')$. These two cases are described below:

Two possible outcomes from the overlaying of $C(j, j')$ with the curve $\max_{k < j} C(k, j')$ are:

1. if $C(j, j+1) \leq C(k_{top}, j+1)$, then
The new curve $C(j, j')$ can't cross $C(k_{top}, j')$ and will always be below

$C(k_{top}, j')$. Thus, the frontier curve at $j + 1$ is still the same to that of j . Figure 2.14(a) illustrates this case.

2. if $C(j, j + 1) > C(k_{top}, j + 1)$, then the curve $\max_{k < j+1} C(k, j')$ as shown below. We need to update it. Figure 2.14(b) illustrates this case.

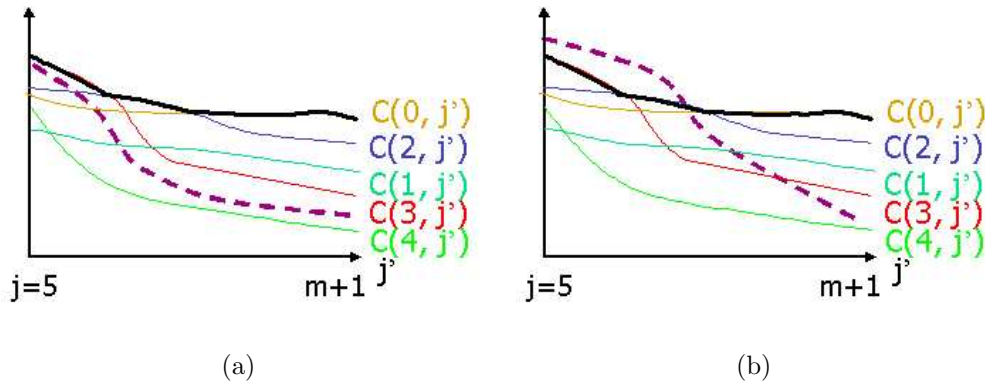


Figure 2.14: Possible outcome of overlaying: (a) below, and (b) above.

Having discussed the way to exploit inherent properties of concave gap penalty function, below is the algorithm for finding the frontier curve:

Algorithm

```

Push  $(1, m + 1)$  onto stack  $S$ 
 $E[1] = C(k_{top}, 1)$ 
For  $j = 1$  to  $m - 1$  {
  if  $C(i, j + 1) > C(k_{top}, j + 1)$  then{
    while  $S \neq \emptyset$  and  $C(j, h_{top} - 1) > C(k_{top}, h_{top} - 1)$  do
      pop  $S$ 
    if  $S = \emptyset$  then
      push  $(j, m + 1)$  onto  $S$ 
    else
      push  $(j, h(k_{top}, j))$ 
  }
   $E[j] = C(k_{top}, j + 1)$ 
}

```

It is easy to see that for every j , we push at most one pair onto stack S . Thus, we push at most m pairs onto stack S , and we can only pop at most m pairs out of the stack S . As the $h()$ value of each pair can be computed in $O(\log m)$ time, by binary search, the total time is $O(m \log(m))$ time.

References

- [1] DOOLITTLE et al., “Simian Sarcoma Virus onc Gene, v-sis, is Derived from the Gene (or Genes) Encoding a Platelet-Derived Growth Factor” *Science*. 221, 275-277, 1983.
- [2] RIORDAN et al., “Identification of the Cystic Fibrosis Gene: Cloning and Characterization of Complementary DNA (in Cystic Fibrosis: Cloning and Genetics)” *Science*. 245, 1066-1073, 1989.
- [3] MYERS et al., “Optimal alignments in linear space” *Comp. Appl. Biosci.* 4, 11-17, 1988.
- [4] GRADSHTEYN, I. S. and RYZHIK, I. M., “Tables of Integrals, Series, and Products”, 5th ed, San Diego, CA: *Academic Press*, p. 1132, 2000.
- [5] EGGLETON, R. B. and GUY, R. K., “Catalan Strikes Again! How Likely is a Function to be Convex?” *Math. Mag.* 61, 211-219, 1988.
- [6] WEBSTER, R. *Convexity*. Oxford, England: Oxford University Press, 1995.
- [7] DAVID EPPSTEIN et al., “Speeding up Dynamic Programming” *Proc. 29th Symp. Foundations of Computer Science*, *IEEE*, 488-496, 1988.