CS5238 Combinatorial Methods in Bioinformatics 2005/2006 Semester I

Lecture 2: Sequence Comparison - August 19, 2004

Lecturer: Asst. Prof. Sung Wing Kin, Ken *Scribe: Yin Hongli*

2.1 Background

2.1.1 Early Research

The earliest research in sequence comparison was as early as in 1983,when Doolittle et al[1] searched for platelet-derived growth factor (PDGF) in their database. They found that PDGF is similar to v-sis oncogene.

```
PDGF-2 01 ------SLGSLTIAEPAMIAECKTREEVFCICRRL?DR??  34
p28sis 61 LARGKRSLGSLSVAEPAMIAECKTRTEVFEISRRLIDRTN 100
```

At that time, the function of v-sis oncogene is still unknown. Based on the similarity between the transforming protein derived from the simian sarcoma virus oncogene, v-sis, and the human platelet-derived growth factor(PDGF), they claimed that the transforming protein of the primate sarcoma virus and the platelet-derived growth factor are derived from the same or closely related cellular genes. This was later proved by scientists. Another research conducted by Riordan et al[14] showed a similar result. They tried to understand the cystic fibrosis gene using multiple sequence alignment in 1989. Similar gene sequences did imply similar amino acids and therefore imply similar functionalities.

So, there is such a well-known conjecture in biology : If any two DNA(or RNA, or Protein)sequences are similar, they will have similar functions or 3D structures. However the reverse may not always be true. For example, tRNAs all have similar structures, but do not exhibit much similarities in their sequences. Researchers in bioinformatics often compare the similarity between two biological sequences to understand their structures or functionalities. Typical applications of sequence comparison including but not limited to:

- Inferring the biological function of a gene (or RNA or protein).
  When one gene looks similar to another gene with a known function, both genes may have similar functions.

- Finding the evolution distance between two species.
  Evolution modifies the genomes of the species through mutation. By measuring the similarity of their genomes, we can know their evolution distance.

For example, we can find the closest evolution distance among human, mouse and chicken by measuring the similarity between their genomes.

- Helping genome assembly.
  This is based on the overlapping information of a huge amount of short DNA fragments. For instance, the Human Genome Project reconstructed the whole genome. The overlapping information was extracted using sequence comparison.

- There are many other applications of sequence comparison which include finding common subsequences of two genomes, finding repeats within a genome, reconstructing long sequences of DNA from overlapping sequence fragments, and searching databases for related sequences and subsequences, etc.

2.2 Global Alignment Problem

2.2.1 String Edit and string alignment

String edit problem or global alignment problem deals with the problem of transforming one string to another using minimum number of operations. Allowed string operations are:

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

Given two strings A = *interestings* and B = *bioinformatics*, A can be transformed to B using 5 replacements, 3 insertions and 1 deletion.

```
-i--nterestings
bioinformatic-s
101101101100110
```

We can associate a cost function to the operations and minimize the total cost. Such a cost is called the *edit distance*. For the above example, the edit distance is 9 if we assume the cost for each of the three operations above is 1. In computational biology, instead of using string edit, people like to use string alignment with a similar concept- String edit problem measures the minimum cost of transforming one string to another by making use of a cost function, while string alignment problem measures the maximum similarity or 'goodness' when one string is compared to another by making use of a similarity function. When the values of a cost function are negated, we get a similarity function. Cost minimization is

	-	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

equivalent to similarity maximization, in fact, string edit and string alignment are dual problems.

String alignment gives two sequences of equal length by inserting a number of spaces into the sequences. When two 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*. A space in the second sequence is called a *delete*. A space must always be aligned with a character. Let us consider two strings $ACAATCC$ and $AGCATGC$. One of the possible alignments is shown below.

S = `A-CAATCC`
T = `AGCA-TGC`

The above alignment has 5 matches, 1 mismatch, 1 insert and 1 delete. 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, which are either single characters or single spaces. Figure 2.1 shows a similarity function for the above example, 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 an alignment is called an *optimal alignment*. Other alignments of the two strings can only get a score that is not more than the optimal alignment. String alignment problem tries to find the alignment with the maximum similarity score. This problem is also called **global alignment problem**. Note that two strings can have more than one optimal alignment.

2.2.2 Needleman-Wunsch Algorithm

This section introduces the Needleman-Wunsch algorithm which applies dynamic programming to find the optimal global alignment between two strings. Consider

two strings: $S[1..n]$ and $T[1..m]$. Define $V(i,j)$ to be the score of the optimal alignment between $S[1..i]$ and $T[1..j]$.
When either i=0 or j=0, we have the following equations which refer to the case when a string is aligned with an empty string.
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: When both $i > 0$ and $j > 0$, for the 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:

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

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. For example, the value at V(1,1) ie., the intersection of the second row and the second column, the value 2 is obtained by choosing the maximum of $(0+2,-1-1,-1-1)$. In case of V(2,3), 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 score of the optimal alignment is obtained at the bottom right corner of the table V(7,7) which is 7. The optimal alignment is obtained by tracing the arrows 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 will be present in the alignment. In Figure 2.4, 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: A-CAATCC and AGC-ATGC.

Analysis We need to fill in all entries in the table for a $n \times m$ matrix. Each entry can be computed in $O(1)$ time, and each entry needs $O(1)$ space storage. Therefore the time and complexities for the algorithm are:

- Space complexity = $O(nm)$.
- Time complexity = $O(nm)$

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

Figure 2.2: The dynamic programming table for $S = AGCATGC$ and $T = ACAATCC$

2.2.3 Problem on Speed

This problem has been studied by a few researchers as shown below.

- 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.

Since 1980, it is open whether the time complexity can be further improved. Suppose we further restrict that the maximum allowed number of insertions and deletions in the alignment is d. Let d be the maximum allowed number of insertions and deletions. Obviously, $0 < d <= n + m$. Note that insertion moves horizontally, and deletion moves vertically. Hence, the alignment should be inside the $(2d + 1)$ band.(see Figure 2.3). 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 $O((n+m)d)$.

2.2.4 Problem on space

Note that the dynamic programming requires $O(mn)$ space. When we compare two very long sequences, space will be a problem. For example, if we compare human genome with mouse genome, we need $3G * 3G = 9$ billion G memory space which is not feasible. Can we solve the string alignment problem using less space?

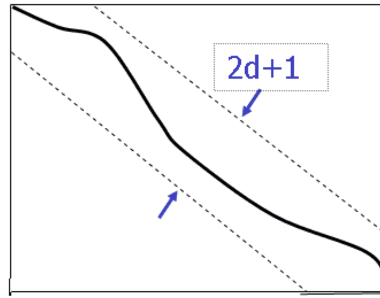

Figure 2.3: $2d + 1$ Band

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

Figure 2.4: 2d+1 Band Example

In the previous example, observe that the table can be filled in row by row. For filling a row in the table, only the values in the the previous row are needed. As shown in Figure 2.5, row 4 is depending on row 3 only in filling its entries.

Therefore, if we just want to know the score of the optimal alignment it is not necessary to store all the values in the table, in general we need two rows instead. A $m \times n$ table can be filled row by row if n is smaller than m, and column by column if m is smaller. Thus, the space complexity becomes $O(min(n, m))$.

In fact, we can deduce the optimal alignment of two strings in $O(n+m)$ space. Based on the cost-only algorithm mentioned above, we can find the midpoint of

	_	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 2.5: V table using O(min(n,m)) space

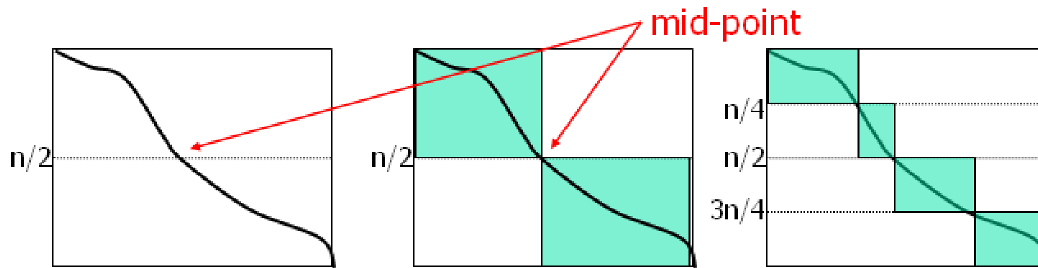

Figure 2.6: Mid-point Example

	_	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 2.7: Mid-point Example

the alignment, divide the problem into two halves, and then recursively deduce the alignments for the two halves.

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 2.6 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:

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!

Figure 2.7 shows an example of how to do this. In step 1,we fill in the first half of the table, and in step 2 we reversely fill the second half. Then in step 3 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 which is 4.

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 detail algorithm is as following:

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

- **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 computing the whole alignment.T(n, m) = time for finding mid-point + time for solving the two subproblems= $O(nm) + T(n/2, j) + T(n/2, m - j)$.By solving the recursive equation, the time complexity is $T(n, m) = O(nm)$.
- **Space**: Working memory for finding 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 $O(m, n)$.

2.2.5 More on string alignments

There are two special cases for string alignment problem:

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. In the LCS problem, we are given two sequences X and Y and wish to find a maximum-length common subsequence of X and Y. This problem can be solved efficiently using dynamic programming.
    - score for match = 1
    - score for mismatch = -ve infinity which is not allowed
    - score for insert/delete = 0
2. Hamming Distance: The Hamming distance is the number of positions in two strings of equal length for which the corresponding elements are different. For example, the Hamming distance between two strings "*toned*" and "*roses*" is 3. It can be computed using the following scoring function:

- score for match = 1
- score for mismatch = 0
- score for insert/delete = -ve infinity
  which is not allowed

2.3 Local Alignment

Global alignment is used to align entire sequences. Sometimes we are interested in finding the subsequences of the input sequences whose alignment has the highest global alignment score among all subsequences generated by the two input sequences. Such a sequence comparison is called **local alignment**. Local alignment searches for regions of local similarity and does not necessarily include the entire sequences. Possible application could be: Given two long DNAs, both of them contain the same gene or closely related gene, can we identify them?

Let us first look at the Brute-Force Solution: Consider two strings $S[1..n]$ and $T[1..m]$. The algorithm compares every substring A of S with every substring B of T, and returns the optimal local alignment.

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.

- Time: There are $n^2/2$ choices for A and $m^2/2$ choices for B. Thus, the total time complexity of this method is $O(n^3m^3)$.

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, let us go through some background information:

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] = ACCGATT, then ACC is a prefix of S and GATT is a suffix of S. Note that empty string is both prefix and suffix of S.

V(i,j) Given two strings A and B with |A| = n, |B| = m. V(i,j) is the maximum score of the 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 $1 \leq j \leq m$.

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

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

The definition of local alignment allows us to choose the best alignment among any substrings from A and B, and all suffixes of S[1..i] or T[1..j] = all substrings of A or B, therefore the score of local alignment is $max_{i,j}V(i,j)$. Similar to global alignment, Smith-Waterman algorithm computes the optimal local alignment using dynamic programming. The details of the algorithm are given as following:

Basis: When $i = 0$ or $j = 0$, the best suffix of a string to align with an an empty string is the empty suffix, so we have:

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

Recurrence: When $i > 0$ and $j > 0$, the values of the table V can be computed row-wise or column-wise using previously computed values.

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

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 2.8 shows the V table. The maximum score 6 is the score of the optimal local alignment of S and T.

The path in Figure 2.8 corresponds to the optimal alignment, which is:

C_ AT_ G
CAATCG

Analysis We need to fill in all entries in the table with $n \times m$ matrix. Each entry can be computed in O(1) time. Finally, we can find the entry with the maximum value. So the time and space complexity is:

- Time: Each entry in the table can be computed in $O(1)$ time. The total time complexity is $O(nm)$.
- Space: Since all the entries in the $n \times m$ table have to be filled, the space complexity is $O(nm)$.

Similar to global alignment, we can reduce the space requirement for local alignment but not time.

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 align two sequences as a whole. The difference lies in the way it does not penalize spaces at the beginning and the end of the alignment. While global alignment does not differentiate between spaces that are sandwiched within two residues and spaces that precede or succeed a sequence. 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. Suppose we have an original sequence S and a target sequence T below:

```
S = ATCCGAA-CATCCAATCGAAGC
T = ------AGCATGCAAT------
```

The score of the alignment is 14: 8 matches (score=16), 1 delete (score=-1), 1 mismatch(score=1).

In the above alignment, we wish to disregard flanking (i.e., starting or trailing) spaces. 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.

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

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

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. Table 2.1 summarizes the neccessary changes.

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

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

For the start gaps, since we would like not to penalize them, this can be accounted for by initializing the first row or first column of the dynamic programming table to zeros. This is to say that he part of the alignment that starts with gaps in S or gaps in T is given a score of zero instead of minus value. Also, gaps at the end of the alignment should be ignored when computing the optimal score. For this, no more modifications are needed in the dynamic programming table. The optimal alignment score is now detected as the maximum value on the last row (when S is aligned with any prefix of T) or column (when T is aligned with any prefix of S).

2.5 Gap Penalty

2.5.1 Background

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 interchangeably. Here, we define gap as follows: A gap in an alignment is a maximal substring of contiguous spaces in any sequence of alignment.

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

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

Previous discussion assumes the penalty for insert/delete is proportional to the length of a gap. This assumption may not be valid in some applications such as:

- Mutation may cause insertion/deletion of a large substring. Such kind of mutation may be as likely as insertion/deletion of a single base.
- Recall that mRNA misses the introns. When aligning mRNA with its gene, the penalty should not be proportional to the length of the gaps.

Hence, it is only natural not to impose a penalty that is strictly proportional to the length of the gap. Such scheme is also preferable in situations when we expect spaces to appear contiguously, for example, when aligning mRNA with its gene.

2.5.2 General gap penalty

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 algorithm.

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

Basis: When $i = 0$ or $j = 0$, $V(i,j)$ is simply the value of the gap function g.

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

Recurrence: When $i > 0$ and $j > 0$, $V(i,j)$ can be computed by using a scoring function δ, the gap function g and the previous values of $V(i,j)$ stored in the table.

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

Analysis We need to fill in all entries in the $n \times m$ table. Each entry can be computed in $O(\max\{n,m\})$ time. So the time and space complexity is:

- Time: To compute the alignment, we need to fill all the entries in the $n \times m$ matrix, each of which can be computed in $O(\max\{n, m\})$ time. In total, we need $O(nm \max\{n, m\})$ time.
- Space: We need to allocate $O(nm)$ memory for the dynamic programming matrix.

2.5.3 Affine gap model

A biologically meaningful model for the cost of a gap is the affine gap model, in which 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 q is:

$$g(q) = h + qs \tag{2.1}$$

Before we look for the dynamic programming solution to it, we first introduce the following notations:

- V (i,j)is the score of a global optimal alignment between $S[1...i]$ and $T[1...j]$
- G (i,j) is the score of a global optimal alignment between $S[1...i]$ and $T[1...j]$ with $S[i]$ matches with $T[j]$
- F (i,j) is the score of a global optimal alignment between $S[1...i]$ and $T[1...j]$ with $S[i]$ matches with a space
- E (i,j)is the score of a global optimal alignment between $S[1...i]$ and $T[1...j]$ with $T[j]$ matches with a space

Basis: When $i = 0$ or $j = 0$, the correct value is not only based on the weight of the space (qs), but also based on the weight of "opening the gap"(g). So the basis for the recurrence equations are as follows:

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

Recurrence: When $i > 0$ and $j > 0$, we define three recurrence relations, for each of V(i, j), G(i, j), E(i, j)and F(i, j). Each can 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) - 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". Hence $V(i,j) = V(i,j-1) - h - s$.

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 simply by taking the maximum of the three. The score for the optimal alignment is $V(n,m)$. Note that the optimal alignment can be recovered by backtracking on the 4 tables.

$$\begin{aligned} V(i,j) &= max\{G(i,j), F(i,j), E(i,j)\} \\ G(i,j) &= V(i-1,j-1) + \delta(S[i], T[j]) \\ F(i,j) &= max\{F(i-1,j) - s, V(i-1,j) - h - s\} \\ E(i,j) &= max\{E(i,j-1) - s, V(i,j-1) - h - s\} \end{aligned} \tag{2.3}$$

Analysis We need to fill in 4 tables, each is of size $n \times m$. Each entry can be computed in O(1) time.

- Time: Since we need to fill in $O(nm)$ entries where each of the entries can be computed in $O(1)$ time, the time complexity is $O(nm)$.
- Space: During the computation process, there is a need to save four $n \times m$ matrices E, F, G and V. Hence, the space complexity is $O(nm)$.

2.6 All-Substring Global Alignment Problem

2.6.1 Background

Assuming the score match is 1, while the scores for unmatch, insert, delete are -1. Consider two strings $A[1..n]$ and $B[1..m]$ with $n << m$. All-substring global alignment problem tries to compute alignments between A and m^2 substrings of B, that is, compute $V(A[1..n], B[i..j])$ for all $1 \leq i \leq j \leq m$, where $V(X,Y)$ is the global alignment score between two strings X and Y. For example, in biology, scientists always compare some short sequence A (e.g. known gene or motif) with many substrings in a long sequence B(normally, a genome).
—–Known Gene A
———————————Genome B
In such a scenario, it may be good to preprocess V(A, B') for all substrings B' of B. The trivial solution is to apply Needleman-Wunsch algorithm to $A[1..n]$ and $B[i..m]$ for every $1 \leq i \leq m$, we compute $V(A[1..n], B[i..j])$ for $i \leq j \leq m$. The time complexity = $\sum_{i=1}^{m} O(n(m-i)) = O(nm^2)$, and the space complexity= $O(m^2)$ as we need to store m^2 values.

Another way to solve the all-substring global alignment problem is by F. Wei etc[13]. They show that the time can be improved from $O(nm^2)$ to $O(n^2m)$; and the space can be improved from $O(m^2)$ to $O(nm)$. Note that normally, $n << m$, so this result is much better than the trivial solution and is practical. Now let's look at the novel solution.

2.6.2 A novel solution

The idea is: We give a 'compressed' representation F_k of $V(A[1..k], B[i..j])$ for all $1 \le i \le j \le m$, and the representation F_k allows us to store $O(m^2)$ values using $O(km)$ space.

The compressed representation $F_k[i,j]$ is defined as $V(A[1..k], B[i..j]) + j - i + 1 + k$, and based on Needleman-Wunsch recursive equation, it can easily verify the following formula:

$$F_k(i,j) = \max \begin{cases} F_k[i,j-1] \\ F_{k-1}[i,j] \\ F_{k-1}[i,j-1] + 2 + \delta(A[k], B[j]) \end{cases}$$

To store $F_k[i,j]$ for all $1 \le i \le j \le m$, it requires $O(m^2)$ space. But interestingly, F_k satisfies three properties which successfully help to reduce its storage space to $O(km)$.

The first property of function F is: $F_k(i,j) \ge F_k(i,j-1)$ which implies the fact that values of F_k are monotonic increasing in each row. The proof is:

$$F_k(i,j) = \max \begin{cases} F_k[i,j-1] \\ F_{k-1}[i,j] \\ F_{k-1}[i,j-1] + 2 + \delta(A[k], B[j]) \end{cases} \ge F_k[i,j-1]$$

The second property of function F is: $F_k(i,j) \le F_k(i-1,j)$ which implies the fact that values of F_k are monotonic decreasing in each column. The proof is:

$F_k[i,j] - F_k[i-1,j] = V(A[1..k], B[i..j]) - V(A[1..k], B[i-1..j]) - 1$
Since $V(A[1..k], B[i..j]) \le 1 + V(A[1..k], B[i-1..j])$
We have $F_k[i,j] - F_k[i-1,j] \le 0$ and thus: $F_k[i,j] \le F_k[i-1,j]$.

The third property of function F is: $0 \le F_k[i,j] \le 3k$ as proved below:

Since each character in A and B contribute at least -1 score, we have:
$V(A[1..k], B[i..j]) \ge -(j - i + 1 + k)$, thus, $F_k[i,j] = V(A[1..k], B[i..j]) + (j -$

$i + 1 + k) \geq 0$
Since there are at least $|j - i + 1 - k|$ insertion or deletion and at most k matches, we have:
$V(A[1..k], B[i..j] \leq k - |j - i + 1 - k|$ and thus, $F_k[i, j] \leq k + (k + j - i + 1) - |j - i + 1 - k| \leq 3k$

In all, the properties of the F function are:

- $F_k(i, j) \geq F_k(i, j - 1)$ (Values of F_k are monotonic increasing in each row)
- $F_k(i, j) \leq F_k(i - 1, j)$ (Values of F_k are monotonic decreasing in each column)
- $0 \leq F_k[i, j] \leq 3k$

Therefore, every row i is an increasing sequence with at most 3k different values. Thus, each row can be stored in $O(k)$ space; F_k has m rows, and thus it can be stored in $O(km)$ space; Similarly, F_k also can be stored column by column.

Now let us see how we can get F_k from F_{k-1}. If we want to get F_k from F_{k-1} directly, we need to decompress F_{k-1} , which takes $O(m^2)$ time and $O(m^2)$ space. Thus we use the following indirect way to compute F_k:
Let $G_k[i, j] = max_{i+1 \leq j' \leq j}\{F_{k-1}[i, j' - 1] + 2 + \delta(A[k], B[j'])\}$.

We can show that $F_k[i, j] = max\{G_k[i, j], F_{k-1}[i, j]\}$ by the following proof:

$$F_k(i, j) = \max \begin{cases} F_k[i, j - 1] \\ F_{k-1}[i, j] \\ F_{k-1}[i, j - 1] + 2 + \delta(A[k], B[j]) \end{cases}$$

$$= \max \begin{cases} F_k[i, j - 2] \\ F_{k-1}[i, j - 1] \\ F_{k-1}[i, j - 1] + 2 + \delta(A[k], B[j]) \\ F_{k-1}[i, j] \\ F_{k-1}[i, j - 1] + 2 + \delta(A[k], B[j]) \end{cases}$$

...

$$= \max \begin{cases} max_{i \leq j' \leq j} F_{k-1}[i, j'] \\ max_{i+1 \leq j' \leq j} F_{k_1}[i, j' - 1] + 2 + \delta(A[k], B[j']) \end{cases}$$

$$= max\{G_k[i,j], F_{k-1}[i,j]\}$$

It can be easily checked that G_k satisfies the following three properties:

- $G_k(i,j) \geq G_k(i,j-1)$ (Values of G_k are monotonic increasing in each row)
- $G_k(i,j) \leq G_k(i-1,j)$ (Values of G_k are monotonic decreasing in each column)
- $0 \leq G_k[i,j] \leq 3k$

Thus, similar to F_k, G_k can also be stored compactly using $O(km)$ space, and it also can be shown G_k can be constructed from F_k in $O(km)$ time using $O(km)$ space.The algorithm for converting F_{k-1} to F_k is like:
First, we compute G_k from F_{k-1} in $O(km)$ time; Then, by computing $max\{G_k[i,j], F_{k-1}[i,j]\}$, we get F_k in $O(km)$ time. Thus F_k can be constructed from F_{k-1} in $O(km)$ time using $O(km)$ space. In summary, $F_n[i,j]$ for all $i \leq i \leq j \leq m$ can be computed in $O(n^2m)$ time using $O(nm)$ space.

References

[1] R. F. Doolittle, M. Hunkapiller, L. E. Hood, S. Devare, K. Robbins, S. Aaronson and H. Antoniades, “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] S. B. Needleman and C. D. Wunsch, “Ageneral method applicable to the seaarch for similarities in the amino acid sequence of two proteins”, *Journal of Molecular Biology*, 48:443-453, 1970.

[3] T. F. Smith and M. S. Waterman, “Identification of common molecular subsequences”, *Journal of Molecular Biology*, 147:195-197, 1981.

[4] M. Waterman, T. F. Smith and W. E. Beyer, “Some biological sequence metrics”, *Advances in Mathematics*, 20:367-387, 1976.

[5] M. O. Dayhoff, R. M. Schwartz and B. C. orcutt, “A model of evolutionary change in proteins”, *In M. O. Dayhoff (ed) Atlas of Protein Sequence and Structure*, volume 5, supplement 3, pp. 345-352, National Biomedical Research Foundation, Washington DC, 1978.

[6] J. Setubal and J. Meidanis, "Introduction to Computational Molecular Biology", *PWS Publishing Company*, 1997.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, "Introduction to Algorithms", *Prentice Hall*, Second Edition, 2001.

[8] R. B. Eggleton and R. K. Guy, "Catalan strikes again! How likely is a function to be convex?", *Math. Mag.*, 61:211-219, 1988.

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

[10] Dan Gusfield, "Algorithms on Strings, Trees and Sequences", *Cambridge University Press*, 1997.

[11] Eric S. Lander, and Michael S. Waterman, "Calculating the secrets of life: applications of the mathematical sciences in molecular biology", *National Academy Press*, p. 171, 1995.

[12] R. Webster, "Convexity", *Oxford University Press*, Oxford, England, 1995.

[13] F. Wei, W. K. Hon, and W. K. Sung, "All-Substring Alignment Problems", *COCOON*, 2003.

[14] Riordan et al., "Identification of the Cystic Fibrosis Gene: Cloning and Characterization of Complementary DNA(in Cysic Fibrosis: Cloning and Genetics)"", *Science*, 245, 1066-1073, 1989.