Lecture 2: November 6, 2000

Lecturer: Ron Shamir *Scribe: Ofira Grebenshuk and Alex Feldman*[1]

2.1 Pairwise Alignment

The lecture introduces the problem of comparing two sequences while allowing certain mismatches between the two. The problem will be referred to as *(pairwise) sequence alignment.* or *inexact matching*.

We first present the problem and provide biological motivation. We then define similarity and difference between sequences and present algorithms for computing them, with analysis of the complexity of these algorithms.

The algorithms use the *dynamic programming* technique. For each algorithm the following information will be given:

- Intuitive explanation of the recursive process.
- Formal definition of the recursive process.
- Discussion of the complexity.

2.1.1 Problem Definition and Biological Motivation

Motivation

A large variety of the biologically motivated problems in computer science primarily involve sequences or strings. For instance:

- Reconstructing long sequences of DNA from overlapping sequence fragments.
- Determining physical and genetic maps from probe data under various experiments protocols.
- Storing, retrieving and comparing DNA sequences.
- Comparing two or more sequences for similarities.
- Searching databases for related sequences and subsequences.

[1]Based in part on a scribe of Eti Ezra and Guy Gelles November 13th, 1998

- Exploring frequently occurring patterns of nucleotides.
- Finding informative elements in protein and DNA sequences.

Many of these research problems aim at learning about functionality or the structure of protein without performing any experiments and actually without having to physically construct the protein itself. The basic idea is that similar sequences produce similar proteins. Thus, in order to predict the characteristics of a protein using only its sequence data, we can use the structure/function information on known protein with similar sequences available in databases.

For instance, when considering protein folding, it usually suffices that two protein sequences are identical at 25% of their positions for their three dimensional structures to be almost identical. Classical example is the establishment of an association between cancer and uncontrolled cell growth [1]. This discovery was enabled by comparing the sequence of a cancer associated gene against the sequence of proteins which had already been known as influencing the cell growth. The correlation between these two sequences was very high, proving the connection between cancer and cellular growth.

2.1.2 Similarity and Difference

The resemblance of two DNA sequences taken from different organisms can be explained by the theory that all contemporary genetic material has one common ancestral DNA. According to this theory, during the course of evolution mutations occurred, creating differences between families of contemporary species. Most of these changes are due to local mutations, each modifying the DNA sequence at a specific manner. These local modifications between nucleotide sequences, or more generally, between strings over an arbitrary alphabet can be either:

- *Insertion* - an insertion of a letter or several letters to the sequence.
- *Deletion* - deleting a letter (or more) from the sequence.
- *Substitution* - replacing a sequence letter by another.

Insertion and deletion are the reverse of one another: given two sequences, if the insertion of a character (or more) into one yields the other, then equivalently its deletion from the latter sequence transforms it to the first one. Due to this reciprocity between insertion and deletion, they are usually called *indel* for short.

The notion of *distance* derives its definition from the concept of mutations by assigning weights to each mutation: Given two sequences, the *distance* between them is the minimal sum of weights for a set of mutations transforming one into the other.

The notion of *similarity* derives its definition from the concept of one ancestral ancient DNA: by assigning weights corresponding to resemblance. Given two sequences the *similarity* between them is the maximal sum of such weights.

Nomenclature

Its impotrant to mention that Biology and Computer Science use different nomenclature. Here is the table that compares the notations:

BIOLOGY	**COMPUTER SCIENCE**
- Sequence	- String,word
- Subsequence	- Substring(contiguous)
- N/a	- Subsequence
- N/a	- Exact matching
- Alignment	- Inexact matching

Explanation: subsequence (in the computer science) is a non contiguous segment of a sequence. We will use the biology nomenclature.

Simplest model: Edit Distance

Definition The edit distance between two sequences is the minimal number of edit operations (insertions, deletions and substitutions) needed to transform one sequence into the other.

Each operation is given a score (weight). Usually, insert and delete (indel) operations are given the same score, and using alignment algorithms, we search for the minimal scoring (or the maximum negative scoring), representing the minimal number of edit operations.

Most of the changes to DNA during evolution are due to the three common local mutations: insertion, deletion and substitution. Therefore the edit distance can be used to roughly measure the number of DNA replications that occurred between two DNA sequences.

Example Given two sequences `a c c t g a` and `a g t t a`, the minimal number of edit noperations required to transform one into another is 3:

```
a c c t g a        a - c c t g a
a g c c t g a
a g c t t g a      a g c t t - a
a g c t t a
```

First, indel operation (inserting **g**) is applied on the sequence `a c t g a`. As a result the sequence `a g c c t g a` is obtained. Then, iside this sequence the substitution of **c** by t is performed and the sequence `a g c t t g a` is received. Finally, to get the sequence `a g c t t a` one should apply another indel operation (deleting **g**) on sequence `a g c t t g a`.

Models for Alignment

In this lecture we consider four of alignment problems all of which are biologically motivated:

Problem 2.1 Global Alignment
INPUT: Two sequences S and T of roughly the same length.
QUESTION: What is the best *difference* (or *similarity*) between them? Find the best alignment.

Problem 2.2 Local Alignment
INPUT: Two sequences S and T.
QUESTION: What is the maximum similarity (minimum difference) between a subsequence of S and a subsequence of T? Find most similar subsequences.

Problem 2.3 Ends free alignment(End space-free alignment)
INPUT: Two sequences S and T (possibly of different length).
QUESTION: Find a best alignment between subsequences of S and T when at least one of these subsequences is a prefix of the original sequence and one (not necessarily the other) is a suffix.

Problem 2.4 Gap penalty

Definition *A gap* is the *maximal* contiguous run of spaces in a single sequence within a given alignment. *The length of a gap* is the number of *indel* operations on it. *Gap penalty function* is a function that measures the cost of a gap as a (nonlinear) function of its length.

INPUT: Two sequences S and T of different length.
QUESTION: What is the similarity between the two sequences, given the gap penalty function?

2.1.3 Global Alignment

Definition Informally, an *alignment* of two sequences S and T is obtained by first inserting chosen spaces, either into or at the ends of S and T so the length of the sequences will match, and then placing the two resulting sequences one above the other so that every character or space in one of the sequences is matched to a unique character or a unique space in the other sequence.

Example Given a sequence "`acgctttg`" and a sequence "`catgtat`", one possible alignment would be:

```
a c - - g c t t t g
- c a t g - t a t -
```

A more useful than the general case is the following problem:

Problem 2.5 INPUT: Two sequences $S = s_1 ... s_n$ and $T = t_1 ... t_n$ (n and m are approximately the same)
QUESTION: Find an optimal alignment according to the alignment quality (or scoring).

Notation Let $\sigma(a, b)$ be the *score* (weight) of the alignment of character a with character b (including spaces).

Notation Let $V(i, j)$ be the *optimal score* of the alignment of $S_1 \ldots S_i$ and $T_1 \ldots T_j$ ($0 \leq i \leq n, 0 \leq j \leq m$).

Lemma 2.1 *V(A,B) has the following properties:*

Base conditions:

$$V(i,0) = \sum_{k=0}^{i} \sigma(S_k, -)$$

$$V(0,j) = \sum_{k=0}^{j} \sigma(-, T_k)$$

Recurrence relation: for $1 \leq i \leq n, 1 \leq j \leq m$:

$$V(i,j) = \max \begin{cases} V(i-1, j-1) + \sigma(S_i, T_j) \\ V(i-1, j) + \sigma(S_i, -) \\ V(i, j-1) + \sigma(-, T_j) \end{cases}$$

Proof
Base condition:
The only way to align the first i elements of the sequence S with zero elements of the sequence T is to align each of the elements with a space in the sequence T. The score for that operation is by definition $\sigma(S_i, -)$ for each of the i elements and $V(i, 0) = \sum_{k=0}^{i} \sigma(S_k, -)$ for the total sum.

Similarly, the expression $V(0, j) = \sum_{k=0}^{j} \sigma(-, T_k)$ follows from matching the first j elements of T with i blanks in sequence S.
Recurrence relation:
Let us consider an optimal alignment of $S_1 \ldots S_i$ and $T_1 \ldots T_j$. We shall distinguish between three cases according to the three possible scoring for the three operations are:

- **Aligning S_i with T_j**: The score in this case is the score $\sigma(S_i, T_j)$ of aligning S_i with T_j plus the score of aligning $i-1$ elements of S with $j-1$ elements of T, namely, $V(i-1, j-1) + \sigma(S_i, T_j)$
- **Aligning S_i with a space character in sequence T**: The score in this case is the score $\sigma(S_i, -)$ of aligning S_i with indel plus the score of aligning the previous $i-1$ elements of S with j elements of T (Since the space is not an original character of T), $V(i-1, j) + \sigma(S_i, -)$
- **Aligning T_j with a space character in sequence S**: Similar to the previous case, the score will be $V(i, j-1) + \sigma(-, T_j)$

■

Tabular computation of optimal alignment

The problem can be evaluated systematically using a tabular computation. In this approach, we compute $V(i, j)$ for the all possible values of i and j. We start from smaller i,j and increase them, filling the table in a row-wise manner. We store these values in table of size $(n+1) \times (m+1)$. Finally $V(n, m)$ is the required alignment score.

The following pseudo code describes the algorithm:

```
for i=1 to n do
begin
    for j=1 to m do
    begin
        Calculate V(i,j) using V(i-1,j-1), V(i,j-1), V(i-1,j)
    end
end
```

Example Figure 2.1 illustrates a snapshot at some point during the computation. In this example and the following one the value of σ is -1 for a mismatch and 2 for a match.

The Traceback

One way to traceback the alignments is to establish pointers in the cells of the table as the values are computed. The direction of the pointer in cell (i, j) indicates the value of which cell was used when $V(i, j)$ was computed.

i \ j		0	1	2	3	4	5	
			c	a	t	g	t	← T
0		0	-1	-2	-3	-4	-5	
1	a	-1	-1	1				
2	c	-2						
3	g	-3						
4	c	-4						
5	t	-5						
6	g	-6						

↑ S

Figure 2.1: Snapshot of computing the table. The figure demonstrates some initial stages of running an algorithm for finding global alignment. In the first column and the first row one can see results of the calculations of $V(i,-)$ and $V(-,j)$. In the second row one can see the propagation of the algorithm.

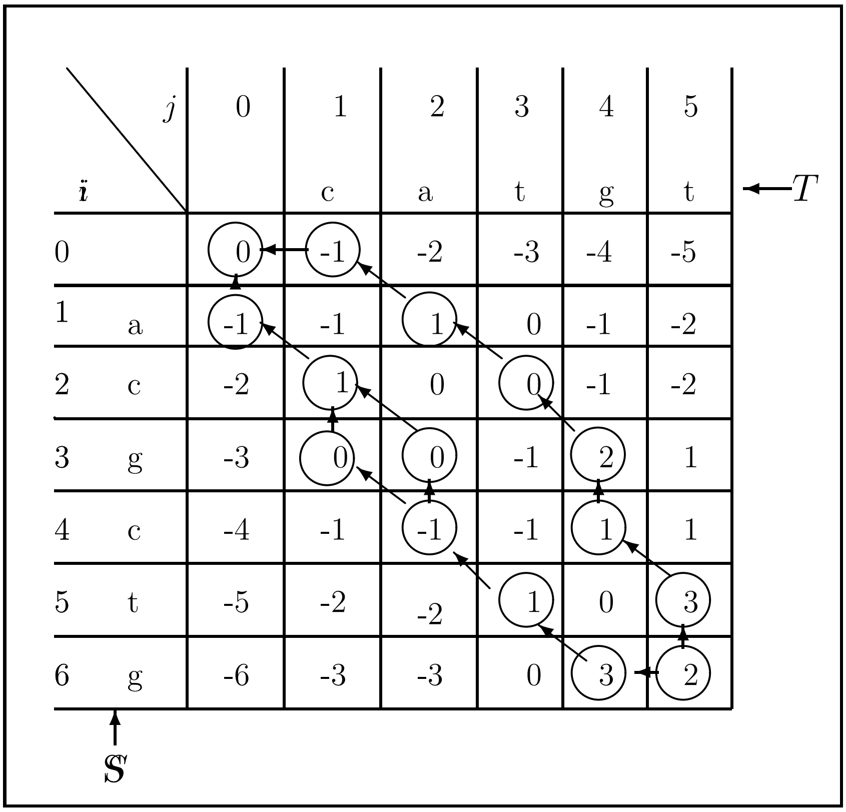

Figure 2.2: Backtracing the alignment. For each cell its outgoing arrows point to the cells from which algorithm could arrive to the current cell. In this figure one can see 3 possible paths that represent alignments that give best score.

Theorem 2.2 *The time complexity of the algorithm is $O(nm)$. Space complexity is $O(n+m)$, if only $V(S,T)$ is required and $O(mn)$ for the reconstruction of the alignment.*

Proof

- **Time complexity.** When computing the value of a specific cell (i,j) only cells $(i-1,j-1)$, $(i,j-1)$ and $(i-1,j)$ are examined, along with two characters S_i and T_j. Hence, to fill in one cell it takes a constant number of cell examinations and comparisons. There are $n \times m$ cells in the table. So the time complexity is also $O(nm)$.

- **Space complexity.** Using the algorithm, computing the value of cell (i,j) involves one cell $(i-1,j)$ in row j and two cells $((i-1,j-1)$ and $(i,j-1))$ in the previous row $j-1$. Since the computation is performed one row at a time, when computing the values in row k only row $k-1$ has to be stored, using, $O(n+m)$ space. In order to reconstruct the alignment from the recursion, pointers must be set to allow the back-tracing. Hence, the space complexity is $O(nm)$.

■

2.1.4 Global Alignment in linear space

The backtracing algorithm requires the entire matrix to be saved in memory. The space complexity consequently increases to $O(nm)$. Hirschberg [4] developed a more practical space-reduction method for solving dynamic programming problems that reduces the required space from $O(nm)$ to $O(m)$ (for $m < n$).

Notation $V^r(i,j)$ will denote an optimal alignment value of last i characters in sequence S against last j characters in sequence T.

Lemma 2.3

$$V(n,m) = \max_{0\le k\le m}\{V(\frac{n}{2},k) + V^r(\frac{n}{2},m-k)\}$$

Proof [3] (chapter 12) For any fixed position k' in T, there is an alignment of S and T consisting of an alignment of $S_1 \dots S_{\frac{n}{2}}$ and $T_1 \dots T_{k'}$ followed by a disjoint alignment of $S_{\frac{n}{2}+1} \dots S_n$ and $T_{k'+1} \dots T_m$. By definition of V and V^r, the best alignment of the first type has value $V(\frac{n}{2},k')$ and the best alignment of the second type has value $V^r(\frac{n}{2},m-k')$, so the combined alignment has value $V(\frac{n}{2},k') + V^r(\frac{n}{2},m-k') \le V(n,m)$. Since this argument holds for any k', it follows that $\max_k[V(\frac{n}{2},k) + V^r(\frac{n}{2},m-k)] \le V(n,m)$.

Conversely, for an optimal alignment of S and T, let k' be the right-most position in T that is aligned with a character at or before position $\frac{n}{2}$ in S. Then the optimal alignment of S and T consists of an alignment of $S_1 \dots S_{\frac{n}{2}}$ and $T_1 \dots T_{k'}$ followed by an alignment of

$S_{\frac{n}{2}+1} \dots T_n$ and $T_{k'+1} \dots T_m$. Let the value of the first alignment be denoted p and the value of the second alignment be denoted q. Then p must be equal to $V(\frac{n}{2}, k')$, for if $p < V(\frac{n}{2}, k')$ we could replace the alignment of $S_1 \dots S_{\frac{n}{2}}$ and $T_1 \dots T_{k'}$ with the alignment of $S_1 \dots S_{\frac{n}{2}}$ and $T_1 \dots T_{k'}$ that allows value $V(\frac{n}{2}, k')$. That would create an alignment of S and T whose value is larger than the claimed optimal. Hence, $p = V(\frac{n}{2}, k')$. By similar reasoning, $q = V^r(\frac{n}{2}, m-k')$. So $V(n,m) = V(\frac{n}{2}, k') + V^r(\frac{n}{2}, m-k') \leq \max_k[V(\frac{n}{2}, k) + V^r(\frac{n}{2}, m-k)]$. Having shown both sides of the inequality, we conclude that $V(n,m) = max_k[V(\frac{n}{2}, k) + V^r(\frac{n}{2}, m-k)]$. ■

The Algorithm:

---

1. Compute $V(A,B)$ while saving the values of the $\frac{n}{2}$-th row. Denote $D(A,B)$ as the *Forward Matrix* F.

2. Compute $V(A^r, B^r)$ while saving the $\frac{n}{2}$-th row. Denote $D(A^r, B^r)$ as the *Backward Matrix* B.

3. Find the column k^* so that the crossing point $(\frac{n}{2}, k^*)$ satisfies:

$$F(\tfrac{n}{2}, k^*) + B(\tfrac{n}{2}, m-k^*) = F(n,m)$$

4. Now that k^* is found, recursively partition the problem to two sub problems:

   (i) Find the path from $(0,0)$ to $(\frac{n}{2}, k^*)$.
   (ii) Find the path from (n,m) to $(\frac{n}{2}, m-k^*)$.

Lemma 2.4 *Time complexity of* Hirschberg*'s algorithm is $O(nm)$.*

Proof Time complexity
Let $T^*(n,m)$ be the time to find the value of a $n \times m$ problem. Let $T(n,m)$ be the time to find the path (solution) of a $n \times m$ problem.

$$T^*(n,m) = 2T(n,m) + T^*(\frac{n}{2}, k^*) + T^*(\frac{n}{2}, m-k^*) \tag{2.1}$$

$T(n,m) = cnm$, therefore $T^*(n,m) = 4T(n,m) = 4cnm$. according to 2.1: $4cnm = 2cnm + 4c(\frac{n}{2} \times k^*) + 4c(\frac{n}{2} \times (m-k^*))$. Therefore, the time complexity will remain $O(nm)$. ■

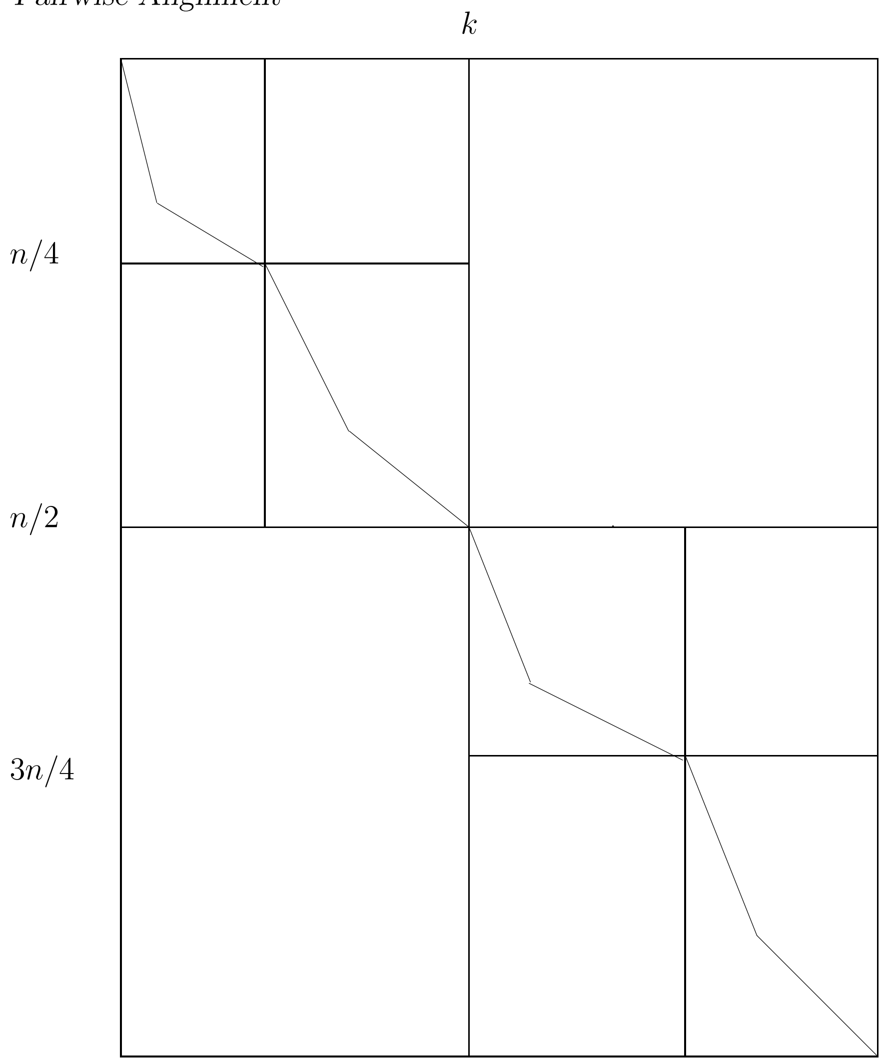

Figure 2.3: “Divide and Conquer” Algorithm (*Hirschberg,[4]*).

Lemma 2.5 *Space complexity of* Hirschberg*'s algorithm is* $O(m)$

Proof Space Complexity
Each dynamic programming computation requires storing one additional row (middle one) which can be discarded once the middle point is found. Therefore the space complexity is $O(m)$. ■

2.1.5 Alignment Graph

It is often useful to represent dynamic programming solutions of sequence problems in terms of a weighted graph.

Definition Given two sequences S and T of lengths n and m respectively. An *alignment graph* is a directed graph $G = (V, E)$ on $(n+1) \times (m+1)$ nodes, each labeled with a distinct pair (i, j) $(0 \leq i \leq n, 0 \leq j \leq m)$, with the following weighted edges:

1. $((i, j), (i+1, j))$ with weight $\sigma(S_{i+1}, -)$
2. $((i, j+1))$ with weight $\sigma(-, T_{j+1})$
3. $((i, j), (i+1, j+1))$ with weight $\sigma(S_{i+1}, T_{j+1})$

A path from node $(0, 0)$ to node (n, m) in the alignment graph corresponds to an alignment and its total weight is the alignment score. Our goal is to find the heaviest path from node $(0, 0)$ to node (n, m).

This *alignment graph* is used to map the problem of optimal alignment into the world of graphs, opening the door for new and exciting algorithms.

2.1.6 Local Alignment

In many applications two sequences may not be highly similar as a whole, but many contain regions that are highly similar. The task is to find and extract a pair of regions, one from each of the two given sequences, that exhibit high similarity. This is called the *local alignment* or *local similarity* problem and formally is defined below:

Definition Given two sequences S and T, *local alignment* problem is defined as the problem of finding the subsequences α and β of S and T respectively, whose *similarity* (optimal global alignment) is maximum over all such pairs of subsequences.

Example Consider the two sequences:
S = `g g t c t g a g`
T = `a a a c g a`
If we give each match a value of 2 and each mismatch a value of -1, then the two subsequences: $\alpha =$ `ctga` and $\beta =$ `c-ga` of S and T respectively have the optimal alignment.

Motivation

In many biological applications local similarity is far more meaningful than global similarity. This is particularly true when long stretches of non-coding DNA are compared, since only small regions within those sequences may be related. When comparing protein sequences, local alignment is also crusial because proteins from very different families often share the same structural or functional subunits, and local alignment is an appropriate tool for searching such module.

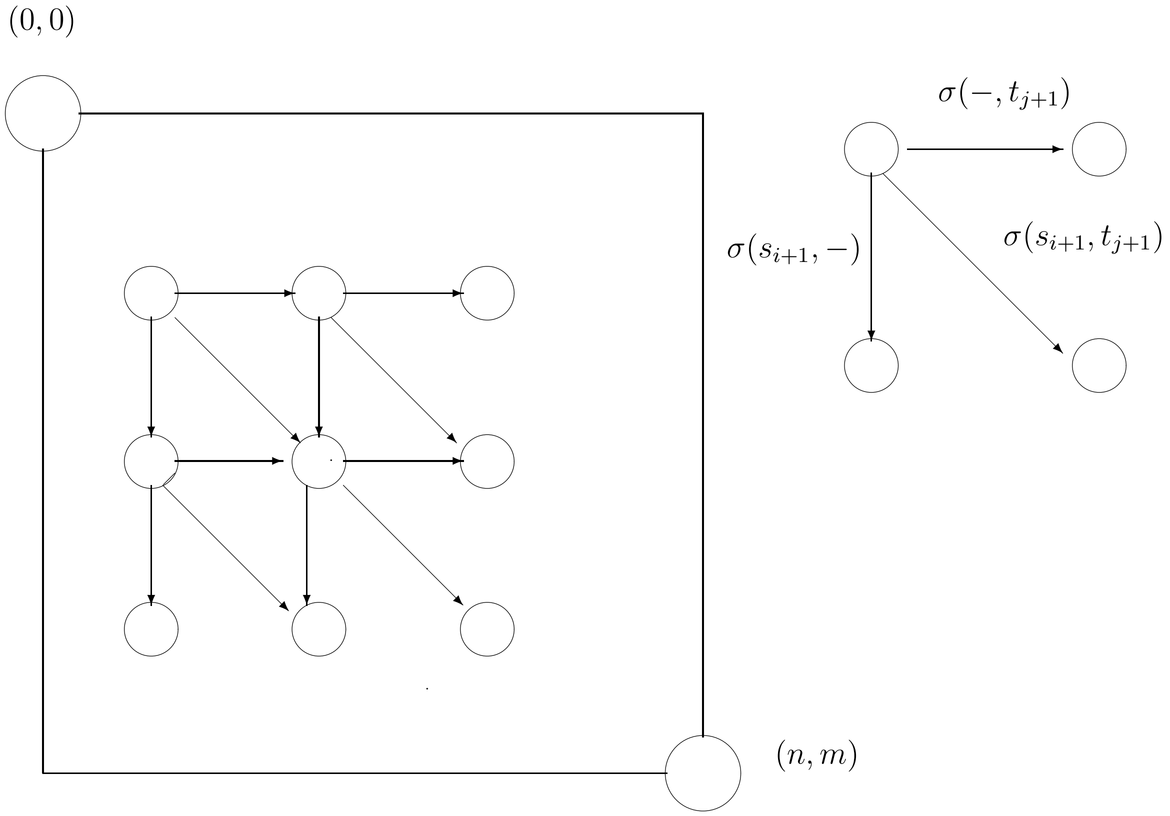

Figure 2.4: Alignment Graph. The alignment graph is a direct acyclic graph. On the right side of the figure one can see the segment of the alignment graph. It represents the three possibilities at each stage of the algorithm.

Computing local alignment

Given a pair of indices $i \leq n$ and $j \leq m$, the local suffix alignment problem is to find a (possibly empty) suffix α of $S_{1...i}$ and a (possibly empty) suffix β of $T_{1...j}$ such that the score of their alignment is the maximum over all scores of alignments of suffixes of $S_{1...i}$ and $T_{1...j}$. We use $V(i,j)$ to denote the value of the optimal local suffix alignment for a given pair i,j of indices.

We choose the weights of the editing operations as:

$$\sigma(x,y) = \begin{cases} \geq 0 & \text{if } x,y \text{ match} \\ \leq 0 & \text{if } x,y \text{ do not match or one of them is -} \end{cases}$$

The algorithm needs to:

1. Find maximum similarity between suffixes of $S_{1...i}$ and $T_{1...j}$.
2. Discard the prefixes $S_{1...i}, T_{1...j}$ whose similarity is ≤ 0, and therefore decreases the overall similarity.
3. Find the best indices i^*, j^* of S and T respectively after which the similarity only decreases.

Note that any extension of the optimal solution either to the right of to the left decreases the overall similarity.

Recursive definition: The base condition will be: $V(i,0) = 0$ and $V(0,j) = 0$ $\forall i,j$ since we can always choose an empty suffix.

For $i > 0$ and $j > 0$ the proper recurrence for $V(i,j)$ is

$$V(i,j) = \max\{0, V(i-1,j-1) + \sigma_{(S_i,T_j)}, V(i,j-1) + \sigma_{(-,T_j)}, V(i-1,j) + \sigma_{(S_i,-)}\}$$

Compute i^*, j^* so that:

$$V(i^*,j^*) = \max_{1 \leq i \leq n, 1 \leq j \leq m} V(i,j)$$

Observe that the recurrence for computing local suffix alignment is almost identical to the one used for computing global alignment. The only difference is the inclusion of zero in the case of local suffix alignment. In both global alignment and local suffix alignment of prefixes $S_{1...i}$ and $T_{1...j}$, the terminating characters of any alignment are specified, but in the case of local suffix alignment, any number of initial characters can be ignored.

The zero in the recurrence implements this, 'restarting' the recurrence. Adding 0 to the maximization makes sure that negative prefixes are discarded from the computation.

Adding the '0' to the constraint only handles mismatched prefixes, there's still a need to determine, when should a computation of a transformation be stopped, so that the similarity value will not decrease. Therefore, after computing the table of $V(i,j)$ values, and there's a need to search for a cell with the maximal value and ignore all table entries from that point on.

Example Figure 2.5 illustrates the calculation of the $n \times m$ entries table for the two sequences taken σ as 2 for match and -1 for mismatch.

As usual, pointers are created while filling in the values of the table. After cell (i^*, j^*) is found, the subsequences α and β giving the optimal local alignment of S and T are found by tracing back the pointers from cell (i^*, j^*) until reaching an entry (i', j') that has value zero. Then the optimal local alignment subsequences are $\alpha = S_{i' \dots i^*}$ and $\beta = T_{j' \dots j^*}$. As it seems from here, space complexity will be $O(mn)$, we will show that only $O(m)$ space is needed:

Lemma 2.6 *The optimal local alignment of two sequences S and T can be computed in linear space.*

Proof The optimal local alignment of S and T identifies subsequences α and β whose global alignment has maximum value over all pairs of subsequences. Hence, if α and β can be found using only linear space, then their actual alignment can be found in linear space, using Hirschberg's method for global alignment. The score of the optimal local alignment is found in cell i^*, j^*. Those indices specify the terminating points of the sequences α and β. The values in each row can be computed in a row wise fashion and the algorithm must store values for only two rows at a time. Hence, the end positions (i^*, j^*) can be computed in linear space. To find the starting position of the two subsequences, the algorithm can execute the reverse dynamic programing using linear space (the details are left as an exercise). ■

Complexity:

- **Time complexity.** Since it takes constant number of operation per cell to compute $V(i,j)$, it takes only $O(mn)$ time to fill in the entire table. The search for $V(i*, j*)$ requires only $O(nm)$ time as well. Hence the total time complexity is $O(nm)$.
- **Space complexity.** As shown in lemma local alignment, the space complexity is $O(m)$.

2.1.7 End free-space alignment

In this variant, any indel operations at the end or the beginning of the alignment contribute a weight of zero, no matter what weight other spaces contribute.

Example Consider the alignment:
S = - - c a c - t g t a c
T = g a c a g t t g - - -

i \ j		0	1	2	3	4	5	6	
			x	x	x	c	d	e	T
0		0	0	0	0	0	0	0	
1	a	0	0	0	0	0	0	0	
2	b	0	0	0	0	0	0	0	
3	c	0	0	0	0	2	1	0	
4	x	0	2	2	2	1	1	0	
5	d	0	1	1	1	1	3	2	
6	e	0	0	0	0	0	2	5	
7	x		2	2	2	1	1	4	
	S								

Figure 2.5: Finding local alignment

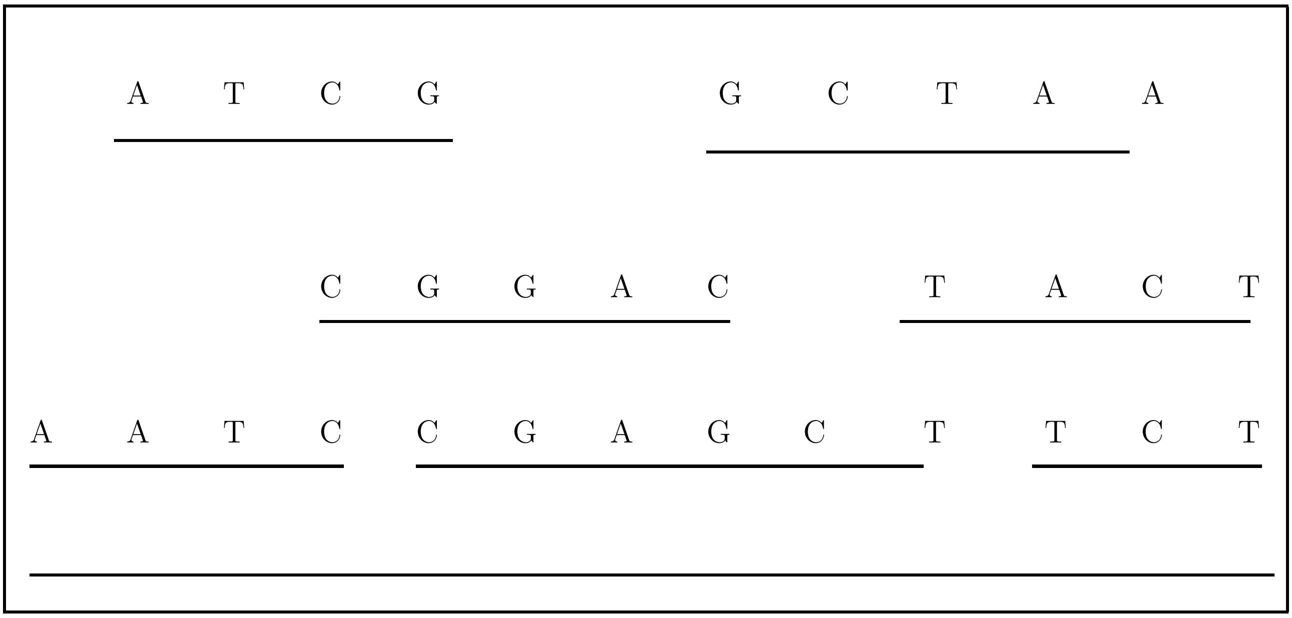

Figure 2.6: sequence assembly

The two leading spaces at the left end of the alignment are free, as well as the three trailing spaces at the right end.

Motivation

One example where end-spaces should be free is in the shotgun sequence assembly procedure. In this problem, one has a large set of partially overlapping subsequences that come from many copies of one original but unknown DNA sequence. The problem is to use comparisons of pairs of subsequences to infer the correct original sequence. Two random subsequences from the set are unlikely to have nearby starting positions along the original sequence, and this is reflected by a low end-space free alignment score for those two subsequences. But if two subsequences do overlap in the original sequence, then an alignment may be between suffix of one to a prefix of the other with only a small number of spaces and mismatches. This overlap is detected, given a high score by an end-space free weighted alignment. Similarly, the case where one subsequence contains another can be detected in this way. See figure 2.6 for illustration.

When comparing two sequences, it is not obvious how to place the two sequences, so that the *similarity* between the two will be maximal. One possibility, denoted by the *ends free problem* is to disregard leading and trailing indel operations (in the usual similarity strategy, all indel operations reduce the similarity).

To implement this we will change the algorithm presented for the *global alignment* problem, as follows:

- Set initial conditions:

$$\begin{aligned} V(i,0) &= 0 \qquad \text{for } 1 \le i \le n \\ V(0,j) &= 0 \qquad \text{for } 1 \le j \le m \end{aligned}$$

- Use the same recurrence for $1 \le i \le n$, $1 \le j \le m$

$$V(i,j) = \max \begin{cases} V(i-1,j-1) + \sigma(S_i,T_j) \\ V(i-1,j) + \sigma(S_i,-) \\ V(i,j-1) + \sigma(-,T_j) \end{cases}$$

- Instead for looking at $V(n,m)$ the algorithm will search for i^* and j^* so that:

$$\begin{aligned} V(n,i^*) &= \max_i V(n,i) \\ V(j^*,m) &= \max_j V(j,m) \end{aligned}$$

- The similarity will be defined as:

$$V(S,T) = \max\{V(n,i^*), V(j^*,m)\}$$

Looking for i^* means searching for a cell in the last row of the table, produced while computing $V(n,m)$. Looking for j^* means searching for a cell in the last column of the same table. This eliminates trailing indel operations. Leading indel operations will not be taken into account due to the changes in the initial conditions.

Complexity:

- **Time complexity.** Computing the matrix takes $O(nm)$. Finding j^* and i^* takes $O(n+m)$. Therefore the total complexity remains $O(nm)$.

- **Space complexity.** Computing the matrix takes $O(n+m)$ space. Computing the maximizing values i^*, j^* requires the last row and column to be saved, which is also $O(n+m)$. Therefore the total complexity remains $O(n+m)$.

2.1.8 Gap Penalty

Up until now the central elements used to measure the score of an alignment have been matches, mismatches and spaces. Now we introduce another important element, *gaps*. Gaps help create alignments that better conform to underlying biological models and more closely fit patterns that one expects to find in meaningful alignments. The idea is to take in account the number of contiguous gaps and not only the number of spaces when calculating an alignment mark This section presents a gap penalty model for evaluating the weight of a sequence of consecutive indel operations. The model states that consecutive indel operations have different total weight than simply the sum of their weights.

Definition A *gap* is any maximal, consecutive run of spaces in a single sequence of a given alignment.

Example Consider the alignment:
S = a t t c - - g a - t g g a c c
T = a - - c g t g a t t - - - c c
This alignment has four gaps containing a total of eight spaces. The alignment would be described as having seven matches, no mismatch, four gaps and eight spaces.

Definition The *length of the gap* will be the number of indel operations in it. The number of gaps in the alignment will be denoted as #gaps.

Motivation

The concept of a gap in an alignment is important in many biological application, because the insertion or deletion of an entire subsequence often occurs as single mutational event. Moreover, many of these single mutational events can create gaps of quite varying sizes. At the protein level, two protein sequences might be relatively similar over several intervals but differ in intervals where one contains a protein subunit that the other does not.

One concrete illustration of the use of gaps in the alignment model comes from the problem of cDNA matching [2] (chapter 11). In this problem, one sequence is much longer than the other, and the alignment best reflecting their relationship should consist of a few regions of very high similarity interspersed with 'long' gaps in the shorter sequence. Note that the matching regions can have mismatches and spaces, but these should amount only to a small fraction of the region.

Recall that an RNA molecule is transcribed from the DNA of a gene. That RNA transcript is a complement of the gene in DNA where each A in the gene is replaced by U in the RNA, each T is replaced by A, each C by G, and each G by C. Moreover, the RNA transcript spans the entire gene, introns as well as exons. Then in a process that is not compleply understood, each introns-exon boundary in the transcript is located, the RNA regions corresponding to the introns are spliced out, and the RNA regions corresponding to exons are concatenated. The resulting RNA molecule is called the *messenger* RNA (*mRNA*): it leaves the cell nucleus and is used to create the protein it encodes.

Each cell (usually) contains a copy of all the chromosomes and hence, of all the genes of the entire individual, yet in each specialized cell (a liver cell for example) only a small fraction of the genes are expressed. That is, only a small fraction of the proteins encoded in the genome are actually produced in that specialized cell. A standard method to determine which proteins are expressed in the specialized cell line, and to hunt for the location of the encoding genes, involves capturing the mRNA in that cell after it leaves the cell nucleus. That mRNA is then used to create a DNA sequence complementary to it. This sequence is called *cDNA* (complementary DNA). Compared to the original gene, the cDNA sequence consists only of the concatenation of exons in the gene. After cDNA is obtained, the problem is to determine where the gene associates with that cDNA resides, and it becomes one of aligning the cDNA sequence against the longer DNA sequence in a way that reveals the exons.

Gap penalties types

- Constant solved in $O(nm)$ time. (see below)
- Affine solved in $O(nm)$ time. (see below)
- Convex solved in $O(nm \log m)$ time. ([2])
- Arbitrary solved in $O(nm^2 + n^2m)$ time. (left as an exercise)

Constant gap weight model

The simplest choice is the *constant* gap weight, where each individual space is free, and each gap is given a weight of W_s independent of the number of spaces in the gap. Letting σ denote the weights of match and mismatch only ($\sigma(x, -) = \sigma(-, x) = 0$ for every character x). Thus we have to find an alignment that maximzes:

$$\Sigma\sigma(S'_i, T'_i) + W_g \times \#gaps$$

where S' and T' represent S and T after inserting space. A generalization of the constant gap weight model is to add a weight W_s for each space in the gap. In this case, W_g represents the cost of starting a gap, and W_s represents the cost of extending the gap by one space.This leads us to the *affine gap weight model*. This is called affine gap weight model because the weight contributed by a single gap of length q is given by the affine function $W_g + qW_s$. The constant gap weight model is simply the affine model with $W_s = 0$. Thus we have to find an alignment that maximizes:

$$\Sigma\sigma(S'_i, T'_i) + W_g \times \#gaps + W_s \times \#spaces$$

while S' and T' represent S and T after inserting space and $\sigma(x, -) = \sigma(-, x) = 0$ for every character x.

It has been suggested that some biological phenomena are better modeled by a gap weight function where each additional space in a gap contributes less to the gap weight than the preceding space. In other words, a gap weight that is a *convex*, but not affine function of its length. An example is the function $W_g + \log q$, where q is the length of the gap. Finally, the most general gap weight that might be considered is the *arbitrary gap weight*, where the weight of a gap is an arbitrary function $\omega(q)$ of its length q. The constant, affine and convex weight models arerestricted cases of the arbitrary weight model.

Affine gaps penalty

To align sequences S, T, consider as usual 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.

Notation We will use $G(i, j)$ to denote the maximum value of any alignment of type 1, $E(i, j)$ as the maximum value of any alignment of type 2 and $F(i, j)$ as the maximum value of any alignment of type 3. We finally define$V(i,j)$ as the maximum value of the three terms $E(i,j)$, $F(i,j)$, $G(i,j)$.

Hence the base conditions are:

$$\begin{aligned} V(i,0) &= E(i,0) = W_g + iW_s \\ V(0,j) &= F(0,j) = W_g + jW_s \end{aligned}$$

and the recursive computation of $V(i,j)$ will be:

$$V(i,j) = \max\{E(i,j), F(i,j), G(i,j)\}$$

while

$$\begin{aligned} G(i,j) &= V(i-1,j-1) + \sigma(S_i, T_j) \\ E(i,j) &= \max\{E(i,j-1) + W_s, V(i,j-1) + W_g + W_s\} \\ F(i,j) &= \max\{F(i-1,j) + W_s, V(i-1,j) + W_g + W_s\} \end{aligned}$$

The optimal value alignment is the maximum value in the nth row or m column.
Complexity

- **Time complexity.** As before $O(nm)$, as we only compute four matrices instead of one.
- **Space complexity.** There's a need to save four matrices (for E, F, G, and V respectively) during the computation. Hence, $O(nm)$ space is needed, for the trivial implementation.

2.1.9 Other Gap Penalty Models

Convex Gap Penalty Model

- Each additional space in a gap contributes less to the gap weight.
- The motivation of its usage is biological behaviour.
- Example: $W_g \log(q)$, where q is the length of the gap.
- The problem is solvable in $O(nm\log(m))$ time

Arbitrary Gap Penalty Model

- The gap weight function can be anything (this is the most general case).
- Weight of a gap is an arbitrary function of its length $w(q)$.
- The problem is solvable in $O(nm(m+n))$ time.

2.1.10 Longest Common Non Contiguous Subsequence

Definition *Non contiguous subsequence* is defined as a subset of the characters of sequence S arranged in their original "relative" order. Formally: a non contiguous subsequence of sequence S of length n is specified by a list of indices $i_1 < i_2 < i_3 < \ldots < i_k$, for some $k \leq n$. The non contiguous subsequence specified by this list of indices is the sequence $S_{i_1} S_{i_2} \ldots S_{i_k}$.

Definition Given two sequences S and T, a **common subsequence** is a subsequence that appears both in S and T. The **longest common non contiguous subsequence problem** is to find a longest non contiguous subsequence common to both S and T.

The problem of Longest Common Non Contiguous Subsequence can be modeled and solved using the optimal alignment algorithm with the following scoring:

$$\begin{aligned} \sigma(x,y) &= \begin{cases} 1 & x = y \\ 0 & x \neq y \end{cases} \\ \sigma(x,-) = \sigma(-,y) &= 0 \end{aligned}$$

Or, directly compute $V(n,m)$ with:

$$\begin{aligned} V(i,0) = V(0,j) &= 0 \\ V(i,j) &= \max \begin{cases} V(i-1,j-1) + \sigma(S_i, T_j) \\ V(i-1,j) \\ V(i,j-1) \end{cases} \end{aligned}$$

Each character in the sequence S can be align with the same character in the sequence T or with a space in T (in this case no substitution is done). Since the goal is to find maximum length, character matched are valued as '1', while a space match is valued '0'.

2.1.11 Polymerase Chain Reaction

It is possible to replicate subsequences of a DNA sequence starting at almost any point as long as we know a small number of the nucleotides, appearing just before that point. This replication is done using a technology called *polymerase chain reaction*, which has had a tremendous impact on experimental molecular biology.

Reaction of replication in a cell starts as a consequence of initiation a replication and then by an enzyme which continues that replication (according to the reading direction of the DNA). When considering *polymerase chain reaction* we perform the initiation and let the enzyme do the rest.

Knowing few nucleotides allows one to synthesize a sequence that is complementary to those few nucleotides. This complementary sequence can be used to create a 'primer', which finds its way to the point in the long DNA sequence containing the complement of the primer. It then hybridizes (bonds) with the longer sequence at that point. This creates the conditions that allow the replication of part of the original sequence: the replication is done by exposing the DNA to an enzyme which has the ability to duplicate sequences called *DNA polymerase*. After a while we get two subsequences. Now we can warm what we got for a short time and the bonds will separate - and again we can apply the enzyme and do the same to the separated subsequences and get more and more copies.

Bibliography

[1] F. Alizadeh, R. Karp, L. Newberg, and D. Weisser. Simian sarcoma virus onc gene v-sis, is derived from the gene encoding a platelet-derived growth factor. *Science*, 221:275–277, 1983.

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

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

[4] D. S. Hirschberg. Algorithms for the longest common subsequence problem. *J.ACM*, 24:664–675, 1977.