6.4 Edit Distance and Alignments

So far, we have been vague about what we mean by "sequence similarity" or "distance" between DNA sequences. Hamming distance (introduced in chapter 4), while important in computer science, is not typically used to compare DNA or protein sequences. The Hamming distance calculation rigidly assumes that the ith symbol of one sequence is already *aligned* against the ith symbol of the other. However, it is often the case that the ith symbol in one sequence corresponds to a symbol at a different—and unknown—position in the other. For example, mutation in DNA is an evolutionary process: DNA replication errors cause substitutions, insertions, and deletions of nucleotides, leading to "edited" DNA texts. Since DNA sequences are subject to insertions and deletions, biologists rarely have the luxury of knowing in advance whether the ith symbol in one DNA sequence corresponds to the ith symbol in the other.

As figure 6.10 (a) shows, while strings ATATATAT and TATATATA are very different from the perspective of Hamming distance, they become very similar if one simply moves the second string over one place to align the $(i+1)$-st letter in ATATATAT against the ith letter in TATATATA for $1 \leq i \leq 7$. Strings ATATATAT and TATAAT present another example with more subtle similarities. Figure 6.10 (b) reveals these similarities by aligning position 2 in ATATATAT against position 1 in TATAAT. Other pairs of aligned positions are 3 against 2, 4 against 3, 5 against 4, 7 against 5, and 8 against 6 (positions 1 and 6 in ATATATAT remain unaligned).

In 1966, Vladimir Levenshtein introduced the notion of the *edit distance* between two strings as the minimum number of editing operations needed to transform one string into another, where the edit operations are insertion of a symbol, deletion of a symbol, and substitution of one symbol for another. For example, TGCATAT can be transformed into ATCCGAT with five editing operations, shown in figure 6.11. This implies that the edit distance between TGCATAT and ATCCGAT is at most 5. Actually, the edit distance between them is 4 because you can transform one to the other with one move fewer, as in figure 6.12.

Unlike Hamming distance, edit distance allows one to compare strings of different lengths. Oddly, Levenshtein introduced the definition of edit distance but never described an algorithm for actually finding the edit distance between two strings. This algorithm has been discovered and rediscovered many times in applications ranging from automated speech recognition to, obviously, molecular biology. Although the details of the algorithms are

```
A T A T A T A T -
  : : : : : : :
- T A T A T A T A
```

(a) Alignment of ATATATAT against TATATATA.

```
A T A T A T A T
  : : : :   : :
- T A T A - A T
```

(b) Alignment of ATATATAT against TATAAT.

Figure 6.10 Alignment of ATATATAT against TATATATA and of ATATATAT against TATAAT.

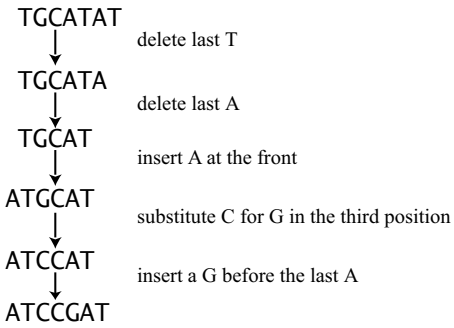

Figure 6.11 Five edit operations can take TGCATAT into ATCCGAT.

slightly different across the various applications, they are all based on dynamic programming.

The *alignment* of the strings **v** (of n characters) and **w** (of m characters, with m not necessarily the same as n) is a two-row matrix such that the first row contains the characters of **v** in order while the second row contains the characters of **w** in order, where spaces may be interspersed throughout the strings in different places. As a result, the characters in each string appear in order, though not necessarily adjacently. We also assume that no column

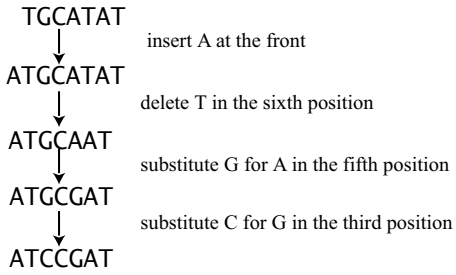

Figure 6.12 Four edit operations can also take TGCATAT into ATCCGAT.

of the alignment matrix contains spaces in both rows, so that the alignment may have at most $n + m$ columns.

A	T	-	G	T	T	A	T	-
A	T	C	G	T	-	A	-	C

Columns that contain the same letter in both rows are called *matches*, while columns containing different letters are called *mismatches*. The columns of the alignment containing one space are called *indels*, with the columns containing a space in the top row called *insertions* and the columns with a space in the bottom row *deletions*. The alignment shown in figure 6.13 (top) has five matches, zero mismatches, and four indels. The number of matches plus the number of mismatches plus the number of indels is equal to the length of the alignment matrix and must be smaller than $n + m$.

Each of the two rows in the alignment matrix is represented as a string interspersed by space symbols "$-$"; for example AT$-$GTTAT$-$ is a representation of the row corresponding to $\mathbf{v}$ = ATGTTAT, while ATCGT$-$A$-$C is a representation of the row corresponding to $\mathbf{w}$ = ATCGTAC. Another way to represent the row AT$-$GTTAT$-$ is 1 2 2 3 4 5 6 7 7, which shows the number of symbols of $\mathbf{v}$ present up to a given position. Similarly, ATCGT$-$A$-$C is represented as 1 2 3 4 5 5 6 6 7. When both rows of an alignment are represented in this way (fig. 6.13, top), the resulting matrix is

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 2 \\ 2 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \end{pmatrix} \begin{pmatrix} 3 \\ 4 \end{pmatrix} \begin{pmatrix} 4 \\ 5 \end{pmatrix} \begin{pmatrix} 5 \\ 5 \end{pmatrix} \begin{pmatrix} 6 \\ 6 \end{pmatrix} \begin{pmatrix} 7 \\ 6 \end{pmatrix} \begin{pmatrix} 7 \\ 7 \end{pmatrix}$$

Each column in this matrix is a coordinate in a two-dimensional $n \times m$ grid;

the entire alignment is simply a path

$$(0,0) \to (1,1) \to (2,2) \to (2,3) \to (3,4) \to (4,5) \to (5,5) \to (6,6) \to (7,6) \to (7,7)$$

from $(0,0)$ to (n,m) in that grid (again, see figure 6.13). This grid is similar to the Manhattan grid that we introduced earlier, where each entry in the grid looks like a city block. The main difference is that here we can move along the diagonal. We can construct a graph, this time called the *edit graph*, by introducing a vertex for every intersection of streets in the grid, shown in figure 6.13. The edit graph will aid us in calculating the edit distance.

Every alignment corresponds to a path in the edit graph, and every path in the edit graph corresponds to an alignment where every edge in the path corresponds to one column in the alignment (fig. 6.13). Diagonal edges in the path that end at vertex (i,j) in the graph correspond to the column $\begin{pmatrix} v_i \\ w_j \end{pmatrix}$, horizontal edges correspond to $\begin{pmatrix} - \\ w_j \end{pmatrix}$, and vertical edges correspond to $\begin{pmatrix} v_i \\ - \end{pmatrix}$. The alignment above can be drawn as follows.

$$\begin{array}{ccccccccccc} & \mathsf{A} & \mathsf{T} & - & \mathsf{G} & \mathsf{T} & \mathsf{T} & \mathsf{A} & \mathsf{T} & - \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 1 \\ 1 \end{pmatrix} & \begin{pmatrix} 2 \\ 2 \end{pmatrix} & \begin{pmatrix} 2 \\ 3 \end{pmatrix} & \begin{pmatrix} 3 \\ 4 \end{pmatrix} & \begin{pmatrix} 4 \\ 5 \end{pmatrix} & \begin{pmatrix} 5 \\ 5 \end{pmatrix} & \begin{pmatrix} 6 \\ 6 \end{pmatrix} & \begin{pmatrix} 7 \\ 6 \end{pmatrix} & \begin{pmatrix} 7 \\ 7 \end{pmatrix} \\ & \mathsf{A} & \mathsf{T} & \mathsf{G} & \mathsf{C} & \mathsf{T} & - & \mathsf{A} & - & \mathsf{C} \end{array}$$

Analyzing the merit of an alignment is equivalent to analyzing the merit of the corresponding path in the edit graph. Given any two strings, there are a large number of different alignment matrices and corresponding paths in the edit graph. Some of these have a surplus of mismatches and indels and a small number of matches, while others have many matches and few indels and mismatches. To determine the relative merits of one alignment over another, we rely on the notion of a scoring function, which takes as input an alignment matrix (or, equivalently, a path in the edit graph) and produces a score that determines the "goodness" of the alignment. There are a variety of scoring functions that we could use, but we want one that gives higher scores to alignments with more matches. The simplest functions score a column as a positive number if both letters are the same, and as a negative number if the two letters are different. The score for the whole alignment is the sum of the individual column scores. This scoring scheme amounts to

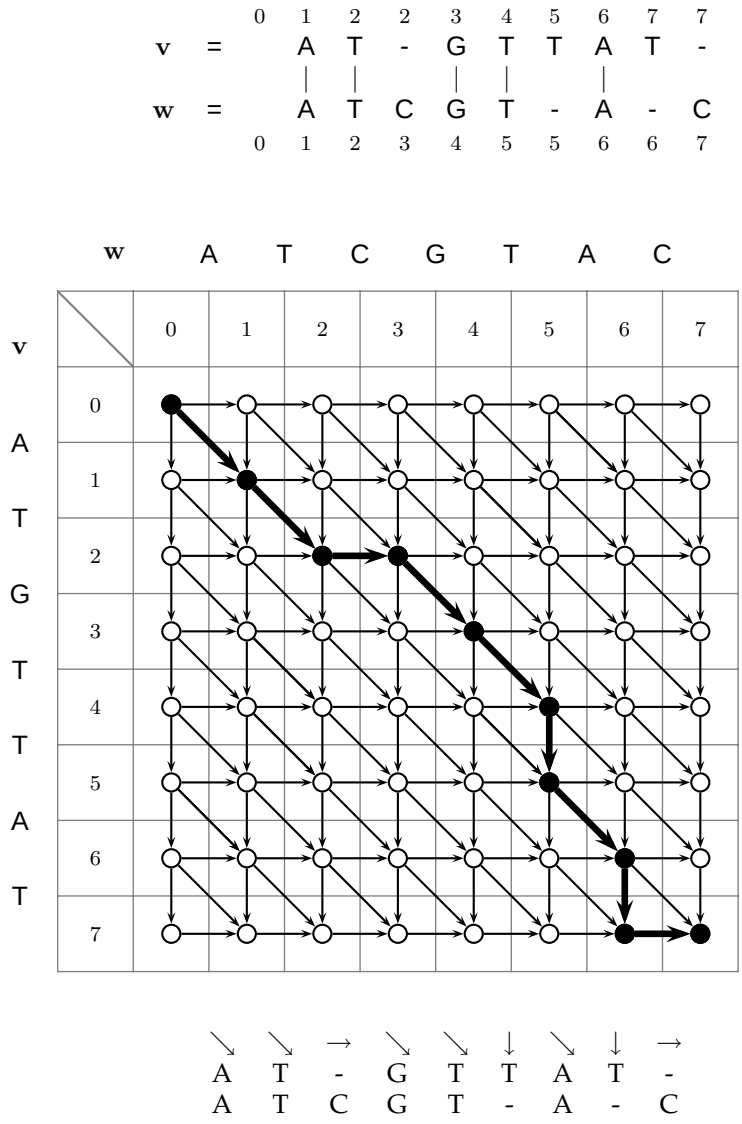

Figure 6.13 An alignment grid for $\mathbf{v} = \mathsf{ATGTTAT}$ and $\mathbf{w} = \mathsf{ATCGTAC}$. Every alignment corresponds to a path in the alignment grid from $(0, 0)$ to (n, m), and every path from $(0, 0)$ to (n, m) in the alignment grid corresponds to an alignment.

assigning weights to the edges in the edit graph.

By choosing different scoring functions, we can solve different string comparison problems. If we choose the very simple scoring function of "$+1$ for a match, 0 otherwise," then the problem becomes that of finding the longest common subsequence between two strings, which is discussed below. Before describing how to calculate Levenshtein's edit distance, we develop the Longest Common Subsequence problem as a warm-up.

6.5 Longest Common Subsequences

The simplest form of a sequence similarity analysis is the Longest Common Subsequence (LCS) problem, where we eliminate the operation of substitution and allow only insertions and deletions. A *subsequence* of a string $\mathbf{v}$ is simply an (ordered) sequence of characters (not necessarily consecutive) from $\mathbf{v}$. For example, if $\mathbf{v} = \mathsf{ATTGCTA}$, then AGCA and ATTA are subsequences of $\mathbf{v}$ whereas TGTT and TCG are not.[9] A *common* subsequence of two strings is a subsequence of both of them. Formally, we define the *common subsequence* of strings $\mathbf{v} = v_1 \dots v_n$ and $\mathbf{w} = w_1 \dots w_m$ as a sequence of positions in $\mathbf{v}$,

$$1 \leq i_1 < i_2 < \cdots < i_k \leq n$$

and a sequence of positions in $\mathbf{w}$,

$$1 \leq j_1 < j_2 < \cdots < j_k \leq m$$

such that the symbols at the corresponding positions in $\mathbf{v}$ and $\mathbf{w}$ coincide:

$$v_{i_t} = w_{j_t} \text{ for } 1 \leq t \leq k.$$

For example, TCTA is a common to both $\mathsf{ATCTGAT}$ and TGCATA.

Although there are typically many common subsequences between two strings $\mathbf{v}$ and $\mathbf{w}$, some of which are longer than others, it is not immediately obvious how to find the longest one. If we let $s(\mathbf{v}, \mathbf{w})$ be the length of the longest common subsequence of $\mathbf{v}$ and $\mathbf{w}$, then the edit distance between $\mathbf{v}$ and $\mathbf{w}$—under the assumption that only insertions and deletions are allowed—is $d(\mathbf{v}, \mathbf{w}) = n + m - 2s(\mathbf{v}, \mathbf{w})$, and corresponds to the mini-

9. The difference between a sub*sequence* and a sub*string* is that a substring consists only of consecutive characters from $\mathbf{v}$, while a subsequence may pick and choose characters from $\mathbf{v}$ as long as their ordering is preserved.

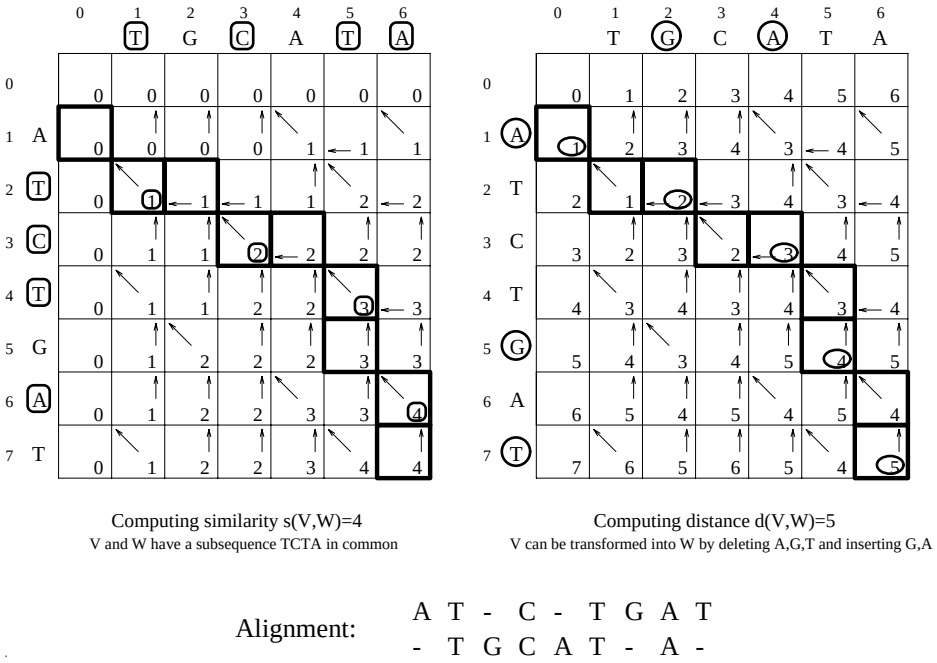

Figure 6.14 Dynamic programming algorithm for computing the longest common subsequence.

mum number of insertions and deletions needed to transform **v** into **w**. Figure 6.14 (bottom) presents an LCS of length 4 for the strings $\mathbf{v} = \mathsf{ATCTGAT}$ and $\mathbf{w} = \mathsf{TGCATA}$ and a shortest sequence of two insertions and three deletions transforming **v** into **w** (shown by "-" in the figure). The LCS problem follows.

Longest Common Subsequence Problem:
Find the longest subsequence common to two strings.

Input: Two strings, **v** and **w**.

Output: The longest common subsequence of **v** and **w**.

What do the LCS problem and the Manhattan Tourist problem have in common? Every common subsequence corresponds to an alignment with no

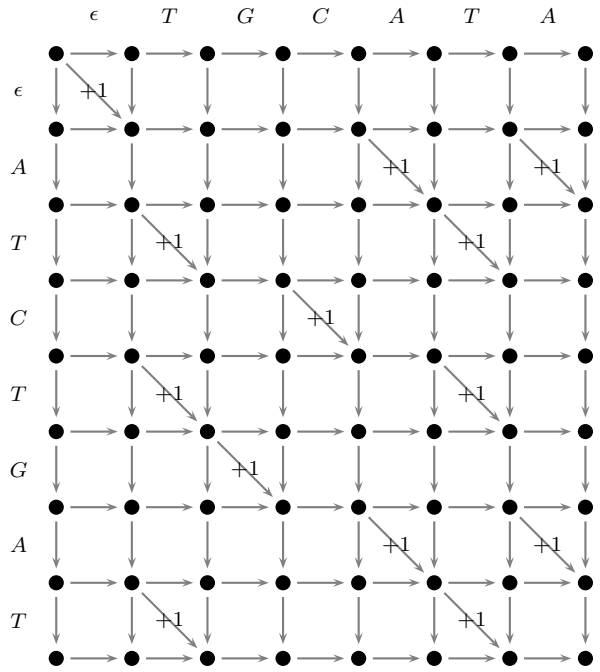

Figure 6.15 An LCS edit graph.

mismatches. This can be obtained simply by removing all diagonal edges from the edit graph whose characters do not match, thus transforming it into a graph like that shown in figure 6.15. We further illustrate the relationship between the Manhattan Tourist problem and the LCS Problem by showing that these two problems lead to very similar recurrences.

Define $s_{i,j}$ to be the length of an LCS between $v_1 \dots v_i$, the i-prefix of $\mathbf{v}$ and $w_1 \dots w_j$, the j-prefix of $\mathbf{w}$. Clearly, $s_{i,0} = s_{0,j} = 0$ for all $1 \leq i \leq n$ and

$1 \leq j \leq m$. One can see that $s_{i,j}$ satisfies the following recurrence:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} \\ s_{i,j-1} \\ s_{i-1,j-1} + 1, & \text{if } v_i = w_j \end{cases}$$

The first term corresponds to the case when v_i is not present in the LCS of the i-prefix of $\mathbf{v}$ and j-prefix of $\mathbf{w}$ (this is a deletion of v_i); the second term corresponds to the case when w_j is not present in this LCS (this is an insertion of w_j); and the third term corresponds to the case when both v_i and w_j are present in the LCS (v_i *matches* w_j). Note that one can "rewrite" these recurrences by adding some zeros here and there as

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + 0 \\ s_{i,j-1} + 0 \\ s_{i-1,j-1} + 1, & \text{if } v_i = w_j \end{cases}$$

This recurrence for the LCS computation is like the recurrence given at the end of the section 6.3, if we were to build a particularly gnarly version of Manhattan and gave horizontal and vertical edges weights of 0, and set the weights of diagonal (matching) edges equal to $+1$ as in figure 6.15.

In the following, we use $\mathbf{s}$ to represent our dynamic programming table, the data structure that we use to fill in the dynamic programming recurrence. The length of an LCS between $\mathbf{v}$ and $\mathbf{w}$ can be read from the element (n, m) of the dynamic programming table, but to reconstruct the LCS from the dynamic programming table, one must keep some additional information about which of the three quantities, $s_{i-1,j}$, $s_{i,j-1}$, or $s_{i-1,j-1} + 1$, corresponds to the maximum in the recurrence for $s_{i,j}$. The following algorithm achieves this goal by introducing *backtracking pointers* that take one of the three values $\leftarrow$, $\uparrow$, or $\nwarrow$. These specify which of the above three cases holds, and are stored in a two-dimensional array $\mathbf{b}$ (see figure 6.14).

LCS($\mathbf{v}, \mathbf{w}$)
1 **for** $i \leftarrow 0$ **to** n
2 $\quad s_{i,0} \leftarrow 0$
3 **for** $j \leftarrow 1$ **to** m
4 $\quad s_{0,j} \leftarrow 0$
5 **for** $i \leftarrow 1$ **to** n
6 $\quad$ **for** $j \leftarrow 1$ **to** m
7 $\quad\quad s_{i,j} \leftarrow \max \begin{cases} s_{i-1,j} \\ s_{i,j-1} \\ s_{i-1,j-1} + 1, & \text{if } v_i = w_j \end{cases}$
8 $\quad\quad b_{i,j} \leftarrow \begin{cases} \text{“}\uparrow\text{”} & \text{if } s_{i,j} = s_{i-1,j} \\ \text{“}\leftarrow\text{”} & \text{if } s_{i,j} = s_{i,j-1} \\ \text{“}\nwarrow\text{”}, & \text{if } s_{i,j} = s_{i-1,j-1} + 1 \end{cases}$
9 **return** $(s_{n,m}, \mathbf{b})$

The following recursive program prints out the longest common subsequence using the information stored in $\mathbf{b}$. The initial invocation that prints the solution to the problem is PRINTLCS($\mathbf{b}, \mathbf{v}, n, m$).

PRINTLCS($\mathbf{b}, \mathbf{v}, i, j$)
1 **if** $i = 0$ **or** $j = 0$
2 $\quad$ **return**
3 **if** $b_{i,j} =$ “$\nwarrow$”
4 $\quad$ PRINTLCS($\mathbf{b}, \mathbf{v}, i-1, j-1$)
5 $\quad$ **print** v_i
6 **else**
7 $\quad$ **if** $b_{i,j} =$ “$\uparrow$”
8 $\quad\quad$ PRINTLCS($\mathbf{b}, \mathbf{v}, i-1, j$)
9 $\quad$ **else**
10 $\quad\quad$ PRINTLCS($\mathbf{b}, \mathbf{v}, i, j-1$)

The dynamic programming table in figure 6.14 (left) presents the computation of the similarity score $s(\mathbf{v}, \mathbf{w})$ between $\mathbf{v}$ and $\mathbf{w}$, while the table on the right presents the computation of the edit distance between $\mathbf{v}$ and $\mathbf{w}$ under the assumption that insertions and deletions are the only allowed operations. The edit distance $d(\mathbf{v}, \mathbf{w})$ is computed according to the initial conditions $d_{i,0} = i$, $d_{0,j} = j$ for all $1 \leq i \leq n$ and $1 \leq j \leq m$ and the following recurrence:

$$d_{i,j} = \min \begin{cases} d_{i-1,j} + 1 \\ d_{i,j-1} + 1 \\ d_{i-1,j-1}, & \text{if } v_i = w_j \end{cases}$$

6.6 Global Sequence Alignment

The LCS problem corresponds to a rather restrictive scoring that awards 1 for matches and does not penalize indels. To generalize scoring, we extend the k-letter alphabet $\mathcal{A}$ to include the gap character "$-$", and consider an arbitrary $(k+1) \times (k+1)$ *scoring matrix* δ, where k is typically 4 or 20 depending on the type of sequences (DNA or protein) one is analyzing. The score of the column $\binom{x}{y}$ in the alignment is $\delta(x, y)$ and the alignment score is defined as the sum of the scores of the columns. In this way we can take into account scoring of mismatches and indels in the alignment. Rather than choosing a particular scoring matrix and then resolving a restated alignment problem, we will pose a general Global Alignment problem that takes the scoring matrix as input.

Global Alignment Problem:
Find the best alignment between two strings under a given scoring matrix.

Input: Strings **v**, **w** and a scoring matrix δ.

Output: An alignment of **v** and **w** whose score (as defined by the matrix δ) is maximal among all possible alignments of **v** and **w**.

The corresponding recurrence for the score $s_{i,j}$ of an optimal alignment between the i-prefix of **v** and j-prefix of **w** is as follows:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \\ s_{i-1,j-1} + \delta(v_i, w_j) \end{cases}$$

When mismatches are penalized by some constant $-\mu$, indels are penalized by some other constant $-\sigma$, and matches are rewarded with $+1$, the resulting score is

$$\#matches - \mu \cdot \#mismatches - \sigma \cdot \#indels$$