CS5238 Combinatorial methods in bioinformatics 2002/2003 Semester 1

Lecture 2: Sequence Comparison - August 16, 2002

Lecturer: Wing-Kin Sung *Scribe: Li Qiu Xiang, Vaibhav Rastogi, V B Vega S, Xu Jinmei, Xu Xin*

2.1 The goal of sequence similarity

There is a well-known conjecture in the industry of biology. Given two DNAs, RNAs or Proteins, if they are highly similar, we can infer that the DNAs, RNAs or Proteins share similar function or similar 3D structure. 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 often modifies the DNA of species in the way of mutation.By measuring the similarity of their genome, we can know their evolution distance. Such knowledge is especially important for biology scientist.
- 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 reconstructing long sequences of DNA from overlapping sequence fragments, 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 The string alignment between two strings

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.

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. During the above example, the alignment contains 8 pairs. 5 pairs are match. 1 pair is mismatch. 1 pair is delete. 1 pair is insert.

$$S = \texttt{A-CAATCC}$$
$$T = \texttt{AGCA-TGC}$$

2.2.2 Goodness evaluations

The goodness of 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 2.1 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**.

Some people like to use **distance function** to evaluate the goodness of alignment. E.g., one distance function is as follows: match(0), mismatch(1), insert(1), delete(1). The **optimal alignment problem** tries to find the alignment with the minimum distance.

Usually, the minimum distance is called the **edit distance**. The edit distance between two sequences could also be thought as the minimal number of edit operations (insertions, deletions and substitutions) needed to transform one sequence into the other, which could be used to roughly measure the replication process of DNA sequences.

	-	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 similarity function

Let's compare the similarity function and distance function. We could find out that if you negate the values in the distance function, it becomes a similarity function.Since similarity and distance are dual concepts, we only study similarity function in this note.
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

2.2.3 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 $T[1..j]$.
Basis:

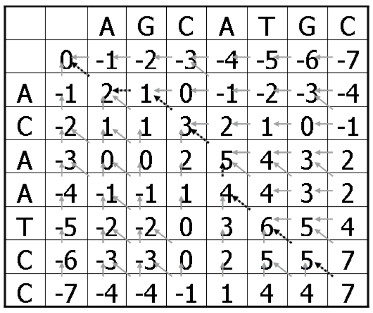
$$\begin{array}{ll} V(0,0) = 0 & \\ V(0,j) = V(0,j-1) + \delta(\sqcup, T[j]) & \text{Insert } j \text{ times} \\ V(i,0) = V(i-1,0) + \delta(S[i], \sqcup) & \text{Delete } i \text{ times} \end{array}$$

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

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

For the alignment between $S[1..i]$ and $T[1..j]$, the last pair of alignment should be match, mismatch,delete or insert. To get the optimal score,we choose the maximum value among these three cases. Thus, we have the above recurrence.

Figure 2.2 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 choosing the maximum of $0+2$,$-1-1$,$-1-1$. Let's calculate another one, e.g., row 2 column



		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 $S = AGCATGC$ and $T = ACAATCC$

3. We choose the maximum value of $-1+2$, $2-1$,$-2-1$. Note that in this case, there are two ways to get the maximum value. In Figure 2.2, we draw arrows to indicate all the ways to get the maximum values.The max alignment score we obtained at the right bottom corner of the table 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.

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 $= O(nm)$

2.2.4 Problem about the 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. In other word, when we fill in a row, we only need the V-values of the current row and the previous row. The V-values in all other rows are not necessary. 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 $= O(min(n, m))$

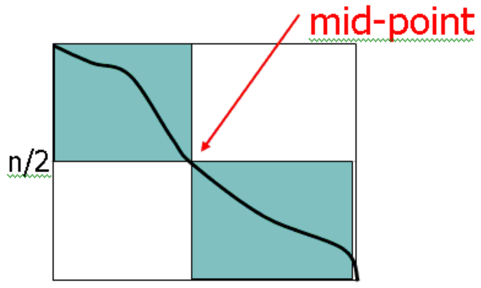


Figure 2.3: Mid-point Example

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. based on the cost-only algorithm in Section 2.2.4.1.

The main observation is the following equation. It means that the optimal alignment of $(S[1..n], T[1..m])$ is the union of (1) the optimal alignment of $(S[1..n/2], T[1..j])$ and (2) the optimal alignment of $S[n/2+1..n], T[j+1..m])$. Figure 2.3 illustrates this idea.

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

The entry $(n/2, j)$ 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:

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

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$(S[i_1..i_2], T[j_1..j_2])$

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 Alignment$(S[i_1..mid], T[j_1..j])$ and Alignment$(S[mid+1..i_2], T[j+1..j_2])$

Analysis

- **Space**: Working memory for finding mid-point takes $O(m)$ space. Each dynamic programming computation requires storing one additional row, which can be discarded once the middle point is found. Thus, in each recursive call, we only need to store the alignment path. If $n < m$ we can store the middle column instead. Therefore the space complexity is $O(min(m, n))$.

- **Time**: Time for step 1 is $O(n/2m)$. Time for step 2 is also $(n/2m)$.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) = time for finding mid-point + time for solving the two subproblems= $O(nm) + T(n/2, j) + T(n/2, m - j)$.Thus, time complexity $= T(n, m) = O(nm)$.

2.2.5 Problem about the 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 $\Omega(nlogn)$ time.

- Masek and Paterson (1980)—Based on Four-Russian's paradigm, the string alignment problem can be solved in $O(nm/logn)$ time.

Let d be the total number of inserts and deletes. Note that $0 < d < n+m$. If d is smaller than $n+m$, we can get a better solution. Observe that the alignment should be inside the $(2d + 1)$ band, since the number of deletes and inserts is at most d (see Figure 2.4). 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 $O((n + m)d)$.
Example (Figure 2.5): We could see the maximum score is 7. The most optimal alignment in this example is A-CAATCC and AGCA-TGC $(2 - 1 - 3 - 5 - 4 - 6 - 5 - 7)$. The optimal alignment is not unique. We may get other optimal alignments if there are more than one back tracing path.

2.3 Local alignment

We know that global alignment is used to align the entire sequences. Sometimes we are interested in finding the subsequences of the input sequences whose alignment has the highest score. In other words, we are ready to ignore any deletions

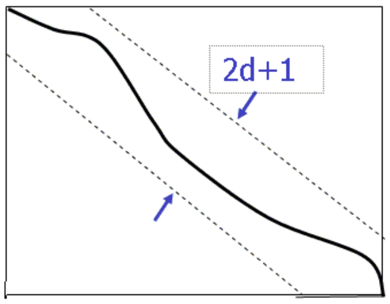
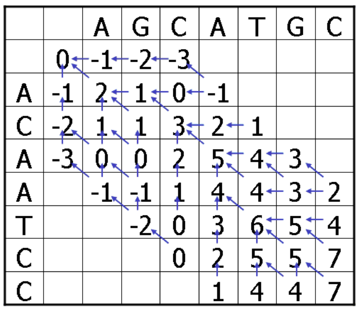


Figure 2.4: $2d + 1$ Band



		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

which happen at the beginning or end of any of the two sequences. Such sequence comparison is called **local alignment**. Local alignment searches for regions of local similarity and does not include the entire length of the 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 as follows. 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 value over all pairs of such substring A and B.

Clearly, there are n^2 choices of substring from S and m^2 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

We can see that 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[7] = 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 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 $1 \leq j \leq m$.

Intuitively, the score of local alignment is $max_{i,j} V(i,j)$. Smith-Waterman dynamic programming algorithm for optimal local alignment is quite similar to that of global alignment. Detail of this algorithm is given as follows.
Basis:

$V(i,0) = 0$
$V(0,j) = 0$
Since the optimal suffix to align with a string of length 0 is the empty suffix.
Recurrence: (for $i > 0$ and $j > 0$)

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

2.3.3 Example

For example, let $S = CTCATGC$ and $T = ACAATCG$, a match score +2, an insert and a delete score -1. Smith-Waterman dynamic programming algorithm fills in the V table with values from top to bottom and left to right. Figure 2.6 shows the V table. The maximum score 6 is the score of the optimal local alignment of S and T.

The path in Figure 2.6 corresponds to the optimal alignment, which is:

```
C_ AT_ G
CAATCG
```

	_	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 2.6: The V table for local alignment between $S = CTCATGC$ and $T = ACAATCG$

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 seen two kinds of sequence alignment: 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 aligns two sequences as a whole. The different 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.

To better appreciate the motivation behind semi-global alignment, let's consider the following example from [SM97]. Suppose we have an original sequence S and a target sequence T below:

$$\begin{aligned} S &= \texttt{CAGCACTTGGATTCTCGG} \\ T &= \texttt{CAGCGTGG} \end{aligned}$$

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

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

However, 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 in 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
TCAACGATCACCGCA--------
```

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

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 column row

Table 2.1: Charging of spaces in semi-global alignment.

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

2.5 Gap penalty model

Following the spirit of relaxing the cost incurred by spaces (which mark deletion or insertion), we shall now delve into the idea of gaps. Some literatures on bioinformatics use the term space and gap interchangibly, in this notes, we define gap as follows:

Definition 2.1 *A gap in an alignment ia a maximal substring of contiguous spaces in any of the aligned sequences.*

By this definition, there are 3 gaps in the following alignment:

```
A-CAACTCGCCTCC--
AGCA-----CCTGCAA
```

In the earlier discussion, we have seen that mutations normally happen in chunks. A string of residue might be inserted or delete in one instance. Hence, it's only natural not to impose a penalty that is strictly propotional to the length of the gap. Such scheme is also preferable in situations that we expect spaces to appear contiguously, for example when aligning mRNA with its gene.

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 gap penalty model using the following dynamic programming.

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

Base cases:
$V(0,0) = 0$
$V(0,j) = g(j)$
$V(i,0) = g(i)$

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

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

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

2.5.2 Affine gap model

How, then, should one assign a cost to a gap? In the Affine gap model, the penalty for a gap is divided into two parts. (1) P_g for initiating the gap, and (2) P_s depending on the length of the gap. The total penalty for a gap of length q is:

$$P_{Total} = P_g + qP_s \tag{2.1}$$

The model is called "affine" after its affine formula above. Note that the constant gap weight model is imply the affine model with P_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. $S----i$
   $T----j$

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. $S----i$
   $T------------j$

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. $S------------i$
   $T----j$

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 maximum value of any alignment between $S[1...i]$ and $T[1...j]$ of type 1, and
- E (i,j) is the maximum value of any alignment between $S[1...i]$ and $T[1...j]$ of type 2, and
- F (i,j) is the maximum value of any 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 indel operations and the correct value is not only based on the weight of the space (qP_s), but also based on the weight of "opening the gap"(P_g).

$\forall i, j > 0$, 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:

$S----i$

$T-------------j$

We identify two situations 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) + P_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 $V(i, j-1) + P_g + P_s$.

Taking the maximum of the two yields the value for $E(i, j)$. We could calculate $F(i, j)$ and $G(i, j)$ following the similar argument. $V(i, j)$ is 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:

$$\begin{array}{ll}
\text{Basis:} & V(0,0) = 0 \\
 & V(i,0) = E(i,0) = P_g + iP_s \\
 & V(0,j) = F(0,j) = P_g + jP_s \\
\text{Recurrence:} & V(i,j) = max\{E(i,j), F(i,j), G(i,j)\} \text{ where} \\
 & G(i,j) = V(i-1, j-1) + \sigma(S[i], T[j]) \\
 & E(i,j) = max\{E(i,j-1) + P_s, V(i,j-1) + P_g + P_s\} \\
 & F(i,j) = max\{F(i-1,j) + P_s, V(i-1,j) + P_g + P_s\}
\end{array} \tag{2.2}$$

Analysis of 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 trully represent the underlying biological mechanism. More is needed than just assigning a fixed cost for opening a gap and a fixed cost for extending a gap. To answer this, the idea of concave gap penalty function was proposed. It is said to describe the biological behaviour better.

Convex and Concave Function

Before we elaborate on concave gap penalty model, let's first refresh our memory on convex and concave function.

Definition 2.2 *Convex function is a function whose value at the midpoint of every interval in its domain does not exceed the average of its values at the ends of the interval.*

In other words, a function f(x) is convex on an interval [a, b] if for any two points x_1 and x_2 in [a, b],

$$f[\frac{1}{2}(x_1 + x_2)] \leq \frac{1}{2}[f(x_1) + f(x_2)] \tag{2.3}$$

[GIRI00].

If f(x) has a second derivative in [a, b], then a necessary and sufficient condition for it to be convex on that interval is that the second derivative $f''(x) > 0$ for all x in [a, b].

If the inequality above is strict for all x_1 and x_2, then f(x) is called strictly convex. Examples of convex functions include x^p for $p \geq 1$, $x \ln x$ for $x > 0$, and $|x|$ for all x. If the sign of the inequality is reversed, the function is called concave.

Definition 2.3 *A function f(x) is said to be concave on an interval $[a, b]$ if, for any points x_1 and x_2 in $[a, b]$, the function $-f(x)$ is convex on the interval.*

If the second Derivative of f, $f''(x)$, is bigger than 0 on an open interval (a, b) (where $f''(x)$ is the second Derivative), then f is convex on the interval. If $f''(x) < 0$ on the interval, then f is concave on it [ERG88] [WR95].

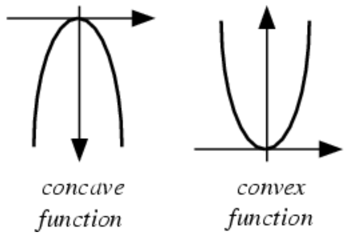


Figure 2.7: Concave function vs. Convex function

Concave vs. convex gap model
In bioinformatics literatures, some authors prefer to use convex function instead of concave function, for example in [G97] page 293. It is worth to note that despite of the different functions used, the underlying motivation is the same. When we talk about concave gap penalty, it is assumed that the function $g(q)$ is negative, i.e. $g(q) \leq 0$. Whereas in discussing convex gap penalty, it is assumed that $g(q)$ is positive, i.e. $g(q) \geq 0$.
Alignment with concave gap model
Under the concave gap model, the gap penalty function $g()$ employed is a concave

function. By this, effectively, the penalty that we assign for a space in a gap is less than the penalty assigned to the space that precede it. In other words, additional penalty incurred by additional space in a gap decreases as the gap gets longer. The penalty for opening a gap (i.e. the first space in the gap), however, is constant. An example of concave gap function, as given in [LM95], is $g(q) = a \log x + b$.

The concave gap penalty function $g(q)$ could readily be applied to the dynamic programming algorithm for the general gap penalty. Below we restate the algorithm for easy reference. Only a small change was made. We introduce two substitution functions $A(i,j)$ and $B(i,j)$ that serve to ease the discussion of the upcoming proof.

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

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

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

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

Filling up the dynamic programming matrix for this algorithm is extremely time consuming, because it takes $O(n)$ time to fill in each $A(i,j)$ or each $B(i,j)$. Fortunately, we could speed them up. Let's assume for a moment that we can perform the speed up such that:

- For a fixed i, $\forall j \in \{1..m\}$, $A(i,j)$ can be computed in $O(m \log m)$ time. Thus, $\forall i \in \{1..n\}, \forall j \in \{1..m\}$, $A(i,j)$ can be computed in $O(nm \log m)$ time.
- For a fixed j, $\forall i \in \{1...n\}$, $B(i,j)$ can be computed in $O(n \log n)$ time. Thus, $\forall i \in \{1..n\}, \forall j \in \{1..m\}$, $B(i,j)$ can be computed in $O(nm \log n)$ time.
- Overall, all entries of $V(i,j)$ can be filled in $O(nm \log(nm))$ time.

The only remaining task now is to prove that we can achieve such speed up. Let's now consider algorithm for filling up $A(i,j)$ where i is fixed. Let's also use the following simplifying substitution functions:

$$E(j) = A(i,j)$$
$$D(k) = V(i,k)$$

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

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

Regular dynamic programming can fill $E(i), .., E(m)$ in $O(m^2)$ time. Here we shall show how to fill them in $O(m\log m)$ time. For that we need the following lemmas. Also, from this point onwards let

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

and thus,

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

Lemma 2.4 *$\forall k_1 < k_2$,*

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

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

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

■

How would Lemma 2.4 impact the filling of the dynamic programming matrix? Once we know that for a particular j, $C(k_1,j) \ge C(k_2,j)$, 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 2.5 *$\forall k_1 < k_2$,*
let $h(k_1,k_2) = \arg\min_j\{C(k_1,j) \ge C(k_2,j)\}$, where $k_2 < j \le n$.
Then we have:

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

and

$$C(k_1,j) \ge C(k_2,j) \text{ for } h(k_1,k_2) \le j \le m \tag{2.6}$$

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 2.4 to prove it. ■

What is Lemma 2.5 trying to say? It simply says that when any two curves $C()$ cross, there can be only one intersection, and that happens at $h(k_1, k_2)$. Figure 2.8 explains this phenomena graphically.

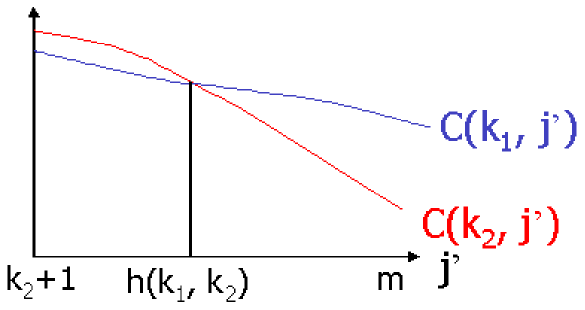
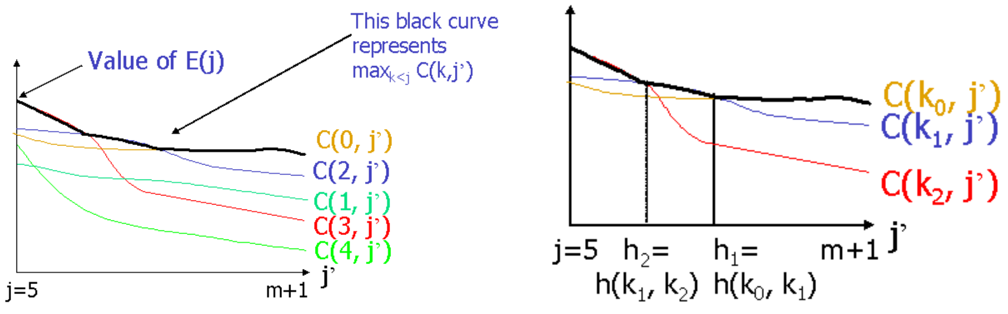


Figure 2.8: Graphical interpretation of lemma 2.5

Notice also that after the intersection, k of the top $C(k, j)$ curve is greater than k of the lower $C(k, j)$ curve. Determining $h(k_1, k_2)$ is known to be of $O(\log m)$ time, utilizing a binary search approach.

Frontier of all curves

Now, given the above lemmas and the fact that, by Equation 2.4, filling in $E(x)$ $(= A(i, x))$ is essentially tracing the enveloping curve or the frontier curve that cover all the curves $C(k, j)$ for $k < j$. For example, in figure 2.5.3 the thick black curve represent the values $max_{k<5}C(k, j')$ for $5 \le j' \le m+1$. In particular, when $j' = 5$, $E[5] = max_{k<5}C(k, j')$



(a) (b)

Figure 2.9: Frontier curves

For a fixed j, the black curve can be represented by a piecewise function that is defined over the intervals $(k_0, h_0), (k_1, h_1), ..., (k_{top}, h_{top})$. For instance, consider the thick black curve in figure 2.5.3. The curve can be defined as:

$$f(j) = \begin{cases} C(k_2, j) & \text{for } j < h_2 \\ C(k_1, j) & \text{for } h_2 \leq j < h_1 \\ C(k_0, j) & \text{for } j \geq h_1 \end{cases}$$

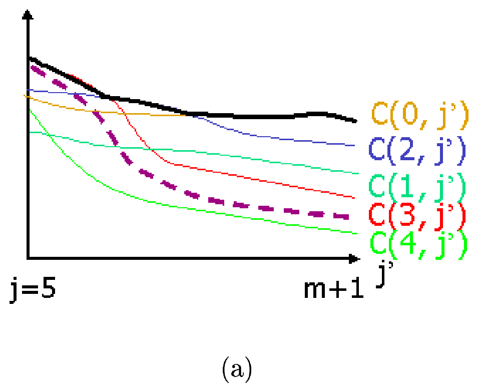
The choice of $C(k_x, j)$ for each interval is not abitrary, but rather it is chosen to be the maximum one.

How can we utilize this? Suppose one has calculated $E[z]$ for all $z \leq j$, and we have $\max_{k<j} C(k, j')$. Now, to fill $E[j+1]$ (i.e $\max_{k<j+1} C(k, j')$) one would only need to overlay the curve $C(j, j')$ with the curve $\max_{k<j} C(k, j')$. One could easily keep $\max_{k<j} C(k, j')$ by storing the intervals $\{(k_0, h_0), ..., (k_{top}, h_{top})\}$, which in the algorithm below is stored in a stack with (k_{top}, h_{top}) at the top of the stack.

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 $C(k_{top}, j')$ (by lemma 2.4). Thus, the frontier curve at $j+1$ is still the same to that of j. Figure 2.5.3 illustrates this case.

2. if $C(j, j+1) > C(k_{top}, j+1)$, then
   A new curve has overtake the old frontier curve. In this case, the curve $\max_{k<j+1} C(k, j')$ is different from the $\max_{k<j} C(k, j')$ curve. Updates would be needed to the set of intervals. Figure 2.5.3 describes this pictorially.



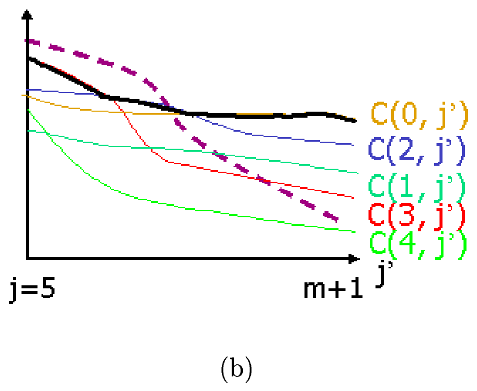


Figure 2.10: Possible outcome of overlaying: (a) below, and (b) above.

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

```
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 and thus, overall, we push at most m pairs onto stack S, which in turn limits the number of pops from stack S to m pops. Since the $h()$ value for each pair can be computed in $O(\log m)$ time, as discussed earlier, the total time to fill $E[j]$ is $O(m \log m)$. By this, we have now established that indeed $V(i, j)$ can be filled in $O(nm \log(nm))$ time.

References

[ERG88] Eggleton, R. B. and Guy, R. K., "Catalan Strikes Again! How Likely is a Function to be Convex?" *Math. Mag.* 61, 211-219, 1988.

[GIRI00] Gradshteyn, I. S. and Ryzhik, I. M., "Tables of Integrals, Series, and Products", 5th ed, San Diego, CA: *Academic Press*, p. 1132, 2000.

[G97] Dan Gusfield, "Algorithms on strings, trees, and sequences", *Cambridge University Press*, 1997.

[LM95] Eric S. Lander and Michael S. Waterman, "Calculating the secrets of life : applications of the mathematical sciences in molecular biology", National Academy Press , p.71, 1995. See http://www.nap.edu/books/0309048869/html/ for the online version.

[SM97] Joao Setubal and Joao Meidanis, "Introduction to Computational Molecular Biology", *PWS Publishing*, 1997.

[WR95] Webster, R. *Convexity.* Oxford, England: Oxford University Press, 1995.