Algorithms for Molecular Biology **Fall Semester, 2001**

Lecture 2: November 1, 2001

Lecturer: Ron Shamir *Scribe: Doron Yaary and Ami Peled*[1]

2.1 Pairwise Alignment

2.1.1 Introduction

The lecture introduces the problem of *pairwise sequence alignment* or *inexact matching*. This is the problem of comparing two sequences while allowing certain mismatches between them.

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.

2.1.2 Problem Definition and Biological 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 experiment protocols.
- Storing, retrieving and comparing DNA sequences in Databases.
- Comparing two or more sequences for similarities.
- Searching databases for related sequences and subsequences.

[1]Based on a scribe of Ofira Grebenshuk and Alex Feldman November 6th, 2000.

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

Many of these research problems aim at learning about the functionality or structure of a 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 proteins with similar sequences available in databases.

For instance, when considering protein folding, it will usually suffice for two proteins to have 25% sequence identity for their three dimensional structures and thus - more importantly - their function to be almost identical. A classical example is the establishment of an association between cancer and uncontrolled cell growth by [2]. This discovery was enabled by comparing the sequence of a cancer associated gene against the sequences 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.

2.1.3 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 base (letter) or several bases to the sequence.
- *Deletion* - deleting a base (or more) from the sequence.
- *Substitution* - replacing a sequence base 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.

2.1.4 Nomenclature

It is important 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 computer science) is a non contiguous segment of a sequence. We will use the biological nomenclature. In particular, a "subsequence" will mean a *contiguous* sequence of letters.

2.1.5 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. 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 c t a`, the minimal number of edit operations required to transform one into the other is 2:

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

First, substitution of `c` by `g` is applied on the sequence `a c c t g a`. As a result the sequence `a g c t g a` is obtained. Then, an indel operation (deletion of `g`) is applied.

Remark: The definition of edit distance implies that all operations are done on one sequence only and the representation shown above might make the false impression that the order of the changes is known.

2.1.6 Alignment

Definition An alignment of two sequences S and T is obtained by first inserting chosen spaces, either into, at the ends of or before S and T, and then placing the two resulting sequences one above the other so that every character or space in either sequence opposite a unique character or a unique space in the other sequence. In the alignment model each two character alignment and character - space alignment is given a score (weight). Usually, insert and delete (indel) operations (alignment of a character and a space) are given the same score. Using alignment algorithms, we search for the minimal scoring (or the maximum negative scoring), representing the minimal difference or maximum similarity between the two sequences. Biological models consider the significance of each mutation and score the alignment operations accordingly. Therefore, the alignment distance can be used to estimate the "biological difference" of two DNA or protein sequences. The substitution matrix $S(i, j)$ represents the weight of each possible alignment.

Example The aligned sequences:

```
SEQ 1  GTAGTACAGCT-CAGTTGGGATCACAGGCTTCT
       ||||  ||  |||  ||||||    ||||||  |||
SEQ 2  GTAGAACGGCTTCAGTTG---TCACAGCGTTC-
```

Distance 1 - match 0, substitution 1, indel 2 $\Rightarrow$ distance = 14.

Distance 2 - match 0, d(A,T)=d(G,C)=1, d(A,G)=1.5 indel 2 $\Rightarrow$ distance = 14.5.

Similarity - match 1, substitution 0, indel -1.5 $\Rightarrow$ similarity = 16.5.

General setup - substitution matrix S(i,j), indel S(i,-) or S(-,j).

2.1.7 Models for Alignment

In this lecture we consider four 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 maximum similarity between them? Find a best alignment.

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

Problem 2.3 (Ends 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.

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. A *gap penalty function* is a function that measures the cost of a gap as a (nonlinear) function of its length.

Problem 2.4 (Gap penalty)
INPUT: Two sequences S and T (possibly of different length).
QUESTION: Find a best alignment between the two sequences using the gap penalty function.

All the problems above are studied for various biological reasons.

2.2 Global Alignment

Definition *Global 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 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 matched to a unique character or a unique space in the other sequence. The term *global* emphasizes that for each sequence, the entire sequence is involved.

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 -
```

Let us now introduce the *global alignment problem*:

Problem 2.5 (Global Alignment)
INPUT: Two sequences $S = s_1...s_n$ and $T = t_1...t_m$ (n and m are approximately the same)
QUESTION: Find an optimal alignment.

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

Lemma 2.1 *Let $V(i, j)$ be the* optimal alignment score *of $S_{1...i}$ and $T_{1...j}$ ($0 \leq i \leq n, 0 \leq j \leq m$). 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:

$$\text{for } 1 \leq i \leq n, 1 \leq j \leq m:$$

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

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 $S_{1...i}$ and $T_{1...j}$. 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 $V(i, j-1) + \sigma(-, T_j)$.

■

2.2.1 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=0 to n do
begin
    for j=0 to m do
    begin
        Calculate V(i,j) using V(i-1,j-1), V(i,j-1), V(i-1,j)
    end
end
```

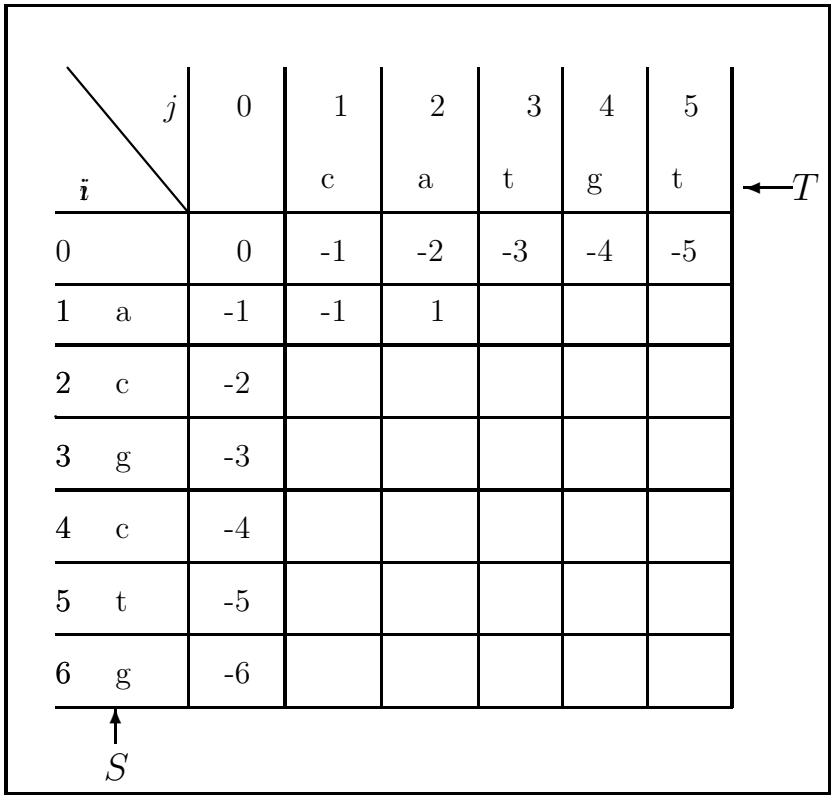

Figure 2.1: Snapshot of computing the table. The figure demonstrates some initial stages of running an algorithm for finding global alignment. In the first column and the first row one can see results of the calculations of $V(i,-)$ and $V(-,j)$. In the second row one can see the propagation of the algorithm. Here a gap and a mismatch cost -1 and a match has value +2.

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.

2.2.2 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.

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.*

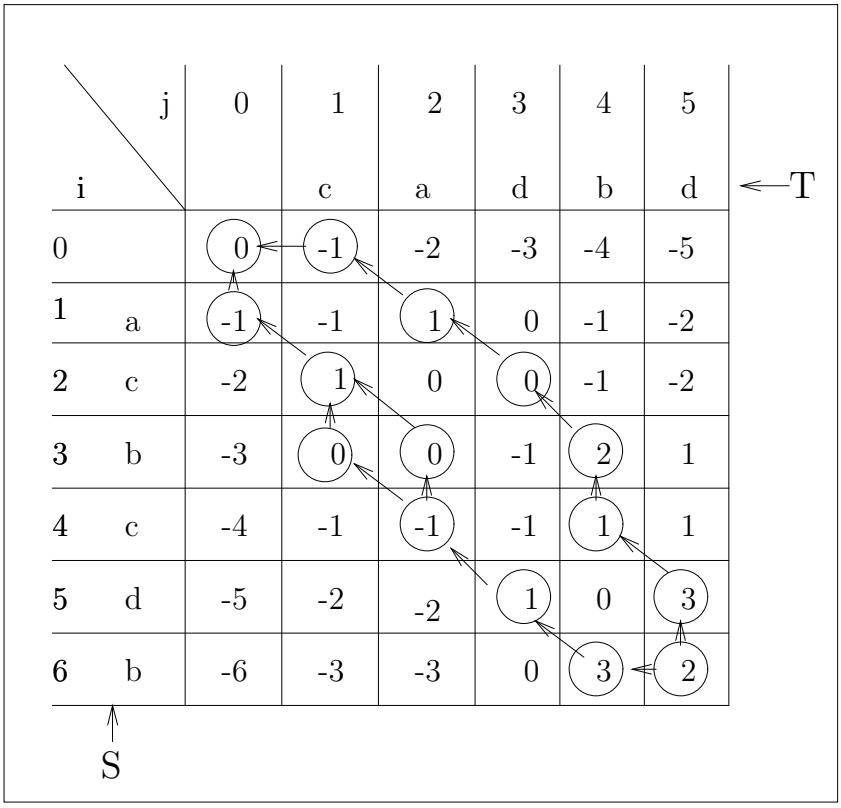

Figure 2.2: Backtracking the alignment. For each cell its outgoing arrows point to the cells from which the algorithm could arrive to the current cell. In this figure one can see 3 possible paths that represent alignments that give the highest score.

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, filling a single cell takes constant time. There are $(n+1) \times (m+1)$ cells in the table. So the time complexity is $O(nm)$.
- **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 $O(nm)$.

■

2.3 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 labelled 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 $\sigma(S_{i+1}, T_{j+1})$

Figure 2.3 illustrates the process of building the *alignment graph*.

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 algorithms.

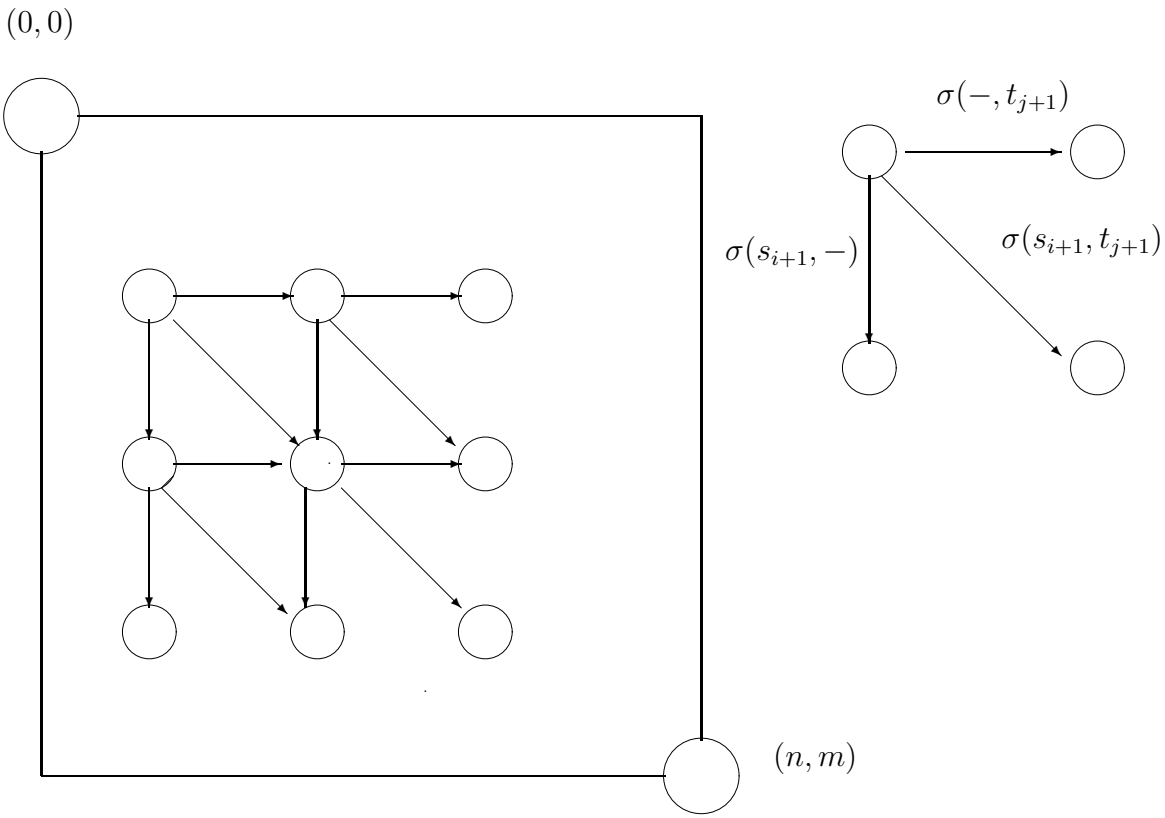

Figure 2.3: Alignment Graph. The alignment graph is a directed acyclic graph. On the right side of the figure one can see the segment of the alignment graph. It represents the three possibilities at each stage of the algorithm.

2.4 Global Alignment in Linear Space

The backtracking algorithm requires the entire matrix to be saved in memory. The space complexity consequently increases to $O(nm)$. In biological applications, the space complexity is the algorithm's bottleneck due to the huge length of DNA sequences. Hirschberg [3] developed a more practical space-reduction method for solving dynamic programming problems that reduces the required space from $O(nm)$ to $O(\min(m,n))$.

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

Lemma 2.3 $V(n,m) = \max_{0\leq k\leq m}\{V(\frac{n}{2},k) + V^r(\frac{n}{2},m-k)\}$

Proof: [1, chapter 12] For any fixed position $k^{'}$ in T, there is an alignment of S and T consisting of an alignment of $S_1 \ldots S_{\frac{n}{2}}$ and $T_1 \ldots T_{k'}$ followed by a disjoint alignment of $S_{\frac{n}{2}+1} \ldots S_n$ and $T_{k'+1} \ldots T_m$. 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 $V(\frac{n}{2},k^{'}) + V^r(\frac{n}{2},m-k^{'}) \leq V(n,m)$. Since this argument holds for any k', it follows that $\max_k[V(\frac{n}{2},k) + V^r(\frac{n}{2},m-k)] \leq V(n,m)$.

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 $S_1 \ldots S_{\frac{n}{2}}$ and $T_1 \ldots T_{k'}$ followed by an alignment of $S_{\frac{n}{2}+1} \ldots T_n$ and $T_{k'+1} \ldots T_m$. Let the value of the first alignment be denoted p and the value of the second alignment be denoted q. Then p must equal to $V(\frac{n}{2},k^{'})$, for if $p < V(\frac{n}{2},k^{'})$ we could replace the alignment of $S_1 \ldots S_{\frac{n}{2}}$ and $T_1 \ldots T_{k'}$ with an alignment of $S_1 \ldots S_{\frac{n}{2}}$ and $T_1 \ldots T_{k'}$ with value $V(\frac{n}{2},k^{'})$. That would create an alignment of S and T whose value is larger than the claimed optimal. 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 $V(n,m) = max_k[V(\frac{n}{2},k) + V^r(\frac{n}{2},m-k)]$. ■

2.4.1 The Algorithm:

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

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

This is a "divide and conquer" type algorithm. It is illustrated in figure 2.4.

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 an $n \times m$ problem. Let $T(n, m)$ be the time to find the path (solution) of an $n \times m$ problem.

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

Since $T(n, m) \leq cnm$, for some c, $T(n, m) \leq 4T^*(n, m) \leq 4cnm$. Therefore, the time complexity remains $O(nm)$. ■

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

Proof: Space Complexity
Each dynamic programming computation requires storing one additional row (middle one) which can be discarded once the middle point is found. If $n < m$ we can store the middle column instead. Therefore the space complexity is $O(\min(m, n))$. ■

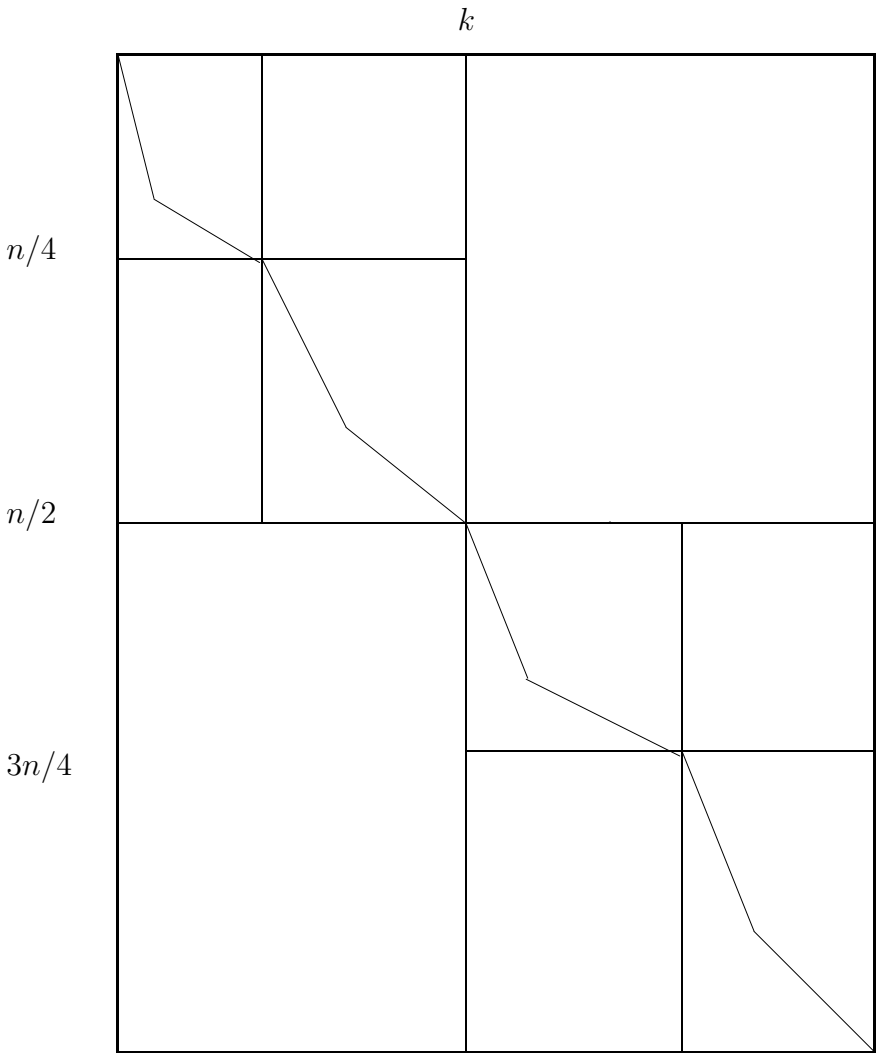

Figure 2.4: Hirschberg's [3] "Divide and Conquer" algorithm.

2.5 Local Alignment

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

Problem 2.6 (Local Alignment)
Given two sequences S and T, *local alignment* problem is defined as the problem of finding the subsequences α of S and β of T, whose similarity (optimal global alignment) is maximal (over all such pairs of subsequences).

reminder: a subsection is a contiguous segment of a sequence

2.5.1 Motivation

Ignore stretches of non-coding DNA:

In the DNA, non-coding regions (introns) are more likely to be subjected to mutations than coding regions (exons). This is since mutations in coding regions might cause a change in a protein, that will have a substantial effect on the organism. A change in a non-coding region is less likely to have an effect, and therefore has a higher probability of getting "accepted". When searching for a *local alignment* between two stretches of DNA (from 2 different specimens), finding a best match is likely to be between 2 exons.

Protein domains:

Proteins of different kind and of different species, often exhibit local similarities called *homeoboxes*. These local similarities are most probably "functional subunits" of the protein. Finding these similarities is done by solving *local alignment* between different sequences of DNA.

Example Consider the two sequences:
S = `g g t c t g a g`
T = `a a a c g a`
editing operations values:
match = 2 indel/substitution = -1
The best *local alignment* is:
α = `ctga` ($\in S$)
β = `c-ga` ($\in T$)

2.5.2 Computing Local Alignment

Definition Given two sequences, S and T, and two indices i and j, the *local suffix alignment* problem is finding a (possibly empty) suffix α of $S_{1...i}$ and a (possibly empty) suffix β of $T_{1...j}$ such that the value of their alignment is maximal over all alignments of suffixes of $S_{1...i}$ and $T_{1...j}$.

The solution to the *local alignment* problem is the same as the maximal solution to the *local suffix alignment* problem over all indices i and j of S and T.

Terminology and Restriction:

- We use $V(i,j)$ to denote the value of the optimal local suffix alignment for a given pair i,j of indices.
- We limit the weights of the editing operations by the following rule:

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

The Scheme of the Algorithm:

1. Compute *local suffix alignment* (for all i and j) of $S'_i = S_{1...i}$ and $T'_i = T_{1...j}$.
   This is done using the global alignment algorithm with a small difference: the prefixes of S' and T' whose alignments are ≤ 0 are discarded, thus allowing the subsequences to start from indices ≥ 1.
2. Search the results and find the indices i^* and j^* of S and T respectively, after which the similarity only decreases.

Recursive Definition:

- Base conditions: $\forall i,j.\ V(i,0) = 0\ ,\ V(0,j) = 0$
- Recurrence relation: $$V(i,j) = \max \begin{cases} 0 \\ V(i-1,j-1) + \sigma(S_i, T_j) \\ V(i-1,j) + \sigma(S_i, -) \\ V(i,j-1) + \sigma(-, T_j) \end{cases}$$
- Compute i^* and j^*: $V(i^*,j^*) = \max_{1\leq i\leq n, 1\leq j\leq m} V(i,j)$

Observe that the recurrence for computing local 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.

The zero in the base conditions and in the recurrence relation allows the alignment to start anywhere in the sequences, rather than forcing it to start at the beginning of each. Searching for i^* and j^* that give the best $V(i^*, j^*)$ allows the alignment to end anywhere in the sequences, rather than at the end of each. Together, we get the best alignment of two subsequences of S and T.

Example Figure 2.5 illustrates the calculation of the $n \times m$ entries table, taking σ as 2 for a match and -1 for a 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 $\alpha = S_{i'...i^*}$ and $\beta = T_{j'...j^*}$.

Lemma 2.6 *Local alignment can be solved 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.
Finding the starting positions of the two subsequences can be done in linear space using reverse dynamic programming (the details are left for the reader to fill).
As mentioned above, after α and β are found, calculating their alignment can be done in linear space using Hirschberg's method. ■

Local alignment algorithm 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 2.6, the space complexity is $O(n + m)$.

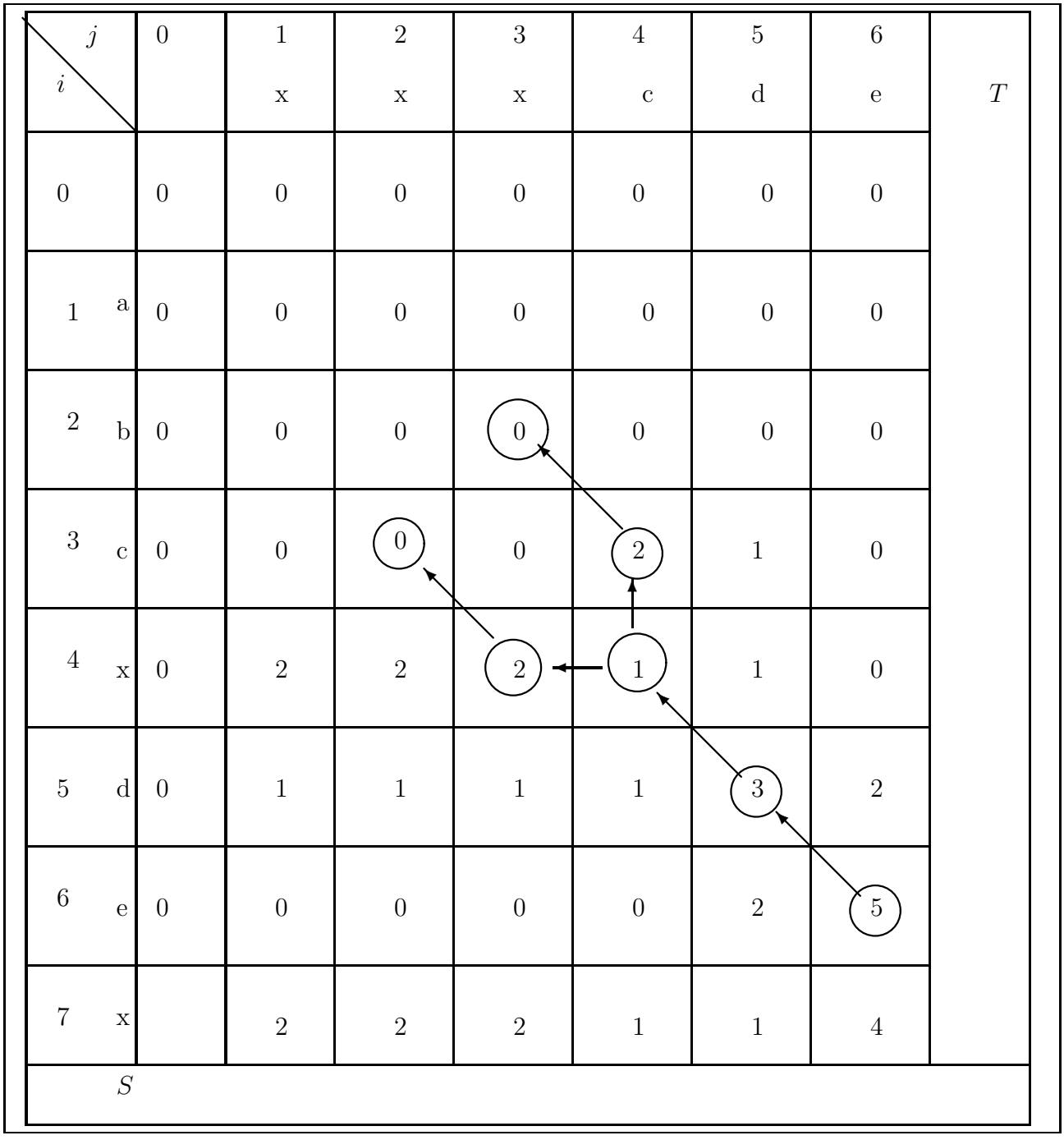

Figure 2.5: Finding local alignment

2.6 End-Space Free Alignment

In this variant of sequence alignment, any number of indel operations at the end or at the beginning of the alignment contribute zero weight. This drops the requirement that sequences start and end at the same place, and allows us to align sequences that overlap / include one another.

Example Consider the sequences:
S = c a c t g t a c
T = g a c a c t t g
Assigning value of 2 for match, and -1 for indel/substitution, the best Global alignment will have value 1 and will look like this:
S = c a c - - t - g t a c
T = g a c a c t t g - - -
while End-space free alignment will have value 9 and will look like this:
S = - - c a c - t g t a c
T = g a c a c t t g - - -
The two leading spaces at the left end of the alignment are free, as well as the three trailing spaces at the right end.

2.6.1 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 original sequence (using the overlapping sections to "paste" the subsequences together).

Two subsequences that are from *different parts* of the original sequence will have low *global alignment* score, as well as low *end-space free alignment* score. Two *overlapping subsequences* from the set are unlikely to have the same starting position (nor the same end position) along the original sequence. Therefore, they will still have a low *global alignment* score. On the other hand, when checking them for *end-space free alignment*, the two subsequences will have high score, since they posses an overlapping section. The overlap will be detected, and the subsequences will be "pasted" together using the alignment found. Similarly, the case where one subsequence contains the other can be detected by *end-space free alignment*. See figure 2.6 for illustration.

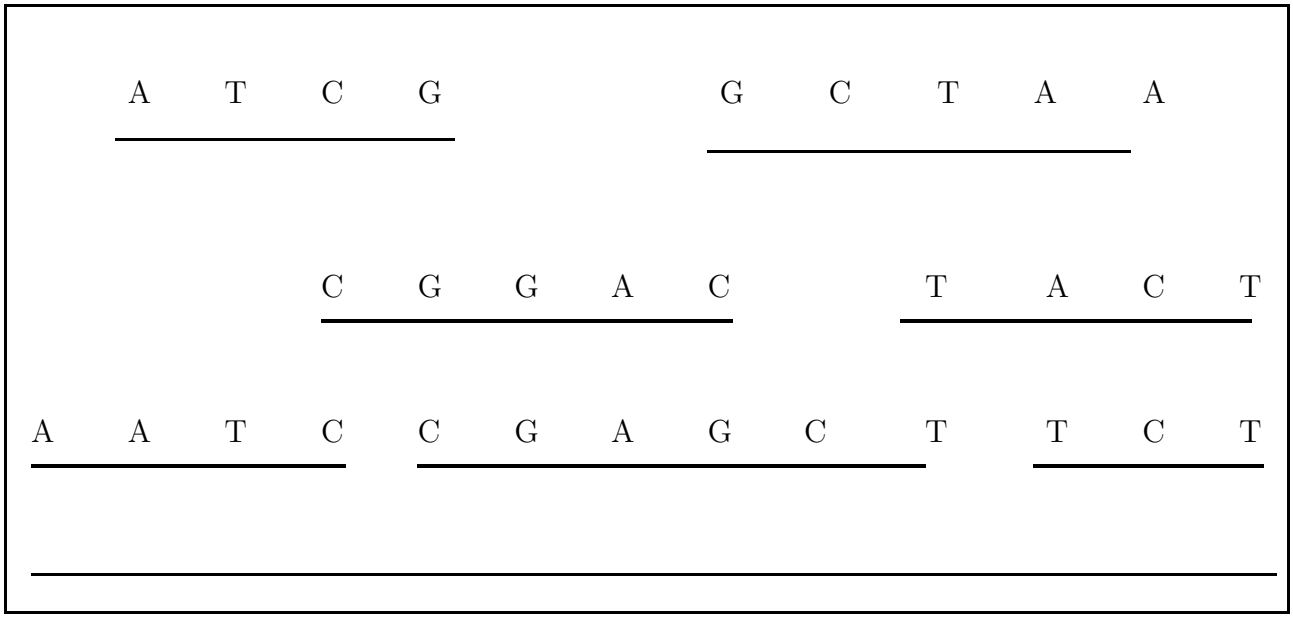

Figure 2.6: Sequence assembly.

2.6.2 The End-Space Free Alignment Algorithm

The *end-space free alignment* algorithm is again a variation of the *Global alignment* algorithm:

- We set the initial conditions to allow zero weight to leading indel operations in (at most) one of the sequences.
- After filling the table with the values of $V(i, j)$, we search for the maximal value in either of the "ending rows", thus allowing (at most) one sequence to end before the other, with zero weight for all indel operations from there on. This value is the best value
- The aligned sequence is tracked (using pointers created while filling the table) from cell $(0, 0)$ in the table until the end of one sequence (bottom row / rightmost column). from there on, all indel operations until cell (n, m) are not counted in the total value (though they are present in the table).

Recursive definition:

- Base conditions: $\forall i, j.\ V(i,0) = 0\ ,\ V(0,j) = 0$
- Recurrence relation: $V(i,j) = \max \begin{cases} V(i-1,j-1) + \sigma(S_i, T_j) \\ V(i-1,j) + \sigma(S_i, -) \\ V(i,j-1) + \sigma(-, T_j) \end{cases}$
- Search for i^* such that: $V(i^*, m) = \max_{1 \le i \le n,m} V(i,j)$
- Search for j^* such that: $V(n, j^*) = \max_{n, 1 \le j \le m} V(i,j)$
- Define alignment score: $V(S,T) = \max \begin{cases} V(n, j^*) \\ V(i^*, m) \end{cases}$

See figure 2.7 for illustration of the $V(i,j)$ table over the sequences $S = actgtta$ and $T = gttactgt$ with match = 2 and indel/substitution = -1 .

Complexity:

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

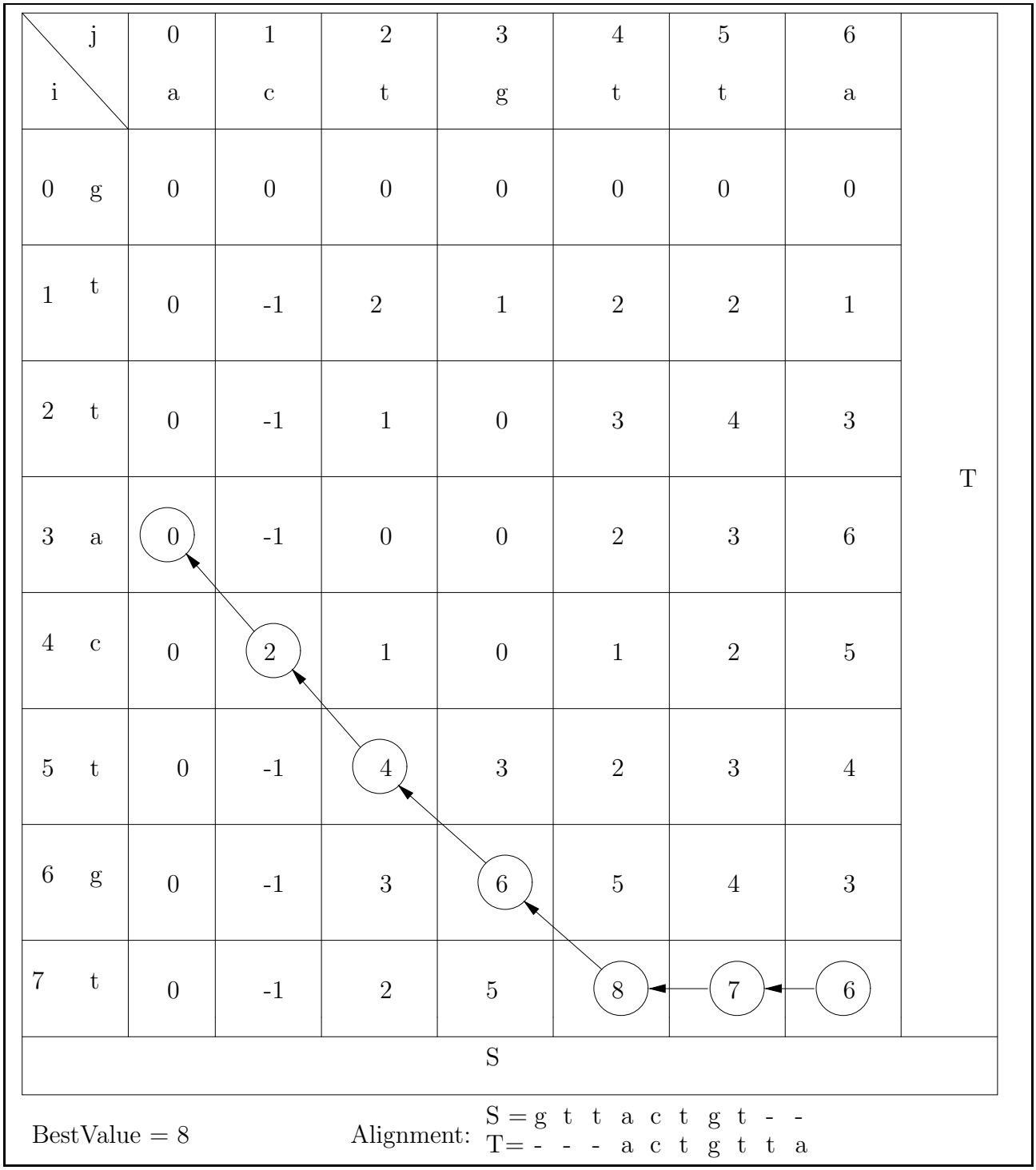

Figure 2.7: End-space free alignment table.

2.7 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*. A *gap* is a consecutive run of spaces in an alignment. 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 treat a gap as a whole, rather than give each of its spaces the same weight. There are many ways a weight/value can be given to a gap. In this section we present several gap penalty models, and an algorithm for finding a best alignment using one of them (affine gap penalty model). The number of gaps in an alignment will be denoted by #gaps.

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

Definition The *length* of a gap is the number of indel operations in it.

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.

2.7.1 Motivation

The concept of a gap in an alignment is important in many biological application, since the insertion or deletion of an entire subsequence often occurs as a single mutational event. Moreover, many of these single mutational events can create gaps of varying sizes. We will need to score a gap as a whole when we try to align two sequences of DNA so as to avoid asigning high cost to these mutations.

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. Again, the introduction of gaps will help us treat these cases as "good matches", although there are long consecutive runs of indel operations in them.

One concrete illustration of the use of gaps in the alignment model comes from the problem of cDNA matching [1, chapter 11]. To better understand the problem, we give a short biological background:

Recall that an RNA molecule is transcribed from the DNA of a gene. The RNA transcript (pre-mRNA) is a complement of the gene's 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 and exons.

After the pre-mRNA is created, a splicing process takes place: each intron-exon boundary 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 includes only regions that correspond to exons. The mRNA 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 and to find the location of the encoding genes, uses the mRNA: After "capturing" the mRNA, it is used to create a DNA sequence complementary to it. This sequence is called *cDNA* (complementary DNA). The cDNA contains the same sequence of bases as the exons on the original DNA that encode for the specific protein, with the introns missing. To determine where the gene associates with that cDNA (hence protein) resides, we now need to match the cDNA with the original DNA.

We know, that in the alignment we search for, there are many long gaps. These gaps are due to introns on the DNA that are missing on the cDNA. Using a good gap penalty model will prevent us from giving a very low score for these alignments (since they have many consecutive indel's in them) and allow us to find the true alignment.

2.7.2 Constant Gap Penalty Model

The simplest choice is the *constant* gap penalty, where each individual space is free (has no weight), and each gap is given a weight of W_g independent of its length.

- Let σ denote the weights of match and mismatch only ($\forall x\ \sigma(x,-) = \sigma(-,x) = 0$)
- Let S' and T' represent S and T after inserting spaces.
- Thus we have to find an alignment that maximizes:

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

A generalization of this model adds weight not only to an existence of a gap (W_g), but also another weight proportional to its length (W_s). This model is described below.

2.7.3 Affine Gap Penalty Model

In the *Affine gap penalty model*, a gap is given two weights. One, W_g is the weight to "open the gap", and the other, W_s is the weight to "extend the gap" with one more space.
The total penalty for a gap of length q is:

$$W_{Total} = W_g + qW_s$$

The model is called "affine" after its affine formula above.

Note that the constant gap weight model is simply the affine model with $W_s = 0$, Thus the algorithm described bellow can be used for the *constant gap penalty model* as well.

Using the same symbols as above, we have to find an alignment that maximizes:

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

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.

Notation Let us use the following notation:

- $G(i,j)$ the maximum value of any alignment of type 1
- $E(i,j)$ the maximum value of any alignment of type 2
- $F(i,j)$ the maximum value of any alignment of type 3
- $V(i,j)$ the maximum value of an alignment

Using these notations, we can recursively define the alignment table, as was done in previous algorithms. In the base conditions, we need to look at indel operations and assign the correct value: not only the weight of the spaces (qW_s), but also the weight of "opening the gap" (W_g).

We will define 3 recurrence relations, one for each of $G(i,j)$, $E(i,j)$ and $F(i,j)$. Each will be calculated from previously computed values.
Take $E(i,j)$ for example. We are looking at alignments in which S ends to the left of T:
S ————i- - - - - - -
T ——————————j
There are two possible cases for the previous alignment:

1. It looked the same, i.e., S ended to the left of T.
   In this case, we only need to add another "extension weight" to the value, forming the new weight $E(i,j-1)+W_s$

2. S and T ended at the same place (type 1 alignment).
   In this case, we need to add both the gap "opening weight" and the gap "extension weight", forming the new weight $V(i,j-1)+W_g+W_s$.

Taking the maximum of the two yields the value for $E(i,j)$.

Calculating $F(i,j)$ and $G(i,j)$ is done using similar arguments. $V(i,j)$ is Calculated by simply taking the maximum of the three. As in *Global alignment*, we search for the value $V(n,m)$, and trace the alignment back using pointers created while filling the table.

Recursive definition:

- Base conditions: $V(0,0) = 0$
  $V(i,0) = E(i,0) = W_g + iW_s$
  $V(0,j) = F(0,j) = W_g + jW_s$

- Recurrence relation: $V(i,j) = \max\{E(i,j), F(i,j), G(i,j)\}$ where
  $G(i,j) = V(i-1,j-1) + \sigma(S_i, T_j)$
  $E(i,j) = \max\{E(i,j-1) + W_s, V(i,j-1) + W_g + W_s\}$
  $F(i,j) = \max\{F(i-1,j) + W_s, V(i-1,j) + W_g + W_s\}$

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.

2.7.4 Convex Gap Penalty Model

- Each additional space in a gap contributes less to the gap weight than the previous space.

- This model is said to better describe biological behavior.

- Example: $W_g \log(q)$, where q is the length of the gap.

- The problem is solvable in $O(nm \log(m))$ time [1].

2.7.5 Arbitrary Gap Penalty Model

- Any gap weight function is acceptable (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.

2.8 Longest Common Non Contiguous Subsequence

We finish with a classical problem in computer science that is not biologically motivated, but which we can be easily solve using the machinery developed above.

Definition A *non contiguous subsequence* of S is defined as a subset of the characters of S arranged in their original "relative" order. Formally: a *non contiguous subsequence* of sequence S is specified by a list of indices $i_1 < i_2 < i_3 < \ldots < i_k$. The non contiguous subsequence specified by this list is $S_{i_1} S_{i_2} \ldots S_{i_k}$. Given two sequences S and T, a *common subsequence* is a subsequence that appears both in S and T.

Problem 2.7 (longest common non contiguous subsequence)
INPUT: Two sequences S and T.
QUESTION: What is the longest *non contiguous subsequence* common to both S and T.

The longest common non contiguous subsequence problem can be modelled and solved using the *global 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 aligned 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 a maximum length subsequence, character matches are valued as '1', while a space match is valued '0'.

Bibliography

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

[2] 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 encoding a platelet-derived growth factor. *Science*, 221:275–277, 1983.

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