6 Alignments

An *alignment* of two sequences A and B aims to highlight how much in common the two sequences have. This concept arises naturally in settings where a sequence A changes over time into B, through some elementary *edit operations*, such as insertion or deletion of a character (operations called *indels*), or substitution/mutation of one character with/into another. An alignment of the characters which have survived over time could be defined informally as a list of pairs of indices (i, j), such that $A[i]$ is considered to *match* $B[j]$.

In a computational biology context, the sequences A and B could be short extracts from the genomes of two living species, fragments considered to have descended from an unknown ancestor sequence C. A biologically meaningful alignment of A and B should take into account the path in the evolutionary tree from A to C and from C to B. However, since in practice C and the evolutionary tree are unknown, then one could parsimoniously prefer an alignment with the minimum number of edit operations.

Taking into account some properties of biological sequences, one can assign different costs to the different edit operations. For example, in a coding region, the substitution of a base is usually less critical than an indel, which alters the synchronization between codons and the amino acid chain under translation. The former is likely to survive in an evolutionary selection, while the latter is not. Therefore, one could assign a high positive cost for indels, some low positive cost for substitutions, and cost 0 for an *identity*, that is, for two matched characters that are also equal. The biologically meaningful alignment could then be approximated by finding the alignment with minimum total cost. This cost of an optimal alignment is also called the *edit distance*.

In biological sequence analysis it has been customary to resort instead to a maximization framework, where in place of cost one assigns a *score* to each operation. The score of an alignment thus becomes a measure of the similarity between two sequences. Continuing our coding region example, identities are then given the highest score, the substitution of a base that does not affect the 3-base codon coding can be given a score almost as high as the one for identity. Indels are typically given negative scores. The correct alignment could then be approximated by finding the alignment with maximum total score.

Distances and similarities can typically be transformed from one to another through reductions, but each has some unique advantages. For example, with alignment similarity it is easier to give a probabilistic interpretation of an alignment, whereas a distance computation can more easily be related to tailored optimizations.

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

We start with a more generic distance interpretation of alignments that works for any two sequences, and then shift our focus to biological sequences.

6.1 Edit distance

Let us now formally define an alignment of two sequences. First, we say that a sequence $C = c_1 c_2 \cdots c_r$ is a *subsequence* of $A = a_1 a_2 \cdots a_m$, if C can be obtained by deleting zero or more characters from A.

DEFINITION 6.1 *An alignment of $A = a_1 \cdots a_m \in \Sigma^*$ and $B = b_1 \cdots b_n \in \Sigma^*$ is a pair of sequences $U = u_1 u_2 \cdots u_h$ and $L = l_1 l_2 \cdots l_h$ of length $h \in [\max\{m, n\}..n + m]$ such that A is a subsequence of U, B is a subsequence of L, U contains $h - m$ characters $-$, and L contains $h - n$ characters $-$, where $-$ is a special character not appearing in A or B.*

The following example illustrates this definition.

Example 6.1 Let us consider two expressions from Finnish and Estonian, having the same meaning (*welcome*), the Finnish `tervetuloa` and the Estonian `tere tulemast`. We set $A =$ `tervetuloa` and $B =$ `teretulemast` (by removing the empty space). Visualize a plausible alignment of A and B.

Solution
Two possible sequences U (for *upper*) and L (for *lower*) from Definition 6.1 for A and B are shown below:

```
tervetulo-a--
ter-etulemast
```

The *cost* of the alignment (U, L) of A and B is defined as

$$C(U, L) = \sum_{i=1}^{h} c(u_i, l_i),$$

where $c(a, b)$ is a non-negative cost of aligning any two characters a to b (the function $c(\cdot, \cdot)$ is also called a *cost model*). In the *unit* cost model, we set $c(a, -) = c(-, b) = 1$, $c(a, b) = 1$ for $a \neq b$, and $c(a, b) = 0$ for $a = b$. An alignment can be interpreted as a sequence of edits to convert A into B as shown in Example 6.2. We denote these edits as $u_i \to l_i$.

Example 6.2 Interpret the alignment from Example 6.1 as a sequence of edit operations applied *only* to A so that it becomes equal to B.

Solution
Sequence `ter` stays as it is, `v` is deleted, `etul` stays as it is, `o` is substituted by `e`, `m` is inserted, `a` stays as it is, `s` is inserted, and `t` is inserted.

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

Problem 6.3 Edit distance

Given two sequences $A = a_1 \cdots a_m \in \Sigma^*$ and $B = b_1 \cdots b_n \in \Sigma^*$, and a cost model $c(\cdot, \cdot)$, find an alignment of A and B having minimum *edit distance*

$$D(A, B) = \min_{(U,L) \in \mathcal{A}(A,B)} C(U, L),$$

where $\mathcal{A}(A, B)$ denotes the set of all valid alignments of A and B.

Fixing $c(\cdot, \cdot)$ as the unit cost model, we obtain the *unit cost* edit distance. This distance is also called the *Levenshtein distance*, and is denoted $D_{\mathsf{L}}(A, B)$.

Setting the cost model to $c(a, -) = c(-, b) = \infty$, $c(a, b) = 1$ for $a \neq b$, and $c(a, b) = 0$ for $a = b$, the resulting distance is called the *Hamming distance*, and is denoted $D_{\mathsf{H}}(A, B)$. This distance has a finite value only for sequences of the same length, in which case it equals the number of positions i in the two sequences which *mismatch*, that is, for which $a_i \neq b_i$.

6.1.1 Edit distance computation

Let us denote $d_{i,j} = D(a_1 \cdots a_i, b_1 \cdots b_j)$, for all $0 \leq i \leq m$ and $0 \leq j \leq n$. The following theorem leads to an efficient algorithm for computing the edit distance between the sequences A and B.

THEOREM 6.2 *For all $0 \leq i \leq m$, $0 \leq j \leq n$, with $i + j > 0$, the edit distances $d_{i,j}$, and in particular $d_{m,n} = D(A, B)$, can be computed using the recurrence*

$$d_{i,j} = \min\{d_{i-1,j-1} + c(a_i, b_j), d_{i-1,j} + c(a_i, -), d_{i,j-1} + c(-, b_j)\}, \tag{6.1}$$

by initializing $d_{0,0} = 0$, and interpreting $d_{i,j} = \infty$ if $i < 0$ or $j < 0$.

Proof The initialization is correct, since the cost of an empty alignment is zero. Fix i and j, and assume by induction that $d_{i',j'}$ is correctly computed for all $i' + j' < i + j$. Consider all the possible ways that the alignments of $a_1 a_2 \cdots a_i$ and $b_1 b_2 \cdots b_j$ can end. There are three cases.

(i) $a_i \to b_j$ (the alignment ends with substitution or identity). Such an alignment has the same cost as the best alignment of $a_1 a_2 \cdots a_{i-1}$ and $b_1 b_2 \cdots b_{j-1}$ plus the cost of a substitution or identity between a_i and b_j, that is, $d_{i-1,j-1} + c(a_i, b_j)$.

(ii) $a_i \to -$ (the alignment ends with the deletion of a_i). Such an alignment has the same cost as the best alignment of $a_1 a_2 \cdots a_{i-1}$ and $b_1 b_2 \cdots b_j$ plus the cost of the deletion of a_i, that is, $d_{i-1,j} + c(a_i, -)$.

(iii) $- \to b_j$ (the alignment ends with the insertion of b_j). Such an alignment has the same cost as the best alignment of $a_1 a_2 \cdots a_i$ and $b_1 b_2 \cdots b_{j-1}$ plus the cost of the insertion of b_j, that is, $d_{i,j-1} + c(-, b_j)$.

Since $c(-, -) = 0$, the case where a possible alignment ends with $- \to -$ does not need to be taken into account; such an alignment has equal cost to one ending with one

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

of the cases above. Thus, the cost of the optimal alignment equals the minimum of the costs in the three cases (i), (ii), and (iii), and hence Equation (6.1) holds. □

As is the case with any such recurrence relation, an algorithm to compute $d_{m,n} = D(A, B)$ follows directly by noticing that one can reduce this problem to a shortest-path problem on a DAG (recall Section 4.1). This DAG is constructed by adding, for every pair (i,j), with $0 \leq i \leq m$ and $0 \leq j \leq n$,

- a vertex $v_{i,j}$,
- the arc $(v_{i-1,j-1}, v_{i,j})$ with cost $c(a_i, b_j)$ (if $i,j > 0$),
- the arc $(v_{i-1,j}, v_{i,j})$ with cost $c(a_i, -)$ (if $i > 0$), and
- the arc $(v_{i,j-1}, v_{i,j})$ with cost $c(-, b_j)$ (if $j > 0$).

The cost of the shortest (that is, of minimum cost) path from $v_{0,0}$ to $v_{m,n}$ is then equal to $d_{m,n} = D(A, B)$. Using the algorithm given in Section 4.1.2, this path can be found in $O(mn)$ time, since the resulting DAG has $O(mn)$ vertices and $O(mn)$ arcs.

However, since the resulting DAG is a *grid*, a simpler tailored algorithm follows easily. Namely, values $d_{i,j}$, for all $0 \leq i \leq m$, $0 \leq j \leq n$, can be computed by *dynamic programming* (sometimes also called *tabulation*, since the algorithm proceeds by filling the cells of a table), as shown in Algorithm 6.1

Algorithm 6.1: Edit distance computation

Input: Sequences $A = a_1 a_2 \cdots a_m$ and $B = b_1 b_2 \cdots b_n$.
Output: Edit distance $D(A, B)$.

```
1 d_{0,0} = 0;
2 for i ← 1 to m do
3 |  d_{i,0} ← d_{i-1,0} + c(a_i, −);
4 for j ← 1 to n do
5 |  d_{0,j} ← d_{0,j-1} + c(−, b_j);
6 for j ← 1 to n do
7 |  for i ← 1 to m do
8 |  |  d_{i,j} ← min{d_{i-1,j-1} + c(a_i, b_j), d_{i-1,j} + c(a_i, −), d_{i,j-1} + c(−, b_j)};
9 return d_{m,n};
```

The time requirement of Algorithm 6.1 is $\Theta(mn)$. The computation is shown in Example 6.4; the illustration used there is called the *dynamic programming matrix*. Referring to this matrix, we often talk about its *column j* defined as

$$D_{*j} = \{d_{i,j} \mid 0 \leq i \leq m\}, \tag{6.2}$$

its *row i*, defined as

$$D_{i*} = \{d_{i,j} \mid 0 \leq j \leq n\}, \tag{6.3}$$

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

and its *diagonal* k, defined as

$$D^k = \{d_{i,j} \mid 0 \leq i \leq m, 0 \leq j \leq n, j - i = k\}. \tag{6.4}$$

Notice that, even though we defined D_{*j}, D_{i*}, and D^k as sets of numbers, we interpret them as sets of *cells* of the dynamic programming matrix.

The *evaluation order* for computing values $d_{i,j}$ can be any order such that the values on which each $d_{i,j}$ depends are computed before. Algorithm 6.1 uses the *column-by-column order*. By exchanging the two internal for-loops, one obtains the *row-by-row order*.

From Example 6.4 one observes that values $d_{i,j}$ at column j depend only on the values at the same column and at column $j-1$. Hence, the space needed for the computation can be easily improved to $O(m)$ (see Exercise 6.2).

As is typical with dynamic programming, the steps (here, edit operations) chosen for the optimal solution (here, for $d_{m,n}$) can be traced back after the dynamic programming matrix has been filled in. From $d_{m,n}$ we can trace back towards $d_{0,0}$ and re-evaluate the decisions taken for computing each $d_{i,j}$. There is no need to store explicit back-pointers towards the cells minimizing the edit distance recurrence (6.1). The traceback is also visualized in Example 6.4. Notice that the whole matrix is required for the traceback, but there is an algorithm to do the traceback in optimal $O(m+n)$ space (see Exercise 6.16).

Example 6.4 Compute the unit cost edit distance for $A =$ `tervetuloa` and $B =$ `teretulemast`, and visualize the optimal alignments.

Solution

We can simulate Algorithm 6.1 as shown below. Notice that the convention is to represent the first sequence $A = a_1 a_2 \cdots a_m$ vertically (and we iterate the index i from top to bottom) and the second sequence $B = b_1 b_2 \cdots b_n$ horizontally (and we iterate the index j from left to right).

		t	e	r	e	t	u	l	e	m	a	s	t
	0	1	2	3	4	5	6	7	8	9	10	11	12
t	1	0	1	2	3	4	5	6	7	8	9	10	11
e	2	1	0	1	2	3	4	5	6	7	8	9	10
r	3	2	1	0	1	2	3	4	5	6	7	8	9
v	4	3	2	1	1	2	3	4	5	6	7	8	9
e	5	4	3	2	1	2	3	4	4	5	6	7	8
t	6	5	4	3	2	1	2	3	4	5	6	7	7
u	7	6	5	4	3	2	1	2	3	4	5	6	7
l	8	7	6	5	4	3	2	1	2	3	4	5	6
o	9	8	7	6	5	4	3	2	2	3	4	5	6
a	10	9	8	7	6	5	4	3	3	3	3	4	5

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

The final solution is in the bottom-right corner, $D_{\mathsf{L}}(A,B) = 5$. The grayed area highlights the cells visited while tracing the optimal paths (alignments). There are two optimal alignments, shown below (the bottom-most path is on the left, and the top-most path is on the right):

```
tervetulo-a--  |  tervetul-oa--
ter-etulemast  |  ter-etulemast
```

6.1.2 Shortest detour

In the previous section we showed how the edit distance between the two sequences $A = a_1a_2\cdots a_m$ and $B = b_1b_2\cdots b_n$ (assume here that $m \leq n$) can be computed efficiently. We discovered a recurrence relation for $d_{m,n} = D(A,B)$, and gave an $\Theta(mn)$-time implementation. However, in practice we often expect to compute the edit distance between two strings that are similar, more precisely, for which $D(A,B)$ is fairly small compared with both n and m. In this section we give an algorithm exploiting this assumption, and running in time $O(dm)$, where $d = D(A,B)$.

Let an *optimal path* be a sequence of cells of the dynamic programming matrix (made up of some entries $d_{i,j}$). Consider a diagonal D^k of the dynamic programming matrix (recall its definition from (6.4)), and denote by $\Delta > 0$ the smallest cost of an insertion or deletion (which depends on the cost model chosen). The crucial observation here is that each time the optimal path changes diagonal, it incurs a cost of at least Δ. Hence, on an optimal path there can be at most $d_{m,n}/\Delta$ changes of diagonal when computing $d_{m,n}$. It is sufficient to limit the computation inside a *diagonal zone*, that includes diagonals $0,\ldots,n-m$ (since at least $n-m$ insertions into A are needed, or equivalently, $n-m$ deletions from B) and possibly some other diagonals. Since the value $d_{m,n}$ is not known beforehand, we actually cannot explicitly define the required diagonal zone beforehand.

Let us first consider the case where a threshold t is given for the distance and we have to answer the question of whether $d_{m,n} \leq t$, and, if so, what the actual value of $d_{m,n}$ is. First, if $t < (n-m)\Delta$, then the answer is "no". Otherwise, the solution for this problem limits the computation inside the diagonal zone $Z = [-x..n-m+x]$, where $x = \lceil t/(2\Delta) - (n-m)/2 \rceil$. Indeed, if the optimal path needs to go outside of this zone, then its cost is at least $((n-m)+2(x+1))\Delta > t$, and we can answer the question negatively. The running time of this algorithm is $O(((n-m)+2x)-m) = O((t/\Delta)m)$, which translates to $O(tm)$ for the unit cost edit distance in which $\Delta = 1$. Thus, we obtained an algorithm whose time complexity depends on an assumption of the edit distance between A and B. See Example 6.5.

Exercise 6.3 asks the reader how to allocate space only for those cells that are to be computed, since otherwise the space allocation would dominate the $O(tm)$ running time of this algorithm.

Example 6.5 Given $A =$ `tervetuloa` and $B =$ `teretulemast`, compute the diagonal zone of the edit distance matrix to decide whether $D_{\mathsf{L}}(A,B) \leq 5$.

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

Solution

The diagonal zone becomes $Z = [-2..4]$. The computation inside Z is shown below.

		t	e	r	e	t	u	l	e	m	a	s	t
	0	1	2	3	4								
t	1	0	1	2	3	4							
e	2	1	0	1	2	3	4						
r		2	1	0	1	2	3	4					
v			2	1	1	2	3	4	5				
e				2	1	2	3	4	4	5			
t					2	1	2	3	4	5	6		
u						2	1	2	3	4	5	6	
l							2	1	2	3	4	5	6
o								2	2	3	4	5	6
a									3	3	3	4	5

The final solution is in the bottom-right corner, $D_{\mathrm{L}}(A, B) = 5$, so the answer is that the edit distance is within the threshold.

Consider now the case when we are not given an upper bound on the edit distance. Then we can run the above algorithm for increasing values of t (starting for example with $t = 1$), such that each value is double the previous one. If this algorithm gives the answer "yes", then we also have the correct edit distance, and we stop. Otherwise, we keep doubling t until we have reached the maximum possible value, which is n. This algorithm is called the *doubling technique*.

The fact that the overall complexity is the desired $O((d/\Delta)m)$, where $d = d_{m,n} = D(A, B)$, can be shown as follows. Let $t = 2^T$ be the *last* iteration of the algorithm, thus when we have that $2^{T-1} < d \leq 2^T$. The overall running time of this doubling technique algorithm is then

$$O\left(\frac{m}{\Delta}(2^0 + 2^1 + \cdots + 2^T)\right).$$

Exploiting the geometric sum formula $\sum_{i=0}^{T} 2^i = 2^{T+1} - 1$, and the fact that $2^{T+1} < 4d$ (since $2^{T-1} < d$), this running time is

$$O\left(\frac{m}{\Delta}(2^0 + 2^1 + \cdots + 2^T)\right) = O\left(\frac{m}{\Delta}4d\right) = O\left(\frac{d}{\Delta}m\right).$$

This implies that, for any cost model in which the smallest cost of an insertion or deletion is greater than a fixed *constant*, the corresponding edit distance d can be computed in time $O(dm)$. In particular, we have the following corollary.

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

COROLLARY 6.3 *Given sequences $A = a_1 \cdots a_m$ and $B = b_1 \cdots b_n$ with $m \leq n$, if the unit cost edit (Levenshtein) distance $D_L(A, B)$ equals d, then $D_L(A, B)$ can be computed in time $O(dm)$.*

*6.1.3 Myers' bitparallel algorithm

The Levenshtein edit distance $D_L(A, B)$ has some properties that make it possible to derive an $O(\lceil m/w \rceil n)$-time algorithm, where w is the computer word size. Namely, the edit distance matrix $(d_{i,j})$ computed for $D_L(A, B)$ has the following properties:

> The diagonal property: $d_{i,j} - d_{i-1,j-1} \in \{0, 1\}$.
> The adjacency property: $d_{i,j} - d_{i,j-1} \in \{-1, 0, 1\}$ and $d_{i,j} - d_{i-1,j} \in \{-1, 0, 1\}$.

For now, we assume $|A| < w$; the general case is considered later. Then one can encode the columns of the matrix with a couple of bitvectors. We will need R_j^+ and R_j^- coding the row-wise differences, C_j^+ and C_j^- the column-wise differences, and D_j the diagonal-wise differences, at column j:

$$\begin{aligned}
R_j^+[i] = 1 \quad &\text{iff} \quad d_{i,j} - d_{i-1,j} = 1, \\
R_j^-[i] = 1 \quad &\text{iff} \quad d_{i,j} - d_{i-1,j} = -1, \\
C_j^+[i] = 1 \quad &\text{iff} \quad d_{i,j} - d_{i,j-1} = 1, \\
C_j^-[i] = 1 \quad &\text{iff} \quad d_{i,j} - d_{i,j-1} = -1, \\
D_j[i] \ = 1 \quad &\text{iff} \quad d_{i,j} - d_{i-1,j-1} = 1.
\end{aligned}$$

In the following, we assume that these vectors have already been computed for column $j-1$ and we want to derive the vectors of column j in *constant time*; this means that we need to find a way to use bitparallel operations (recall Exercise 2.6 from page 18). In addition to the vectors of the previous column, we need indicator bitvectors I^c having $I^c[i] = 1$ iff $A[i] = c$.

The update rules are shown in Algorithm 6.2. These should look magical until we go through the proof on why the values are computed correctly. To gather some intuition and familiarity with bitparallel notions, Example 6.6 simulates the algorithm.

Example 6.6 Simulate Algorithm 6.2 for computing the fourth column of the dynamic programming matrix in Example 6.4.

Solution

In the following visualization, the logical operators in column titles take one or two previous columns as inputs depending on the operator type and on the presence or absence of the bitvector name specified on the right-hand side. First, the part of the original matrix involved is shown and then the computation is depicted step by step. The top-most column titles mark the output bitvectors.

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

			$j=4$	Inputs		Previous column input to next column							D_j
	e	r	e	R^+_{j-1}	R^-_{j-1}	I^e	$\&R^+_{j-1}$	$+R^+_{j-1}$	$\oplus R^+_{j-1}$	$\mid I^e$	$\oplus R^-_{j-1}$	$\sim$	$\mid 1$
	2	3	4	0	0	0	0	0	0	0	0	1	1
t	1	2	3	0	1	0	0	0	0	0	1	0	0
e	0	1	2	0	1	1	0	0	0	1	1	0	0
r	1	0	1	0	1	0	0	0	0	0	1	0	0
v	2	1	1	1	0	0	0	1	0	0	0	1	1
e	3	2	1	1	0	1	1	0	1	1	1	0	0
t	4	3	2	1	0	0	0	0	1	1	1	0	0
u	5	4	3	1	0	0	0	0	1	1	1	0	0
l	6	5	4	1	0	0	0	0	1	1	1	0	0
o	7	6	5	1	0	0	0	0	1	1	1	0	0
a	8	7	6	1	0	0	0	0	1	1	1	0	0

Previous column input to next column			C^+_j		C^-_j
$\sim D_j$	$\mid R^+_{j-1}$	$\sim$	$\mid R^-_{j-1}$	$\sim D_j$	$\&R^+_{j-1}$
0	0	1	1	0	0
1	1	0	1	1	0
1	1	0	1	1	0
1	1	0	1	1	0
0	1	0	0	0	0
1	1	0	0	1	1
1	1	0	0	1	1
1	1	0	0	1	1
1	1	0	0	1	1
1	1	0	0	1	1
1	1	0	0	1	1

Previous column(s) input to next column					R^+_j			R^-_j
$\sim D_j$	$(C^+_j << 1)$	$\mid$	$\sim$	$C^-_j << 1$	$\mid$	$\sim D_j$	$(C^+_j << 1)$	$\&$
0	0	0	1	0	1	0	0	0
1	1	1	0	0	0	1	1	1
1	1	1	0	0	0	1	1	1
1	1	1	0	0	0	1	1	1
0	1	1	0	0	0	0	1	0
1	0	1	0	0	0	1	0	0
1	0	1	0	1	1	1	0	0
1	0	1	0	1	1	1	0	0
1	0	1	0	1	1	1	0	0
1	0	1	0	1	1	1	0	0
1	0	1	0	1	1	1	0	0

Algorithm 6.2: Myers' bitparallel algorithm for computing one column of edit distance computation on strings A and B, $|A| < w$

Input: Bitvectors $R^+_{j-1}, R^-_{j-1}, C^+_{j-1}, C^-_{j-1}, D_{j-1}, I^c$ for $c = B[j]$, $d = D(A, B_{1..j-1})$, and $m = |A|$.

Output: Edit distance $D(A, B_{1..j})$ and bitvectors encoding column j.

$D_j \leftarrow \sim(((I^c \,\&\, R^+_{j-1} + R^+_{j-1}) \oplus R^+_{j-1}) \mid I^c \mid R^-_{j-1}) \mid 1;$

$C^+_j \leftarrow R^-_{j-1} \mid \sim(\sim D_j \mid R^+_{j-1});$

$C^-_j \leftarrow \sim D_j \,\&\, R^+_{j-1};$

$R^+_j \leftarrow (C^-_j << 1) \mid \sim(\sim D_j \mid (C^+_j << 1));$

$R^-_j \leftarrow \sim D_j \,\&\, (C^+_j << 1);$

if $C^+[m] = 1$ **then**

  $d \leftarrow d + 1;$

if $C^-[m] = 1$ **then**

  $d \leftarrow d - 1;$

return *d and the bitvectors* $R^+_j, R^-_j, C^+_j, C^-_j, D_j$;

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

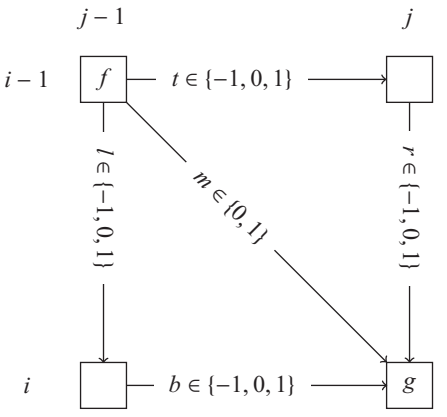


Figure 6.1 Difference DAG on four neighboring cells in a Levenshtein edit distance dynamic programming matrix.

In deriving Algorithm 6.2, we start with the four last update rules. For all of these, it suffices to consider how bits depend among cells $(i-1,j-1)$, $(i,j-1)$, $(i-1,j)$, and (i,j) and find bit-wise operation sets that implement the logical formulas simultaneously for all i. Instead of trying to mimic how these formulas were discovered in the first place, we follow a computerized approach. One could go over all total orders of the four variables for which we are trying to solve, through all logical formulas involving already computed variables, computing a truth table for each formula and the variable in question. Each combination of the input variables fixes some constraints on the edge weights in the *difference DAG* visualized in Figure 6.1.

As in Figure 6.1, we denote $f = d_{i-1,j-1}$, $l = d_{i,j-1} - d_{i-1,j-1}$, $t = d_{i-1,j} - d_{i-1,j-1}$, $m = d_{i,j} - d_{i-1,j-1}$, $b = d_{i,j} - d_{i,j-1}$, $r = d_{i,j} - d_{i-1,j}$, and $g = d_{i,j}$. We obtain equations $f + l + b = g$, $f + m = g$, and $f + t + r = g$ with the diagonal property and adjacency property as the initial constraints; each input value combination gives additional constraints, so that one can hope to solve for the variable in question. If the constraints define the output variable value on each input value combination and the logical formula in question gives the same answer, we have found a solution. Such a solution can be discovered in exponential time in the number of variables. With a constant number of variables, this is still feasible. Assuming the four update rules have been found in this way, it is sufficient to check the truth tables.

LEMMA 6.4 $\quad C_j^+ = R_{j-1}^- \mid \sim(\sim D_j \mid R_{j-1}^+) \mid 1$

Proof The satisfying truth table is shown below. Impossible input combinations, where $R_{j-1}^-[i]$ and $R_{j-1}^+[i]$ are 1 at the same time, are left out. The last column is derived by analyzing the constraints given by fixed input parameters. For example, the first row sets constraints $l \geq 0$, $f = g$, and $l \leq 0$ (see Figure 6.1). That is, one can solve b from $f + l + b = g$, giving $b = 0$ and thus $C_j^+[i] = 0$.

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

$x = R^-_{j-1}[i]$	$y = D_j[i]$	$z = R^+_{j-1}[i]$	$x \mid \sim(\sim y \mid z)$	$C^+_j[i]$
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
1	0	0	1	1
0	1	1	0	0
1	1	0	1	1

For $C^+_j[0]$ the formula is not defined, but the initialization $d_{0,j} = d_{0,j-1} + 1$ is enforced by $|1$. □

LEMMA 6.5 $C^-_j = \sim D_j \,\&\, R^+_{j-1}$

Proof The satisfying truth table is shown below.

$x = D_j[i]$	$y = R^+_{j-1}[i]$	$\sim x \,\&\, y$	$C^-_j[i]$
0	0	0	0
0	1	1	1
1	0	0	0
1	1	0	0

For $C^-_j[0]$ the formula is not defined, but the initialization $d_{0,j} = d_{0,j-1}+1$ is guaranteed if, for example, $D_j[0]$ is initialized to 1. □

LEMMA 6.6 $R^+_j = (C^-_j << 1) \mid \sim(\sim D_j \mid (C^+_j << 1))$

Proof The satisfying truth table is shown below. Impossible input combinations, where $C^-_j[i-1]$ and $C^+_j[i-1]$ are 1 at the same time, are left out.

$x = C^-_j[i-1]$	$y = D_j[i]$	$z = C^+_j[i-1]$	$x \mid \sim(\sim y \mid z)$	$R^+_j[i]$
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
1	0	0	1	1
0	1	1	0	0
1	1	0	1	1

For $R^+_j[0]$ the formula is not defined, so we can leave the bit-wise formula to set that bit arbitrarily. The formula uses values shifted by one, and this is correctly executed by the bit-wise left-shifts. □

LEMMA 6.7 $R^-_j = \sim D_j \,\&\, (C^+_j << 1)$

Proof The satisfying truth table is shown below.

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

$x = D_j[i]$	$y = C_j^+[i-1]$	$\sim x \,\&\, y$	$R_j^-[i]$
0	0	0	0
0	1	1	1
1	0	0	0
1	1	0	0

For $R_j^-[0]$ the formula is not defined, so we can leave the bit-wise formula to set that bit arbitrarily. The formula uses values shifted by one, and this is correctly executed by the bit-wise left-shift. □

For the first update rule, we could follow the same procedure as above using local dependencies between cells, but that would give us a circular dependency. Instead, we need to use a more global property of the values to break this cycle.

LEMMA 6.8 $D_j = \sim(((I^c \,\&\, R_{j-1}^+ + R_{j-1}^+) \oplus R_{j-1}^+) \mid I^c \mid R_{j-1}^-) \mid 1$

Proof Recall Figure 6.1. For $I^c[i] = 1$ we obtain the equation $f + m = f$, so $m = D_j[i] = 0$. For $R_{j-1}^- = 1$ we obtain the equation $f - 1 + l = g$. Combined with $l \leq 1$, $f + m = g$, and $m \geq 0$, the only solution is $m = D_j[i] = 0$. For i such that $I^c[i] = 0$ and $R_{j-1}^-[i] = 0$ we claim that

$$d_{i,j} = d_{i-1,j-1} \text{ iff } \exists h \,:\, I^c[h] = 1 \text{ and } R_{j-1}^+[q] = 1, \text{ for } q = [h..i-1]. \tag{6.5}$$

Assuming the right side holds, one can derive from $d_{h,j} = d_{h-1,j-1}$ and $d_{h,j-1} = d_{h-1,j-1} + 1$ that $d_{h,j} - d_{h,j-1} = -1$. On combining the latter with $d_{h+1,j} - d_{h,j} \leq 1$, one obtains $d_{h+1,j} = d_{h,j-1}$. Continuing in this way, one obtains $d_{i,j} = d_{i-1,j-1}$.

For the other direction, the assumptions $I^c[i] = 0$, $R_{j-1}^-[i] = 0$, and $d_{i,j} = d_{i-1,j-1}$ indicate that the optimal path to $d_{i,j}$ must come from $d_{i-1,j}$, and it follows that $d_{i-1,j} - d_{i-1,j-1} = -1$. Combining this with the diagonal and adjacency properties, we get that $d_{i-1,j-1} - d_{i-2,j-1} = 1$. If $I^c[i-1] = 0$, the optimal path to $d_{i-1,j}$ must come from $d_{i-2,j}$, and it follows that $d_{i-2,j} - d_{i-2,j-1} = -1$. Continuing in this way, there has to be h such that the optimal path to $d_{h,j}$ comes from $d_{h-1,j-1}$, that is, $I^c[h] = 1$: otherwise $d_{0,j} - d_{0,j-1} = -1$, which contradicts on how the matrix has been initialized.

Finally, the operation $I' = I^c \,\&\, R_{j-1}^+$ sets $I'[i] = 1$ iff $I^c[i] = 1$ and there is an overlapping run of 1-bits in R_{j-1}^+. The operation $P = I' + R_{j-1}^+$ propagates each $I'[i] = 1$ over the overlapping run of 1-bits in R_{j-1}^+, setting $P[i'] = 0$ for $i \leq i' \leq r$, and $P[r+1] = 1$, where r is the end of the run. Other parts of R_{j-1}^+ are copied to P as such. If there are multiple 1-bits in I^c overlapping the same run, all but the first location of those will be set to 1. The operation $P' = P \oplus C_{j-1}^+$ sets these back to 0, along with runs not overlapped with any $I^c[i] = 1$. In the end, exactly those indices i corresponding to Equation (6.5) have $P'[i] = 1$. With the complement operation, the claimed final formula follows, except for the $|1$; this sets $D_j[0] = 1$, which is required in order to have $C_j^-[0]$ computed correctly (see the proof of Lemma 6.5). □

It is straightforward to extend the algorithm for longer sequences by representing a column with arrays of bitvectors; one needs to additionally take care that the bits at the border are correctly updated.

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

THEOREM 6.9 *Given sequences $A = a_1 \cdots a_m$ and $B = b_1 \cdots b_n$, $m \leq n$, the edit distance $d = D_L(A, B)$ can be computed in time $O(\lceil m/w \rceil n)$ in the RAM model, where w is the computer word size.*

Adding traceback to Myers' algorithm for an optimal alignment is considered in Exercises 6.8 and 6.9.

6.2 Longest common subsequence

The edit distance in which the substitution operation is forbidden has a tight connection to the problem of finding a longest common subsequence of two sequences. Recall that a sequence $C = c_1 c_2 \cdots c_r$ is a *subsequence* of $A = a_1 a_2 \cdots a_m$, if C can be obtained by deleting zero or more characters from A.

Problem 6.7 Longest common subsequence (LCS)

Given two sequences A and B, find a sequence C that is a subsequence of both A and B, and is the longest one with this property. Any such sequence is called a *longest common subsequence* of A and B, and is denoted $\mathsf{LCS}(A, B)$.

Let $D_{\mathtt{id}}(A, B)$ be the (*indel*) edit distance with $c(a_i, b_j) = \infty$ if $a_i \neq b_j$, and $c(a_i, b_j) = 0$ if $a_i = b_j$, $c(a_i, -) = c(-, b_j) = 1$. Observe that, similarly to the general edit distance, also $D_{\mathtt{id}}(A, B)$ captures the minimum set of elementary operations needed to convert a sequence into any other sequence: any substitution can be simulated by a deletion followed by an insertion.

The connection between $D_{\mathtt{id}}(A, B)$ and $\mathsf{LCS}(A, B)$ is formalized in the theorem below, and is illustrated in Example 6.8.

THEOREM 6.10 $|\mathit{LCS}(A, B)| = (|A| + |B| - D_{\mathtt{id}}(A, B))/2$.

We leave the proof of this connection to the reader (Exercise 6.11).

Example 6.8 Simulate the computation of $D_{\mathtt{id}}(A, B)$ for $A = \mathtt{tervetuloa}$ and $B = \mathtt{teretulemast}$. Extract a longest common subsequence from an alignment of A and B.

Solution

The final solution is in the bottom-right corner, $D_{\mathtt{id}}(A, B) = 6$. Observe that $|\mathsf{LCS}(A, B)| = (|A| + |B| - D_{\mathtt{id}}(A, B))/2 = (10 + 12 - 6)/2 = 8$, as indicated by Theorem 6.10. The grayed area consists of the cells visited while tracing the optimal paths (alignments). One of the alignments is

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

		t	e	r	e	t	u	l	e	m	a	s	t
	0	1	2	3	4	5	6	7	8	9	10	11	12
t	1	0	1	2	3	4	5	6	7	8	9	10	11
e	2	1	0	1	2	3	4	5	6	7	8	9	10
r	3	2	1	0	1	2	3	4	5	6	7	8	9
v	4	3	2	1	2	3	4	5	6	7	8	9	10
e	5	4	3	2	1	2	3	4	5	6	7	8	9
t	6	5	4	3	2	1	2	3	4	5	6	7	8
u	7	6	5	4	3	2	1	2	3	4	5	6	7
l	8	7	6	5	4	3	2	1	2	3	4	5	6
o	9	8	7	6	5	4	3	2	3	4	5	6	7
a	10	9	8	7	6	5	4	3	4	5	6	7	8

```
tervetulo--a--
ter-etul-emast
```

Taking all the identities from the alignments, we have that

$$\mathsf{LCS}(\texttt{tervetuloa}, \texttt{teretulemast}) = \texttt{teretula}\,.$$

In fact, this is *the* longest common subsequence, as all optimal alignments give the same result.

In the next section we develop a sparse dynamic programming and invariant technique for computing the LCS. This method will also be deployed in Section 15.4 for aligning RNA transcripts to the genome.

6.2.1 Sparse dynamic programming

The goal in *sparse dynamic programming* is to compute only the cells of the dynamic programming matrix needed for obtaining the final value. Let $A = a_1 a_2 \cdots a_m$ and $B = b_1 b_2 \cdots b_n$ be two sequences, and denote by $M(A,B)$ the set of matching character pairs, that is, $M(A,B) = \{(i,j) \mid a_i = b_j\}$. We will here abbreviate $M(A,B)$ by M.

We will next show how to compute the edit distance $D_{\texttt{id}}$ in time proportional to the size of M.

LEMMA 6.11 *Let $A = a_1 a_2 \cdots a_m$ and $B = b_1 b_2 \cdots b_n$ be two sequences, and let $M = M(A,B) = \{(i,j) \mid a_i = b_j\} \cup \{(0,0)\}$. For all values $d_{i,j} = D_{\texttt{id}}(A_{1..i}, B_{1..j})$, with $(i,j) \in M$, it holds that*

$$d_{i,j} = \min\{d_{i',j'} + i - i' + j - j' - 2 \mid i' < i, j' < j, (i',j') \in M\}, \tag{6.6}$$

with the initialization $d_{0,0} = 0$. Moreover, we have that $D_{\texttt{id}}(A,B) = d_{m+1,n+1}$.

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

Proof Consider two consecutive characters of A in $\mathsf{LCS}(A,B)$, say $a_{i'}$ and a_i. They have counterparts $b_{j'}$ and b_j in B. The corresponding optimal alignment $D_{\mathtt{id}}(A,B)$ contains only deletions and insertions between the two identity operations $a_{i'} \to b_{j'}$ and $a_i \to b_j$. The smallest number of deletions and insertions to convert $A_{i'+1..i-1}$ into $B_{j'+1..j-1}$ is $\big(i-1-(i'+1)+1\big)+\big(j-1-(j'+1)+1\big) = i-i'+j-j'-2$. The recurrence considers, for all pairs (i,j) with $a_i = b_j$, all possible preceding pairs (i',j') with $a_{i'} = b_{j'}$. Among these, the pair that minimizes the overall cost of converting first $A_{1..i'}$ into $B_{1..j'}$ (cost $d_{i',j'}$) and then $A_{i'..i}$ into $B_{j'..j}$ (cost $i-i'+j-j'-2$) is chosen. The initialization is correct: if $a_i \to b_j$ is the first identity operation, the cost of converting $A_{1..i-1}$ into $B_{1..j-1}$ is $i-1+j-1 = d_{0,0}+i-0+j-0-2 = d_{i,j}$.

If we assume by induction that all values $d_{i',j'}$, $i' < i, j' < j$, are computed correctly, then it follows that each $d_{i,j}$ receives the correct value. Finally, $d_{m+1,n+1} = D_{\mathtt{id}}(A,B)$, because, if (i',j') is the last identity in an optimal alignment, converting $A_{i'+1..m}$ into $B_{j'+1..n}$ costs $m-i'+n-j'$, which equals $d_{m+1,n+1} - d_{i',j'}$. □

A naive implementation of the recurrence (6.6) computes each $d_{i,j}$ in time $O(|M|)$, by just scanning M and selecting the optimal pair (i',j'). Let us now see how we can improve this time to $O(\log m)$. Observe that the terms i, j, and -2 in the minimization (6.6) do not depend on the pairs $(i',j') \in M$ with $i' < i, j' < j$. Therefore, we can rewrite (6.6) as

$$d_{i,j} = i+j-2+\min\{d_{i',j'} - i' - j' \mid i' < i, j' < j, (i',j') \in M\},$$

by bringing all such *invariant* values out from the minimization. Compute now all values $d_{i,j}$ in *reverse column-order*, namely in the order $<^{\mathtt{rc}}$ such that

$$(i',j') <^{\mathtt{rc}} (i,j) \text{ if and only if } j' < j \text{ or } (j' = j \text{ and } i' > i).$$

The crucial property of this order is that, if all the values $d_{i',j'}$, with $(i',j') <^{\mathtt{rc}} (i,j)$, *and only them*, are computed before computing $d_{i,j}$, then the condition $j' < j$ is automatically satisfied. Thus, among these values computed so far in reverse column-order, we need select only those pairs with $i' < i$. We can write the recurrence as

$$d_{i,j} = i+j-2+\min\{d_{i',j'} - i' - j' \mid i' < i, (i',j') <^{\mathtt{rc}} (i,j), (i',j') \in M\}. \tag{6.7}$$

We can then consider the cells $(i,j) \in M$ in the reverse column-order, and, after having computed $d_{i,j}$ we store the value $d_{i,j} - i - j$ in a range minimum query data structure $\mathcal{T}$ of Lemma 3.1 on page 20, with key i. Computing $d_{i,j}$ itself can then be done by retrieving the minimum value d from $\mathcal{T}$, corresponding to a pair (i',j') with $i' < i$. Then, $d_{i,j}$ is obtained as $d_{i,j} = i+j-2+d$, since we need to add to d the invariant terms $i+j-2$. Figure 6.2 illustrates this algorithm, whose pseudocode is shown in Algorithm 6.3. The derivation used here is called the *invariant technique*.

THEOREM 6.12 *Given sequences $A = a_1 a_2 \cdots a_m$ and $B = b_1 b_2 \cdots b_n$ from an ordered alphabet, the edit distance $D_{\mathtt{id}}$ and $|\mathsf{LCS}(A,B)|$ can be computed using Algorithm 6.3 in time $O((m+n+|M|)\log m)$ and space $O(m+|M|)$, where $M = \{(i,j) \mid a_i = b_j\}$.*

Proof The reverse column-order guarantees that the call $\mathcal{T}.\mathsf{RMQ}(0,i-1)$ corresponds to taking the minimum in Equation (6.7). The only difference is that $\mathcal{T}$ contains only

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

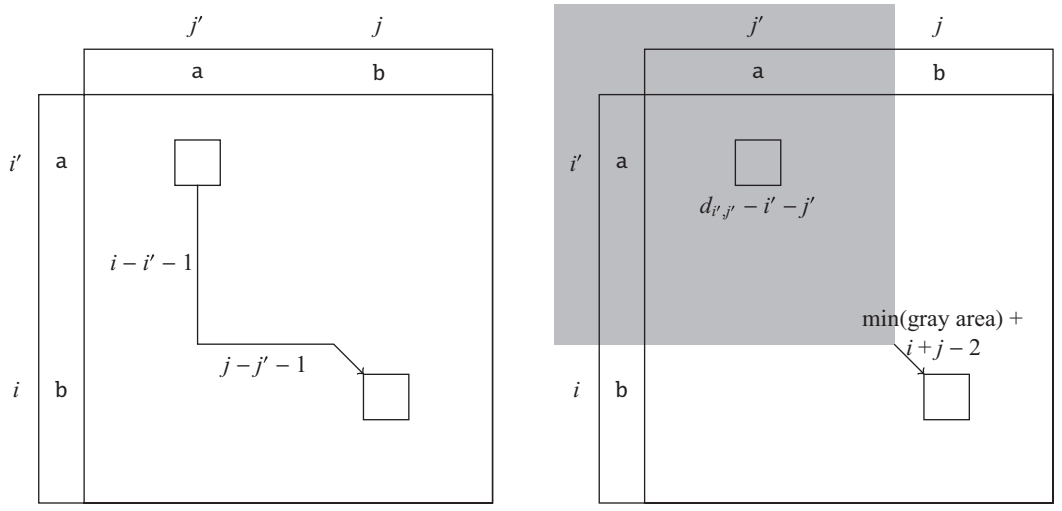


Figure 6.2 Geometric interpretation of Equation (6.7). On the left, we show a path from (i',j') to (i,j) such that in $\mathsf{LCS}(A,B)$ we have the consecutive identities $a_{i'} \to b_{j'}$ and $a_i \to b_j$ (with $a_{i'} = \mathtt{a}$ and $a_i = \mathtt{b}$). On the right, the invariant value is stored at (i',j') and the grayed area depicts from where the minimum value is taken for computing the cell (i,j). After taking the minimum, the actual minimum cost is computed by adding the invariant.

Algorithm 6.3: Sparse dynamic programming for edit distance $D_{\mathtt{id}}(A,B)$

Input: Sequences $A = a_1 a_2 \cdots a_m$ and $B = b_1 b_2 \cdots b_n$.
Output: Edit distance $D_{\mathtt{id}}(A,B)$.

```
1 Construct set M = {(i,j) | a_i = b_j} and sort it into an array M[1..|M|] in reverse
  column-order;
2 Initialize a search tree T with keys 0, 1, ..., m;
3 T.update(0, 0);
4 for p ← 1 to |M| do
5 |   (i, j) ← M[p];
6 |   d ← i + j − 2 + T.RMQ(0, i − 1);
7 |   T.update(i, d − i − j);
8 return T.RMQ(0, m) + m + n;
```

the smallest value at row i', which does not affect the correctness. The algorithm calls $O(|M|)$ times the operations in $\mathcal{T}$. Each of these takes $O(\log m)$ time. Finally, set M can be easily constructed in $O(\sigma + m + n + |M|)$ time on an alphabet $[1..\sigma]$, or in $O((m+n)\log m + |M|)$ time on an ordered alphabet: see Exercise 6.12. □

6.3 Approximate string matching

So far we have developed methods to compare two entire sequences. A more common scenario in biological sequence analysis is that one of the sequences represents the

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

whole genome (as a concatenation of chromosome sequences) and the other sequence is a short extract from another genome. A simple example is a *homology search* in prokaryotes, where the short extract could be a contiguous gene from one species, and one wants to check whether the genome of another species contains a similar gene. This problem is captured as follows.

Let $S = s_1 s_2 \cdots s_n \in \Sigma^*$ be a *text* string and $P = p_1 p_2 \cdots p_m \in \Sigma^*$ a *pattern* string. Let k be a constant, $0 \leq k \leq m$. Consider the following search problems.

Problem 6.9 Approximate string matching under k mismatches

Search for all substrings X of S, with $|X| = |P|$, that differ from P in at most k positions, namely $D_{\mathsf{H}}(P, X) \leq k$.

Problem 6.10 Approximate string matching under k errors

Search for all the ending positions of substrings X of S for which $D_{\mathsf{L}}(P, X) \leq k$ holds.

In both problems, instead of fixing k, one could also take as input an error level α, and set k as αm. Recall Equation (6.1) for the edit distance. It turns out that only a minimal modification is required in order to solve Problem 6.10: initialize $d_{0,j} = 0$ for all $0 \leq j \leq n$, and consider the values $d_{i,j}$ otherwise computed as in Equation (6.1).

LEMMA 6.13 *Suppose that an optimal path ending at some $d_{m,j}$ (that is, $d_{m,j}$ equals the minimum value on row m) starts at some $d_{0,r} = 0$. Then $D_{\mathsf{L}}(P, s_{r+1} s_{r+2} \cdots s_j) = d_{m,j}$ and $d_{m,j} = \min\{D_{\mathsf{L}}(P, s_t s_{t+1} \cdots s_j) \mid t \leq j\}$.*

Proof Assume for a contradiction that there is $t \neq r+1$ for which $d = D_{\mathsf{L}}(P, s_t \cdots s_j) < D_{\mathsf{L}}(P, s_{r+1} \cdots s_j)$. It follows that the optimal path from $d_{0,t-1}$ to $d_{m,j}$ has cost $d < d_{m,j}$, which is a contradiction, since $d_{m,j}$ is assumed to be the cost of the optimal path. □

The optimal path to $d_{m,j}$ is visualized in Figure 6.3.

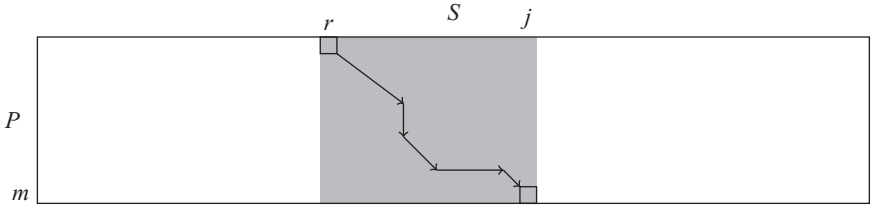


Figure 6.3 An optimal path in the dynamic programming matrix for solving approximate string matching under k errors (Problem 6.10).

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

THEOREM 6.14 *Approximate string matching under k errors (Problem 6.10) can be solved in $O(mn)$ time and $O(m)$ space, when the outputs are the occurrence end positions, and in space $O(m^2)$, if the start positions of the occurrences are also to be reported.*

Proof Modify Algorithm 6.1 to take inputs P and S in place of A and B. Initialize instead $d_{0,j} = 0$, for all $0 \leq j \leq n$. After computing $d_{m,j}$ for column j, report j as the ending position of a k-errors match if $d_{m,j} \leq k$. As in Exercise 6.2, the space of the algorithm can be improved to $O(m)$ by storing values only from two consecutive columns at a time.

To find an occurrence start position, one should keep the part of the matrix containing an optimal path ending at each column j with $d_{m,j} \leq k$. Notice that such an optimal path must start at least at column $j - m - k$, so an $O(m^2)$ size portion of the matrix is required. When proceeding column-by-column from left to right, one can maintain the required part of the matrix in a circular buffer. □

This simple modification of edit distance computation for the approximate matching solution is called the *first-row-to-zero* trick.

It is worth remarking that, with a different initialization, Myers' bitparallel algorithm can also be employed to solve the k-errors problem.

COROLLARY 6.15 *Approximate string matching under k errors (Problem 6.10) limited to the Levenshtein edit distance can be solved in $O(\lceil m/w \rceil n)$ time, when the outputs are the occurrence end positions, in the RAM model of computation, where w is the computer word size.*

The proof of this result is left as an exercise for the reader (Exercise 6.4). In Chapter 10 we will study more advanced approximate string matching techniques.

6.4 Biological sequence alignment

As discussed earlier, an alignment of two biological sequences is typically defined through a maximum score alignment, rather than a minimum-cost one, as in edit distance. Algorithm-wise, the differences are minimal, but the maximization framework allows a probabilistic interpretation for alignments. Let us define $s(a, b)$ to be the *score* of aligning character a with character b (that is, of substituting a with b). Let δ be the *penalty* of an *indel*, that is, the penalty for inserting or deleting a character. Then the weight of an alignment is defined by the sum of substitution scores minus the sum of the indel penalties in it.

For example, the following *substitution matrix* is often used for DNA sequence alignments:

$s(a,b)$	A	C	G	T
A	1	−1	−0.5	−1
C	−1	1	−1	−0.5
G	−0.5	−1	1	−1
T	−1	−0.5	−1	1

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

The reason for these numerical values is that so-called *transition* mutations (here with score -0.5) are twice as frequent as so-called *transversions* (here with score -1).

For example, if $\delta = 1$, then the weight, or total score, of the alignment

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

is $1 - 1 + 1 - 1 + 1 - 1 + 1 - 0.5 = 0.5$.

6.4.1 Global alignment

Global alignment is the dual of edit distance computation, with minimization replaced by maximization, defined as follows. The *weight*, or *score*, of an alignment U, L is

$$W(U, L) = \sum_{i=1}^{h} s(u_i, l_i), \tag{6.8}$$

where $s(a, -) = s(-, b) = -\delta$.

Problem 6.11 Global alignment

Given two sequences $A = a_1 \cdots a_m \in \Sigma^*$ and $B = b_1 \cdots b_n \in \Sigma^*$, find an alignment of A and B having the maximum score, that is, having the score

$$S(A, B) = \max_{(U,L) \in \mathcal{A}(A,B)} W(U, L),$$

where $\mathcal{A}(A, B)$ denotes the set of all valid alignments of A and B.

Let us denote $s_{i,j} = S(a_1 \cdots a_i, b_1 \cdots b_j)$, for all $0 \le i \le m$, $0 \le j \le n$.

THEOREM 6.16 *The global alignment scores $s_{i,j}$, and, in particular, $s_{m,n} = S(A, B)$, can be computed using the recurrence*

$$s_{i,j} = \max\{s_{i-1,j-1} + s(a_i, b_j), s_{i-1,j} - \delta, s_{i,j-1} - \delta\}, \tag{6.9}$$

for all $1 \le i \le m$, $1 \le j \le n$, and using the initializations

$$\begin{aligned} s_{0,0} &= 0, \\ s_{i,0} &= -i\delta, \text{ for all } 1 \le i \le m, \text{ and} \\ s_{0,j} &= -j\delta, \text{ for all } 1 \le j \le n. \end{aligned}$$

The correctness proof is identical to the corresponding theorem on edit distance. For completeness, a pseudocode is given in Algorithm 6.4. The algorithm is also called *Needleman-Wunsch*, after its inventors.

Tracing back the optimal alignment(s) is identical to the corresponding procedure for edit distances.

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

Algorithm 6.4: Global alignment using dynamic programming

Input: Sequences $A = a_1 a_2 \cdots a_m$ and $B = b_1 b_2 \cdots b_n$.
Output: Global alignment score $S(A, B)$.

$s_{0,0} = 0$;
for $i \leftarrow 1$ ***to*** m **do**
  $s_{i,0} \leftarrow s_{i-1,0} - \delta$;
for $j \leftarrow 1$ ***to*** n **do**
  $s_{0,j} \leftarrow s_{0,j-1} - \delta$;
for $j \leftarrow 1$ ***to*** n **do**
  for $i \leftarrow 1$ ***to*** m **do**
    $s_{i,j} \leftarrow \max\{s_{i-1,j-1} + s(a_i, b_j), s_{i-1,j} - \delta, s_{i,j-1} - \delta\}$;
return $s_{m,n}$;

6.4.2 Local alignment

An approximate pattern matching variant of global alignment is called *semi-local alignment*. This is the problem of finding the substring of a long sequence S having the highest scoring alignment with shorter sequence A. With the limitation of searching for only semi-local alignments with positive scores, one can use the first-row-to-zero trick from Section 6.3 in the global alignment computation. Then, the best-scoring semi-local alignments can be found analogously as explained in Section 6.3.

Continuing the example of a homology search in prokaryotes, we assume we have the full genomes of two different species. To identify potentially homologous genes, we should be able to identify a gene in one species in order to apply approximate string matching/semi-local alignment. However, there is a way to do the same thing without resorting to gene annotation, through *local alignment*, defined next.

Problem 6.12 Local alignment

Compute the maximum-scoring global alignment $L(A, B)$ over all substrings of $A = a_1 \cdots a_m$ and $B = b_1 \cdots b_n$, that is,

$$L(A, B) = \max\left\{S(A_{i'..i}, B_{j'..j}) \mid \text{for all } i', i \in [1..m], j', j \in [1..n] \text{ with } i' \leq i, j' \leq j\right\}.$$

The pair $(A_{i'..i}, B_{j'..j})$ with $L(A, B) = S(A_{i'..i}, B_{j'..j})$ can be interpreted as a pair of "genes" with highest similarity in the homology search example. One can modify the problem statement into outputting all substring pairs (A', B') with high enough similarity, which constitute the sufficiently good candidates for homologies.

For an arbitrary scoring scheme, this problem can be solved by applying global alignment for all suffix pairs from A and B in $O((mn)^2)$ time. However, as we learn in

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

Insight 6.1, the scoring schemes can be designed so that local alignments with score less than zero are not statistically significant, and therefore one can limit the consideration to non-zero-scoring local alignments. To compute such non-zero-scoring alignments, one can use a version of the first-row-to-zero trick: use the global alignment recurrence, but add an option to start a new alignment at any suffix/suffix pair by assigning score 0 for an empty alignment. This observation is the *Smith–Waterman* algorithm for local alignment.

Insight 6.1 Derivation of substitution scores

Local alignments exploit some properties of substitution matrices, which we derive next. For clarity, we limit the discussion to the case of trivial alignments with the only allowed mutation being substitution. Consider now that two equal-length strings A and B are generated identically and independently by a *random* model R with probability q_a for each character $a \in \Sigma$. Then the joint probability under R for the two sequences A and B is

$$\mathbb{P}(A, B \mid R) = \prod_i q_{a_i} q_{b_i}. \tag{6.10}$$

Now, consider the *match* model M, which assigns a joint probability p_{ab} for a substitution $a \to b$. This can be seen as the probability that a and b have a common ancestor in evolution. Then the probability of the trivial alignment is

$$\mathbb{P}(A, B \mid M) = \prod_i p_{a_i b_i}. \tag{6.11}$$

The ratio $\mathbb{P}(A, B \mid M)/P(A, B \mid R)$ is known as the *odds ratio*. To obtain an additive scoring scheme, the product is transformed into summation by taking logarithm, and the result is known as the *log-odds ratio*:

$$S(A, B) = \sum_i s(a_i, b_i), \tag{6.12}$$

where

$$s(a, b) = \log\left(\frac{p_{ab}}{q_a q_b}\right). \tag{6.13}$$

This scoring matrix has the property that the expected score of a random match at any position is negative:

$$\sum_{a,b} q_a q_b s(a, b) < 0. \tag{6.14}$$

This can be seen by noticing that

$$\sum_{a,b} q_a q_b s(a, b) = -\sum_{a,b} q_a q_b \log\left(\frac{q_a q_b}{p_{ab}}\right) = -\mathsf{H}(q^2 \mid p), \tag{6.15}$$

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

where $\mathsf{H}(q^2 \mid p)$ is the *relative entropy* (or *Kullback–Leibler divergence*) of distribution $q^2 = q \times q$ with respect to distribution p. The value of $\mathsf{H}(q^2 \mid p)$ is always positive unless $q^2 = p$ (see Exercise 6.19).

A straightforward approach for estimating probabilities p_{ab} is to take a set of well-trusted alignments and just count the frequencies of substitutions $a \to b$. One can find in the literature more advanced methods that take into account different evolutionary divergence times, that is, the fact that alignment of a to b could happen through a sequence of substitutions $a \to c \to \cdots \to b$.

Let us denote $l_{i,j} = \max\{S(a_{i'} \cdots a_i, b_{j'} \cdots b_j) \mid i' \le i, j' \le j\}$, for all $0 \le i \le m$, $0 \le j \le n$.

THEOREM 6.17 *The non-negative local alignment scores $l_{i,j}$ can be computed using the recurrence*

$$l_{i,j} = \max\{0, l_{i-1,j-1} + s(a_i, b_j), l_{i-1,j} - \delta, l_{i,j-1} - \delta\}, \tag{6.16}$$

for all $1 \le i \le m$, $1 \le j \le n$, and using the initializations

$$\begin{aligned} l_{0,0} &= 0, \\ l_{i,0} &= 0, \textit{ for all } 1 \le i \le m, \textit{ and} \\ l_{0,j} &= 0, \textit{ for all } 1 \le j \le n. \end{aligned}$$

It is easy to modify Algorithm 6.4 for computing the values $l_{i,j}$. After the computation, one can either locate $\max l_{i,j}$ or maintain the maximum value encountered during the computation. In the latter case, $O(m)$ space is sufficient for finding the maximum (computing the recurrence column-by-column). To trace back the alignment, one cannot usually afford $O(mn)$ space to store the matrix. Instead, one can easily modify the dynamic programming algorithm to maintain, in addition to the maximum score, also the left-most positions in A and B, say i' and j', respectively, where a maximum-scoring alignment ending at $l_{i,j}$ starts. Then it is easy to recompute a matrix of size $(i - i' + 1) \times (j - j' + 1)$ to output (all) local alignment(s) with maximum score ending at $l_{i,j}$. The space is hence quadratic in the length of the longest local alignment with maximum score. Exercise 6.15 asks you to develop the details of this idea and Exercise 6.16 asks for an optimal space solution.

Figure 6.4 illustrates the differences of global, local, semi-local, and overlap alignment, the last of which is our next topic.

6.4.3 Overlap alignment

An *overlap alignment* between two strings ignores the prefix of the first string and the suffix of the second one, as long as the remaining two (suffix–prefix overlapping) substrings have a good alignment. Computing overlap alignments is important for *fragment assembly*, a problem in which a genomic sequence is reconstructed, or *assembled*, from only a random subset of its substrings. Many assembly methods are based on a so-called

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

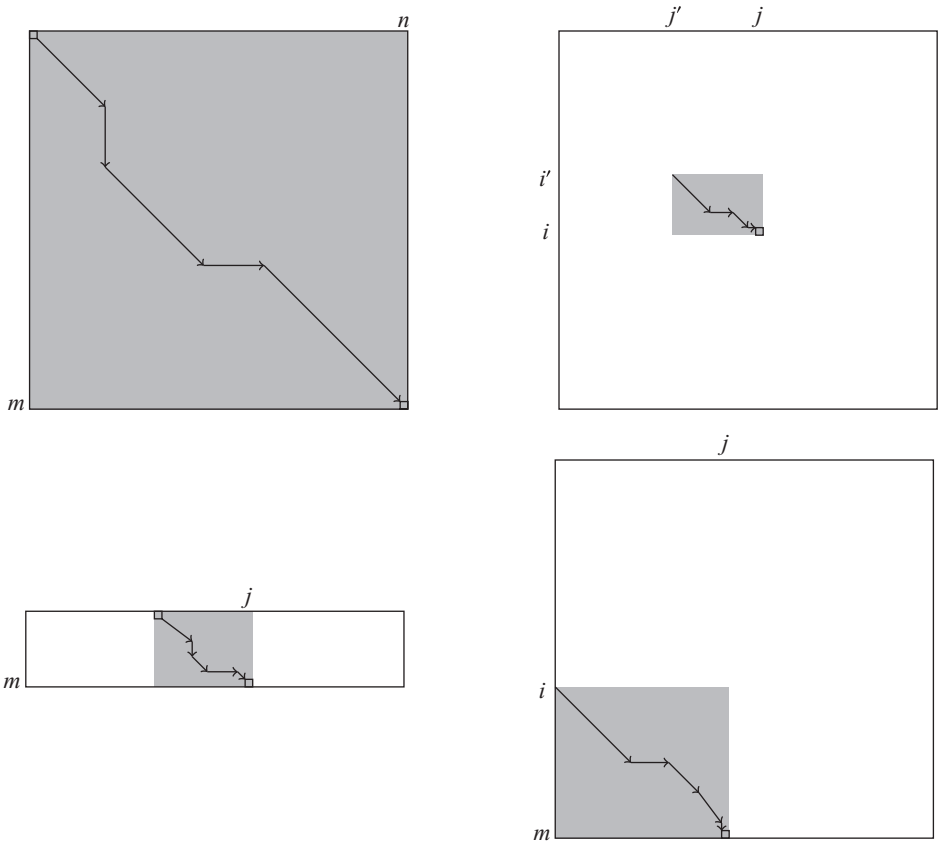


Figure 6.4 Summary of interpretations of global (top-left), local (top-right), semi-local (bottom-left), and overlap alignment (bottom-right).

overlap graph, in which these substrings are vertices, and in which two substrings are connected by an arc if they have an overlap alignment with a high enough score. Paths in this overlap graph represent possible partial assemblies of the originating sequence. In Chapter 13 we will study the fragment assembly problem in greater detail.

Problem 6.13 Overlap alignment

Compute the maximum-scoring global alignment $O(A, B)$ between the suffixes of A and the prefixes of B, that is,

$$O(A, B) = \max \left\{ S(A_{i..m}, B_{1..j}) \mid \text{for all } i \in [1..m], j \in [1..n] \right\}.$$

Like in the local alignment problem, we are interested only in alignments with score greater than zero. It is easy to see that a variant of the first-row-to-zero trick works again, as follows.

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

Let us denote $o_{i,j} = \max\{S(a_{i'} \cdots a_i, b_1 \cdots b_j) \mid i' \leq i\}$, for all $0 \leq i \leq m$, $0 \leq j \leq n$.

THEOREM 6.18 *The non-negative overlap alignment scores $o_{i,j}$ can be computed using the recurrence*

$$o_{i,j} = \max\{o_{i-1,j-1} + s(a_i, b_j), o_{i-1,j} - \delta, o_{i,j-1} - \delta\}, \tag{6.17}$$

for all $1 \leq i \leq m$, $1 \leq j \leq n$, and using the initializations

$$o_{0,0} = 0,$$
$$o_{i,0} = 0, \textit{ for all } 1 \leq i \leq m, \textit{ and}$$
$$o_{0,j} = -\delta j, \textit{ for all } 1 \leq j \leq n.$$

Again, the modifications to Algorithm 6.4 are minimal. Figure 6.4 illustrates the idea behind the computation. We shall study overlap alignments further in Section 10.4 and in Chapter 13.

6.4.4 Affine gap scores

In biological sequence evolution, indels often occur in blocks, and the simple *linear gap model* scoring scheme studied so far, with a penalty δ for each inserted or deleted base, is not well grounded. We cover here a more realistic scheme, called the *affine gap model*, which also admits efficient algorithms.

Recall the definition of alignment through sequences U and L containing A and B as subsequences, respectively, together with some gap characters $-$. We say that $U[l..r], L[l..r]$ is a *run of gaps* in the alignment U, L if

$$U[i] = -, \text{ or } L[i] = -, \text{ for all } i \in [l..r].$$

Symmetrically, we say that $U[l..r], L[l..r]$ is a *run of matches* in the alignment U, L if

$$U[i] \neq - \text{ and } L[i] \neq - \text{ for all } i \in [l..r].$$

Notice that a "match" refers here both to an identity and to a substitution, differing from the meaning we attributed to it in the context of longest common subsequences.

Any alignment can be partitioned into a sequence of runs of gaps and runs of matches. Let $U = U^1 U^2 \cdots U^k$ and $L = L^1 L^2 \cdots L^k$ denote such a partitioning, where, for all $i \in [1..k]$, we have $|U^i| = |L^i|$, and U^i, L^i is either a run of gaps or a run of matches. Let $\mathcal{P}(U, L)$ denote the set of all possible such partitions. Let (see Example 6.14)

$$W_G(U^i, L^i) = \begin{cases} W(U^i, L^i) & \text{if } U^i, L^i \text{ is a run of matches,} \\ -\alpha - \beta(|U^i| - 1) & \text{if } U^i, L^i \text{ is a run of gaps,} \end{cases}$$

where we recall that $W(U^i, L^i)$ is the global alignment score defined in (6.8). In the above definition, α is the penalty for opening a gap, and β is the penalty for extending a gap. In practice, α is chosen to be greater than β, the idea being that starting a gap always involves significant costs, but making it longer is cheap.

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

Example 6.14 Consider the following alignment U, L of $A = \texttt{ACTCGATC}$ and $B = \texttt{ACGTGAATGAT}$, where we have marked a partition of U into $U^1U^2U^3U^4$ and of L into $L^1L^2L^3L^4$:

$$\begin{array}{rcccc} & U^1 & U^2 & U^3 & U^4 \\ U = & \texttt{ACT} & \texttt{---} & \texttt{CCATC} & \texttt{--} \\ L = & \texttt{ACGTGA} & \texttt{--} & \texttt{ATC} & \texttt{AT} \\ & L^1 & L^2 & L^3 & L^4 \end{array}$$

We have

- $W_G(U^1, L^1) = s(\texttt{T}, \texttt{G})$,
- $W_G(U^2, L^2) = -\alpha - 4\beta$,
- $W_G(U^3, L^3) = 0$,
- $W_G(U^4, L^4) = -\alpha - \beta$.

The following problem asks for a best global alignment under affine gap scores.

Problem 6.15 Global alignment under affine gap scores

Given two sequences $A = a_1 \cdots a_m \in \Sigma^*$ and $B = b_1 \cdots b_n \in \Sigma^*$, find an alignment of A and B having maximum score under the affine gap model, that is, having score

$$S_G(A, B) = \max_{(U,L) \in \mathcal{A}(A,B)} \max \left\{ \sum_{i=1}^{k} W_G(U^i, L^i) \mid (U^1 \cdots U^k, L^1 \cdots L^k) \in \mathcal{P}(U, L) \right\},$$

where $\mathcal{A}(A, B)$ denotes the set of all valid alignments of A and B.

Let us denote $\texttt{sm}_{i,j} = S_G(a_1 \cdots a_{i-1}, b_1 \cdots b_{j-1}) + s(a_i, b_j)$ for all $0 \le i \le m$, $0 \le j \le n$, where we interpret $s(a_0, b_0) = s(a_{m+1}, s_{n+1}) = 0$. In particular, $S_G(A, B) = \texttt{sm}_{m+1,n+1}$. That is, we aim to store the optimal alignment score for each prefix pair over all alignments ending with a match.

THEOREM 6.19 *The scores* $\texttt{sm}_{i,j}$*, and in particular the global alignment under affine gap model* $S_G(A, B) = \texttt{sm}_{m+1,n+1}$*, can be computed using the recurrence*

$$\texttt{sm}_{i,j} = \max \Big(\texttt{sm}_{i-1,j-1} + s(a_i, b_j), \max_{i' < i, j' < j, (i',j') \neq (i-1,j-1)} \{ \texttt{sm}_{i',j'} - \alpha - \beta(j - j' + i - i' - 3) + s(a_i, b_j) \} \Big), \tag{6.18}$$

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

for all $1 \le i \le m$, $1 \le j \le n$, *and using the initializations*

$$\begin{aligned} \mathtt{sm}_{0,0} &= 0, \\ \mathtt{sm}_{i,0} &= -\infty, \textit{ for all } 1 \le i \le m, \textit{ and} \\ \mathtt{sm}_{0,j} &= -\infty, \textit{ for all } 1 \le j \le n. \end{aligned}$$

Proof First, note that the initializations are correct, since no alignment with an empty string can end with a match. By definition, the alignment score for two empty strings is 0.

Assume by induction that the recurrence is correct for all i', j' such that $i' < i, j' < j$. In any alignment of $a_1 \cdots a_i$ and $b_1 \cdots b_j$ ending with the match $a_i \to b_j$, there must be some previous match $a_{i'} \to b_{j'}$ (possibly the conceptual initialization $a_0 \to b_0$). The run of gaps between the matches $(a_{i'}, b_{j'})$ and (a_i, b_j) has score $-\alpha - \beta(j - j' + i - i' - 3)$, since the gap length is $j - 1 - (j' + 1) + 1 + i - 1 - (i' + 1) + 1 = j - j' + i - i' - 2$. An exception is the gap of length 0 that is taken into account separately by the term $\mathtt{sm}_{i-1,j-1} + s(a_i, b_j)$. We take the maximum over all such possible shorter alignments ending with a match, adding the score of the gap, or the score $s(a_i, b_j)$. □

The running time for computing $S_G(A, B)$ with the recurrence in Equation (6.18) is $O((mn)^2)$.

Notice that this recurrence is general, in the sense that $-\alpha - \beta(j - j' + i - i' - 3)$ can be replaced with any other function on the gap length. For example, it is easy to modify the recurrence to cope with various alternative definitions of a "run of gaps". One could separately define a run of insertions and a run of deletions. In fact, this definition is more relevant in the context of biological sequence evolution, since insertions and deletions are clearly separate events. For this alternative definition, the recurrence changes so that one takes separately $\max_{i' < i} \mathtt{sa}_{i',j} - \alpha - \beta(i - i' - 1)$ and $\max_{j' < j} \mathtt{sa}_{i,j'} - \alpha - \beta(j - j' - 1)$, where $\mathtt{sa}_{i,j}$ stores the optimal alignment score over all alignments (not just those ending with a match). The full details are left to the reader. The running time then reduces to $O(mn(m + n))$.

In what follows, we will continue with the general definition, and ask the reader to consider how to modify the algorithms for separate runs of insertions and deletions.

There are several ways to speed up the affine gap model recurrences. The most common one is *Gotoh's algorithm*, which fills two or more matrices simultaneously, one for alignments ending at a match state and the others for alignments ending inside a run of gaps. Let

$$S_G(A, B | \text{match}) = \max_{(U,L) \in \mathcal{A}_{\mathcal{M}}(A,B)} \max \left\{ \sum_{i=1}^{k} W_G(U^i, L^i) \,|\, (U^1 \cdots U^k, L^1 \cdots L^k) \in \mathcal{P}(U, L) \right\},$$

where $\mathcal{A}_{\mathcal{M}}(A, B)$ denotes the set of all valid alignments of A and B ending with a match. Let

$$S_G(A, B | \text{ gap}) = \max_{(U,L) \in \mathcal{A}_{\mathcal{G}}(A,B)} \max \left\{ \sum_{i=1}^{k} W_G(U^i, L^i) \mid (U^1 \cdots U^k, L^1 \cdots L^k) \in \mathcal{P}(U, L) \right\},$$

where $\mathcal{A}_{\mathcal{G}}(A, B)$ denotes the set of all valid alignments of A and B ending with a gap.

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

Then the value $\mathtt{sm}_{i,j}$ we defined earlier equals $S_G(A_{1..i}, B_{1..j} \mid \text{match})$, and we also let $\mathtt{sg}_{i,j}$ denote $S_G(A_{1..i}, B_{1..j} \mid \text{gap})$. It holds that $S_G(A_{1..i}, B_{1..j}) = \max(\mathtt{sm}_{i,j}, \mathtt{sg}_{i,j})$.

THEOREM 6.20 *The scores* $\mathtt{sm}_{i,j}$ *and* $\mathtt{sg}_{i,j}$*, and in particular the global alignment under affine gap model* $S_G(A,B) = \mathtt{sm}_{m+1,n+1} = \max(\mathtt{sm}_{m,n}, \mathtt{sg}_{m,n})$*, can be computed using the recurrences*

$$\begin{aligned} \mathtt{sm}_{i,j} &= \max\{\mathtt{sm}_{i-1,j-1} + s(a_i, b_j), \mathtt{sg}_{i-1,j-1} + s(a_i, b_j)\}, \\ \mathtt{sg}_{i,j} &= \max\{\mathtt{sm}_{i-1,j} - \alpha, \mathtt{sg}_{i-1,j} - \beta, \mathtt{sm}_{i,j-1} - \alpha, \mathtt{sg}_{i,j-1} - \beta\}, \end{aligned} \tag{6.19}$$

where $1 \le i \le m$*,* $1 \le j \le n$*, and using the initializations*

$$\begin{aligned} &\mathtt{sm}_{0,0} = 0, \\ &\mathtt{sm}_{i,0} = -\infty, \textit{ for all } 1 \le i \le m, \\ &\mathtt{sm}_{0,j} = -\infty, \textit{ for all } 1 \le j \le n, \\ &\mathtt{sg}_{0,0} = 0, \\ &\mathtt{sg}_{i,0} = -\alpha - \beta(i-1), \textit{ for all } 1 \le i \le m, \textit{ and} \\ &\mathtt{sg}_{0,j} = -\alpha - \beta(j-1), \textit{ for all } 1 \le j \le n. \end{aligned}$$

Proof The initialization works correctly, since in the match matrix the first row and the first column do not correspond to any valid alignment, and in the gap matrix the cases correspond to alignments starting with a gap.

The correctness of the computation of $\mathtt{sm}_{i,j}$ follows easily by induction, since take to next line $\max\{\mathtt{sm}_{i-1,j-1}, \mathtt{sg}_{i-1,j-1}\}$ is the maximum score of all alignments where a match can be appended. The correctness of the computation of $\mathtt{sg}_{i,j}$ can be seen as follows. An alignment ending in a gap is either (1) opening a new gap or (2) extending an existing one. In case (1) it is sufficient to take the maximum over the scores for aligning $A_{1..i-1}$ with $B_{1..j}$ and $A_{1..i}$ with $B_{1..j-1}$ so that the previous alignment ends with a match, and then add the cost for opening a gap. These maxima are given (by the induction assumption) by $\mathtt{sm}_{i-1,j}$ and $\mathtt{sm}_{i,j-1}$. In case (2) it is sufficient to take the maximum over the scores for aligning $A_{1..i-1}$ with $B_{1..j}$ and $A_{1..i}$ with $B_{1..j-1}$ so that the previous alignment ends with a gap, and then add the cost for extending a gap. These maxima are given (by the induction assumption) by $\mathtt{sg}_{i-1,j}$ and $\mathtt{sg}_{i,j-1}$. □

The running time is now reduced to $O(mn)$. Similar recurrences can be derived for local alignment and overlap alignment under affine gap score.

6.4.5 The invariant technique

It is instructive to study an alternative $O(mn)$ time algorithm for alignment under affine gap scores. This algorithm uses the *invariant technique*, in a similar manner to that in Section 6.2.1 for the longest common subsequence problem.

Consider the formula $\mathtt{sm}_{i',j'} - \alpha - \beta(j - j' + i - i' - 3) + s(a_i, b_j)$ inside the maximization in Equation (6.18). This can be equivalently written as $-\alpha - \beta(j + i - 3) + s(a_i, b_j) + \mathtt{sm}_{i',j'} + \beta(j' + i')$. The maximization goes over all valid values of i', j'

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

and the first part of the formula, namely $-\alpha - \beta(j+i-3) + s(a_i, b_j)$, is not affected. Recall the search tree $\mathcal{T}$ of Lemma 3.1 on page 20. With a symmetric change it can be transformed to support range maximum queries instead of range minimum queries; let $\mathcal{T}.\mathsf{RMaxQ}(c,d)$ return the *max* of values associated with keys $[c..d]$. Then, by storing values $\mathtt{sm}_{i,j} + \beta(j+i)$ together with key i to the search tree, we can query the maximum value and add the invariant. Using the reverse column-order as in Algorithm 6.3, we obtain Algorithm 6.5, which is almost identical to the sparse dynamic programming solution for the longest common subsequence problem; the main difference is that here we need to fill the whole matrix. We can thus obtain an $O(mn \log m)$ time algorithm for computing the global alignment under affine gap scores.

Algorithm 6.5: Affine gap score $S_G(A,B)$ using range maximum queries

Input: Sequences $A = a_1 a_2 \cdots a_m$ and $B = b_1 b_2 \cdots b_n$.
Output: Global alignment under affine gap scores $S_G(A,B)$.

```
1 Initialize the search tree T of Lemma 3.1 with keys 0, 1, ..., m;
2 T.update(0, 0);
3 sm_{0,0} ← 0;
4 for j ← 1 to n do
5     for i ← m down to 1 do
6         sm_{i,j} ← max{sm_{i-1,j-1} + s(a_i, b_j),
                         T.RMaxQ(0, i-1) − α − β(i+j-3) + s(a_i, b_j)};
7         T.update(i, sm_{i,j} + β(i+j));
8 sm_{m+1,n+1} ← max{sm_{m,n}, T.RMaxQ(0, m) − α − β(m+n-1)};
9 return sm_{m+1,n+1};
```

However, the use of a data structure in Algorithm 6.5 is in fact unnecessary. It is sufficient to maintain the maximum value on each row, and then maintain on each column the maximum of the row maxima; this is the same as the incremental computation of maxima for semi-infinite rectangles in a grid as visualized in Figure 6.5. The modified $O(mn)$ time algorithm is given as Algorithm 6.6.

6.5 Gene alignment

Consider the problem of having a protein sequence (produced by an unknown gene) and looking for this unknown gene producing it in the DNA. With prokaryotes, the task, which we call *prokaryote gene alignment*, is relatively easy: just modify the semi-local alignment (global alignment with the first row initialized to zero) so that $s_{i-1,j-1} + s(a_i, b_j)$ is replaced by $s_{i-1,j-3} + s(a_i, \mathtt{aa}[B[j-2..j]])$, where the first sequence A is the protein sequence, the second sequence B is the DNA sequence, and $\mathtt{aa}[xyz]$ is the amino acid encoded by codon xyz (recall Table 1.1). For symmetry, also $s_{i,j-1} - \delta$ should be replaced by $s_{i,j-3} - \delta$, and the initializations modified accordingly. An alternative and more rigorous approach using the DAG-path alignment of Section 6.6.5 is considered in Exercise 6.27.

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

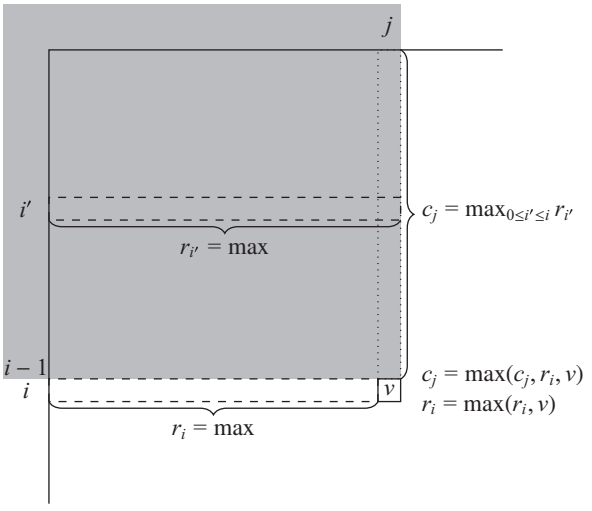


Figure 6.5 Maintaining sliding maxima in semi-infinite rectangles $[-\infty..i] \times [-\infty..j]$ over a grid of values. Here r_i is the maximum at row i computed with column-by-column evaluation order before reaching cell (i, j). After the updates visualized, c_j keeps the maximum over values in $[-\infty..i] \times [-\infty..j]$.

With eukaryotes, introns need to be taken into account. The most common way to do this is to use affine gap penalties and assign a large gap opening cost and a very small gap extension cost in the DNA side, and use a linear gap cost on the protein side. Call this approach *eukaryote gene alignment with affine gaps*. We leave it as an exercise for the reader to formulate an $O(mn)$ algorithm for solving this variant. However, this approach is clearly problematic because intron lengths can vary from tens to tens of thousands of nucleotides: there is no good choice for the gap extension cost.

Moreover, combining scores from two different phenomena into one objective function is like comparing apples and oranges. In what follows, we systematically avoid such formulations, and instead choose to optimize only one feature at a time, while limiting the others with thresholds.

For simplicity of exposition, we consider in what follows that the pattern $P = A$ is a cDNA sequence (the complement of an RNA sequence) instead of a protein sequence and denote the genomic DNA sequence $T = B$; the protein-to-DNA versions follow using the idea explained above.

Let us now develop a better approach for gene alignment in eukaryotes exploiting the fact that the number of introns in genes is usually quite small. For example, in the human genome, the average number of introns is 7.8, the maximum being 148. We can fix the maximum number of introns allowed in an alignment as follows.

Problem 6.16 Eukaryote gene alignment with a limited number of introns

For all $i \in [1..m]$ and $j' \in [1..j]$, let $e_{i,j,k}$ be the maximum-scoring semi-local alignment under linear gap scores of $P_{1..i}$ and $T_{j^s..j}$, using k *maximal free runs* of deletions in

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

$T_{j^s..j-1}$. Here by *free* we mean that each deletion in such a run incurs no cost to the overall score and by *maximal* that a free run cannot be succeeded by another free run. Compute the value $\max_{1\le j\le n, 0\le k\le \mathtt{maxin}} e_{m,j,k}$, where `maxin` is a given limit for the allowed number of introns.

Algorithm 6.6: Affine gap score $S_G(A,B)$ maintaining maxima of semi-infinite rectangles

Input: Sequences $A = a_1a_2\cdots a_m$ and $B = b_1b_2\cdots b_n$.
Output: Global alignment under affine gap score $S_G(A,B)$.

$\mathtt{sm}_{0,0} \leftarrow 0$;
for $i \leftarrow 1$ ***to*** m **do**
  $\mathtt{sm}_{i,0} \leftarrow -\infty$;
  $r_i \leftarrow -\infty$;
for $j \leftarrow 1$ ***to*** n **do**
  $\mathtt{sg}_{0,j} \leftarrow -\infty$;
  $c_i \leftarrow -\infty$;
for $j \leftarrow 1$ ***to*** n **do**
  for $i \leftarrow 1$ ***to*** m **do**
    $\mathtt{sm}_{i,j} \leftarrow \max\{\mathtt{sm}_{i-1,j-1} + s(a_i,b_j),$
      $\max(c_{j-1}, r_{i-1}) - \alpha - \beta(i+j-3) + s(a_i,b_j)\}$;
    $c_{j-1} \leftarrow \max(c_{j-1}, r_{i-1})$;
  for $i \leftarrow 1$ ***to*** m **do**
    $r_i \leftarrow \max(r_i, \mathtt{sm}_{i,j} + \beta(i+j))$;
for $i \leftarrow 1$ ***to*** m **do**
  $c_n \leftarrow \max(c_n, r_{i-1})$;
$\mathtt{sm}_{m+1,m+1} \leftarrow \max\{\mathtt{sm}_{m,n}, \max(c_n, r_m) - \alpha - \beta(m+n-1)\}$;
return $\mathtt{sm}_{m+1,n+1}$;

This problem can be easily solved by extending the semi-local alignment recurrences with the fact that each j can be the start of an exon, and hence position j can be preceded by any $j' < j-1$, where j' is the end of a previous exon. In such cases, one needs to spend one more free run of deletions in the genomic DNA to allow an intron. That is, one needs to look at alignment scores with $k-1$ runs of deletions, stored in $e_{*,*,k-1}$, when computing the best alignment with k runs of deletions. The solution is given below.

THEOREM 6.21 *The eukaryote gene alignment with a limited number of introns problem can be solved by computing all the scores $e_{i,j,k}$, using the recurrence*

$$e_{i,j,k} = \max\left\{ e_{i-1,j-1,k} + s(p_i,t_j),\ e_{i-1,j,k} - \delta,\ e_{i,j-1,k} - \delta,\right.$$
$$\left.\max_{j'<j-1}\{e_{i-1,j',k-1} + s(p_i,t_j),\ e_{i,j',k-1} - \delta\}\right\} \tag{6.20}$$

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

where $1 \le i \le m$, $1 \le j \le n$, *and using the initializations*

$$\begin{aligned} e_{0,0,0} &= 0, \\ e_{i,0,0} &= -\delta i, \textit{ for all } 1 \le i \le m, \\ e_{0,j,0} &= 0, \textit{ for all } 1 \le j \le n, \textit{ and} \\ e_{i,j,k} &= -\infty \textit{ otherwise.} \end{aligned}$$

The running time is $O(mn^2 \cdot \texttt{maxin})$, but this can be improved to $O(mn \cdot \texttt{maxin})$ by replacing each $\max_{j'<j-1} e_{x,j',k-1}$ with $m_{x,k-1}$ such that $m_{*,k}$ maintains the row maximum for each k during the computation of the values $e_{i,j,k}$. That is, $m_{i,k} = \max(m_{i,k}, e_{i,j,k})$. It is left as an exercise for the reader to find a correct evaluation order for the values $e_{i,j,k}$ and $m_{i,k}$ so that they get computed correctly.

Obviously, the values j giving the maximum for $\max_{1\le j\le n, 0\le k\le \texttt{maxin}} e_{m,j,k}$ are plausible ending positions for the alignment. The alignment(s) can be traced back with the standard routine.

6.6 Multiple alignment

Multiple alignment is the generalization of pair-wise alignment to more than two sequences. Before giving a formal definition, see below one possible alignment of three sequences:

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

What is no longer obvious is how to score the alignments and how to compute them efficiently.

6.6.1 Scoring schemes

A multiple alignment of d sequences $T^1, T^2, \ldots, T^d \in \Sigma^*$ is a $d \times n$ matrix M. We denote by M_{i*} the ith row in the alignment, and by M_{*j} the jth column in the alignment. The alignment is required that, for all $i \in [1..d]$, T^i must be a subsequence of M_{i*}, and the remaining $n - |T^i|$ characters of M_{i*} must be $-$, where $- \notin \Sigma$.

The *sum-of-pairs (SP) score* for M is defined as

$$\sum_{j=1}^{n} \sum_{\substack{i',i\in[1..d] \\ i'<i}} s(M[i',j], M[i,j]),$$

where $s(a,b)$ is the same score as that used for pair-wise alignments. The sum-of-pairs score can be identically written row-by-row as $\sum_{i',i\in[1..d],\, i'<i} W(M_{i'*}, M_{i*})$, using our earlier notion of pair-wise alignment scores.

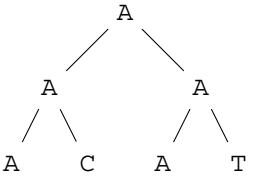
https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

Although this is a natural generalization from pair-wise alignments, this score has the drawback of not being well motivated from an evolutionary point of view: the same substitutions are possibly counted for many pairs of sequences, although they could in some cases be explained by just one substitution in the evolutionary tree of the biological sequences. An obvious improvement over the SP score for biological sequences is to find the most parsimonious evolutionary tree for the sequences to be aligned, and to count the sum of scores of substitutions and indels from that tree. This will be called the *parsimony score* of the multiple alignment.

To define this measure more formally, let us first assume that an evolutionary tree (V, E) with d leaves is given for us, where V denotes the set of nodes and E the set of edges. We will compute a score for every column M_{*j} of a multiple alignment. The final score of the alignment can be taken as the sum of the column scores.

Thus, for every column M_{*j} of a multiple alignment, we label each leaf i of the tree with $\ell(i) = M[i,j]$. The *small parsimony* problem is to label also the internal nodes $v \in V$ with some $\ell(v) \in \Sigma \cup \{-\}$ such that the sum of the scores $s(\ell(v), \ell(w))$ over all edges $(v, w) \in E$ is maximized.

For example, a solution to the small parsimony problem for a column `ACAT` of a multiple alignment could be the following:



With this example, the sum of scores would be $4s(\mathtt{A}, \mathtt{A}) + s(\mathtt{A}, \mathtt{C}) + s(\mathtt{A}, \mathtt{T})$.

To find this optimal assignment for a fixed column M_{*j}, a simple dynamic programming algorithm works. Let $s_v(a)$ denote the optimal score of the subtree rooted at v, when v is labeled with the character $a \in \Sigma \cup \{-\}$. It is easy to see that the following recurrence holds:

$$s_v(a) = \sum_{w\,:\,(v,w)\in E} \left(\max_{b \in \Sigma \cup \{-\}} s_w(b) + s(a,b) \right). \tag{6.21}$$

One can compute $s_v(a)$, for all $v \in V$ and $a \in \Sigma \cup \{-\}$, in a bottom-up manner, from leaves to the root. The optimal score is $\max_{a \in \Sigma \cup \{-\}} s_{\mathtt{root}}(a)$. The optimal labeling can then be computed top-down from the root to the leaves, by selecting the labels leading to the maximum-score solution.

Recall our original goal of defining a parsimony score for a given multiple alignment. Assuming that an evolutionary tree is given for us, we can now compute the score in optimal time on constant alphabets. Without this assumption, we should enumerate through all possible trees, and choose the one that gives the maximum score summed over all columns of the multiple alignment. There are $\Theta(4^{2d}/d^{3/2})$ binary evolutionary trees with d leaves, so the computation of the parsimony score of a given alignment is exponential in d.

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

6.6.2 Dynamic programming

Having described different scoring schemes for a multiple alignment, let us now see how to actually compute an optimal multiple alignment, for such scoring schemes.

Let $s_{i_1,i_2,\ldots,i_d}$ denote the optimal scoring alignment of $T^1_{1..i_1}, T^2_{1..i_2}, \ldots, T^d_{1..i_d}$, under some fixed score. To compute $s_{i_1,i_2,\ldots,i_d}$, one needs to consider all the $2^d - 1$ ways the last column of the alignment can look (any combinations of j sequences that can have a gap, for all $0 \leq j < d$). For example, in the three-dimensional case we have

$$s_{i_1,i_2,i_3} = \max \begin{cases} s_{i_1-1,i_2,i_3} + s(t^1_{i_1}, -, -) \\ s_{i_1,i_2-1,i_3} + s(-, t^2_{i_2}, -) \\ s_{i_1,i_2,i_3-1} + s(-, -, t^3_{i_3}) \\ s_{i_1-1,i_2-1,i_3} + s(t^1_{i_1}, t^2_{i_2}, -) \\ s_{i_1-1,i_2,i_3-1} + s(t^1_{i_1}, -, t^3_{i_3}) \\ s_{i_1,i_2-1,i_3-1} + s(-, t^2_{i_2}, t^3_{i_3}) \\ s_{i_1-1,i_2-1,i_3-1} + s(t^1_{i_1}, t^2_{i_2}, t^3_{i_3}) \end{cases} \tag{6.22}$$

where $s(x, y, z)$ denotes the score for a column. The running time of the d-dimensional case is $O(n^d(d^2 + 2^d))$ in the case of an SP score, $O(n^d(d + 2^d))$ in the case of a parsimony score with a fixed evolutionary tree, and $O(n^d(d + 2^d)4^{2d})$ in the case of a general parsimony score (enumerating over all trees).

6.6.3 Hardness

We will now show that it is unlikely that one will be able to find much faster algorithms (namely, polynomial) for multiple alignment. For such a proof it suffices to consider the simplest scoring scheme.

Problem 6.17 Longest common subsequence (LCS) on multiple sequences

Find a longest sequence C that is a subsequence of all the given sequences $T^1, T^2, \ldots, T^d$.

This problem is identical to finding a maximum-scoring multiple alignment with the scoring scheme $s(t^1_{i_1}, t^2_{i_2}, \ldots, t^d_{i_d}) = 1$ if and only if $t^1_{i_1} = t^2_{i_2} = \cdots = t^d_{i_d}$, and 0, otherwise.

THEOREM 6.22 *The longest common subsequence problem on multiple sequences is NP-hard.*

Proof We reduce from the independent set problem by showing that, given a graph $G = (V = \{1, \ldots, n\}, E)$, we can construct n strings $S^1, \ldots, S^n$, over the alphabet V, such that G has an independent set of size k if and only if $S^1, \ldots, S^n$ have a common subsequence of length k.

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

For each vertex i, let $j_1 < j_2 < \cdots < j_{t(i)}$ denote all the vertices of G which are not adjacent to i and are smaller than i. Then the ith string is

$$S^i = j_1 \cdot j_2 \cdot \cdots \cdot j_{t(i)} \cdot i \cdot 1 \cdot 2 \cdots (i-1) \cdot \cdots (i+1) \cdot (i+2) \cdot \cdots \cdot n,$$

where, in particular, $S^1 = 1 \cdot 2 \cdot \cdots \cdot n$.

If $\{i_1, \ldots, i_k\}$, with $i_1 < \cdots < i_k$, is an independent set of size k of G, then $i_1, \ldots, i_k$ is a common subsequence of $S^1, \ldots, S^k$. Indeed, if $j \notin \{i_1, \ldots, i_k\}$, by construction $i_1, \ldots, i_k$ appear in this order in S^j. If $j \in \{i_1, \ldots, i_k\}$ then the elements of $\{i_1, \ldots, i_k\} \setminus \{j\}$ smaller than j have been added, by construction, also to the left of j in S^j.

Conversely, assuming that $i_1, \ldots, i_k$ is a common subsequence of $S^1, \ldots, S^n$, it is also a subsequence of S_1, and thus $i_1 < \cdots < i_k$. We argue that $\{i_1, \ldots, i_k\}$ is an independent set of size k of G. Assume for a contradiction that there is an edge between $i_\ell, i_m \in \{i_1, \ldots, i_k\}$, where, say, $i_\ell < i_m$. Then in S^m, by construction, i_ℓ appears only to the right of i_m, which contradicts the fact that $i_1, \ldots, i_k$ is a subsequence also of S^m. □

6.6.4 Progressive multiple alignment

Owing to the unfeasibility of multiple alignment with many sequences, most bioinformatics tools for multiple alignments resort to ad-hoc processes, like building the multiple alignment incrementally: first do pair-wise alignments and then combine these into larger and larger multiple alignments. The process of combining alignments can be guided by an evolutionary tree for the sequences.

Assume we have constructed such a *guide tree*. We can construct a multiple alignment bottom-up by doing first pair-wise alignment combining neighboring leaves and then pair-wise alignments of two multiple alignments (of size $1, 2, \ldots$) until we reach the root. The only difficulty is how to do pair-wise alignment of two multiple alignments (called *profiles* in this context). The trick is to simply consider each profile as a sequence of columns and to apply normal pair-wise alignment when defining $s(M_{*j}, M'_{*j})$, for example as an SP score, where M and M' are the two profiles in question. This ad-hoc process is called the *once a gap, always a gap* principle, since the gaps already in M and M' are not affected by the alignment; new gaps are formed one complete column at a time. Example 6.18 clarifies the process.

Example 6.18 Construct a multiple alignment of sequences $A =$ `AAGACAC`, $B =$ `AGAACC`, $C =$ `AGACAC`, and $D =$ `AGAGCAC` using progressive alignment.

Solution

Assume the guide tree has an internal node v with A and B as children, an internal node w with C and D as children, and a root with v and w as children. To form the profile for node v, we need to compute the optimal pair-wise alignment of A and B, which could be

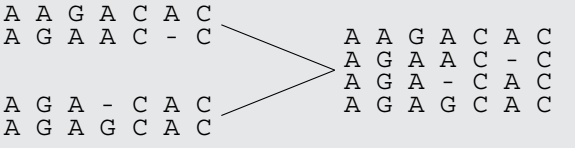
```
A A G A C A C
A G A A C - C
```

with some proper choice of scores. Likewise, the profile for w could be

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

```
A G A - C A C
A G A G C A C
```

Finally, for the root, we take the profiles of v and w as sequences of columns, and obtain the following multiple alignment as the result:



6.6.5 DAG alignment

The alignment of profiles raises the question of whether there is some approach to avoid adding a full column of gaps at a time. We will next explore an approach based on labeled DAGs (recall Chapter 4) that tackles this problem.

A multiple alignment M can be converted to a labeled DAG as follows. On each column j, create a vertex for each character c appearing in it. Let such a vertex be $v_{j,c}$. Add an arc from $v_{j',c'}$ to $v_{j,c}$ if there is an i such that $M[i,j'] = c'$, $M[i,j] = c$, and $M[i,j''] = -$ for all j'' such that $j' < j'' < j$. Add a global source by connecting it to all previous sources of the DAG. Likewise, add a global sink and arcs from all previous sinks to it. Example 6.20 on page 106 illustrates the construction (but not showing the added global source and global sink).

An interesting question is as follows: given two labeled DAGs as above, how can one align them? The alignment of DAGs could be defined in many ways, but we resort to the following definition that gives a feasible problem to be solved.

Problem 6.19 DAG-path alignment

Given two labeled DAGs, $A = (V^A, E^A)$ and $B = (V^B, E^B)$, both with a unique source and sink, find a source-to-sink path P^A in A and a source-to-sink path P^B in B, such that the global alignment score $S(\ell(P^A), \ell(P^B))$ is maximum over all such pairs of paths, where $\ell(p)$ denotes the concatenation of the labels of the vertices of P.

Surprisingly, this problem is easy to solve by extending the global alignment dynamic programming formulation. Consider the recurrence

$$s_{i,j} = \max \left\{ \max_{i' \in N_A^-(i), j' \in N_B^-(j)} s_{i',j'} + s(\ell^A(i), \ell^B(j)),\ \max_{i' \in N_A^-(i)} s_{i',j} - \delta,\ \max_{j' \in N_B^-(j)} s_{i,j'} - \delta \right\}, \tag{6.23}$$

where i and j denote the vertices of A and B with their topological ordering number, respectively, $N_A^-(\cdot)$ and $N_B^-(\cdot)$ are the functions giving the set of in-neighbors of a given

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

vertex, and $\ell^A(\cdot)$ and $\ell_B(\cdot)$ are the functions giving the corresponding vertex labels. By initializing $s_{0,0} = 0$ and evaluating the values in a suitable evaluation order (for example, $i = 1$ to $|V^A|$ and $j = 1$ to $|V^B|$), one can see that $s_{|V^A|,|V^B|}$ evaluates to the solution of the DAG-path alignment of Problem 6.19: an optimal alignment can be extracted with a traceback.

THEOREM 6.23 *Given two labeled DAGs $A = (V^A, E^A)$ and $B = (V^B, E^B)$, the DAG-path alignment problem (Problem 6.19) can be solved in $O(|E^A||E^B|)$ time.*

Proof The correctness can be seen by induction. Assume $s_{i',j'}$ gives the optimal alignment score between any two paths from the source of A to i' and from the source of B to j', and this value is computed correctly for all $i' \in N^-(i)$ and $j' \in N^-(j)$ for some fixed (i,j). Any alignment of a pair of paths to i and j ends by $\ell^A(i) \to \ell^B(j)$, by $\ell^A(i) \to -$, or by $- \to \ell^B(j)$. Equation (6.23) chooses among all the possible shorter alignments the one that gives the maximum score for $s_{i,j}$.

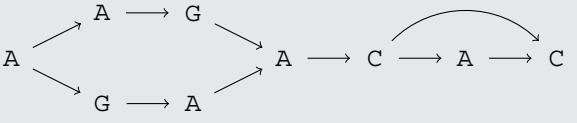
Each pair of arcs of A and B is involved in exactly one cross-product in Equation (6.23), which gives the claimed running time. □

The idea of using the DAG alignment in progressive multiple alignment is to convert the pair-wise alignments at the bottom level to DAGs, align the DAGs at each internal node bottom-up so that one always extracts the optimal paths into a pair-wise alignment, and converts it back to a DAG. One can modify the substitution scores to take into account how many bottom-level sequences are represented by the corresponding vertex in the DAG. This ad-hoc process is illustrated in Example 6.20.

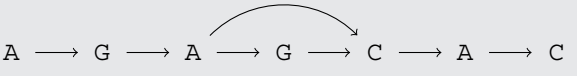
Example 6.20 Construct a multiple alignment of the sequences $A =$ AAGACAC, $B =$ AGAACC, $C =$ AGACAC, and $D =$ AGAGCAC using a DAG-based progressive alignment.

Solution

Assume the same guide tree as in Example 6.18. That pair-wise alignment of A and B can be converted to the following DAG:



Likewise, that pair-wise alignment of C and D can be converted to the following DAG:



For the root, we find the pair of paths from the two DAGs that have the best alignment. These paths form the following alignment:

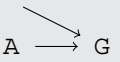
https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

```
A A G A - C A C
A - G A G C A C
```

We project the final aligment to the four input sequences using the columns of the root profile as anchors, and partially re-align the areas between consecutive projected anchors. This can be done recursively using the same approach. We obtain the following multiple alignment, where * indicates the anchors projected to all four sequences and % indicates the anchors projected to some of the sequences:

```
* % %   *   * % *
A A G - A - C A C
A - G A A - C - C
A - G - A - C A C
A - G - A G C A C
```

The re-alignment between the two first * anchors was made between the corresponding part of the root DAG and the corresponding part of all sequences causing the partial anchors %. In this case, sequence B alone caused the partial anchors %. That is, we aligned GA with the DAG:



Similarly, sequence B causes the last partial anchor %, and there the only possible alignment is adding a gap in B.

6.6.6 Jumping alignment

Consider a database of multiple alignments. For example, protein sequences are often clustered into *protein families* sharing some common functionality. Each such family can be represented as a multiple alignment. With a new protein whose function is unclear, one can try to align the new sequence to all families in the database, and link that sequence to the family with which it best aligns.

Jumping alignment can be used to model this alignment problem. The idea is to limit the amount of times the sequence to be aligned can change row inside the multiple alignment (profile). This can be accomplished for example by assigning a large penalty for changing row, or by fixing some upper bound on the number of allowed jumps.

Let us consider the case of a limited number of jumps. Let $M[1..d, 1..n]$ denote the profile and $A = a_1 \cdots a_m$ the sequence to be aligned to M. Let $s^l_{i,j,k}$ denote the score of the best global alignment of $a_1 \cdots a_i$ against $M[k_0, 1..j_0]M[k_1, j_0 + 1..j_1] \cdots M[k_l, j_l + 1..j]$ with $k_l = k$. The recurrence for computing the values $s^l_{i,j,k}$ is

$$s^l_{i,j,k} = \max_{k'} \Big(s^l_{i-1,j-1,k} + s(a_i, M[k,j]), s^l_{i,j-1,k} - \delta, s^l_{i-1,j,k} - \delta, \\ s^{l-1}_{i-1,j-1,k'} + s(a_i, M[k',j]), s^{l-1}_{i-1,j,k'} - \delta, s^{l-1}_{i,j-1,k'} - \delta \Big).$$

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

With a proper initialization and a suitable evaluation order, values $s^l_{i,j,k}$ can be computed in time $O(mndL)$, where L is the limit for the number of jumps.

This formulation has the shortcoming that existing gaps in M are aligned to A. Exercise 6.27 asks for a better formulation, whose idea is presented in Example 6.21.

Example 6.21 Find an optimal alignment of $A =$ AAGAAGCC to the profile

$$M = \begin{array}{c} \texttt{A A G - A - C A C} \\ \texttt{A - G A A - C - C} \\ \texttt{A - G - A - C A C} \\ \texttt{A - G - A G C A C} \end{array}$$

with at most $L = 3$ jumps.

Solution

Using the better formulation asked in Exercise 6.27, there is a solution with exact match, visualized below with lower-case letters. In this formulation, the gaps in M can be bypassed for free:

```
a a g - A - C A C
A - G a a - c - c
A - G - A - C A C
A - G - A g C A C
```

6.7 Literature

Edit distance computation (Levenshtein 1966; Wagner & Fischer 1974) is a widely studied stringology problem that has been explored thoroughly in many textbooks. We covered here one general speedup technique, namely the shortest detour (Ukkonen 1985), and one more specific technique, namely the bitparallel algorithm tailored for the Levenshtein edit distance and approximate string matching (Exercise 6.4) (Myers 1999). For the latter, we followed essentially the version described by Hyyrö (2001). Interestingly, two other kinds of technique also lead to sub-quadratic running times (Masek & Paterson 1980; Crochemore *et al.* 2002): we develop some ideas related to the former in Exercise 6.10. Space-efficient traceback (see the exercises) is from Hirschberg (1975). The approximate string matching algorithms under Levenshtein and Hamming distances can be sped up to $O(kn)$ time (Landau & Vishkin 1988) using the index structures we study in Chapter 8. Many practical techniques for obtaining efficient exact and approximate string matching algorithms are covered in Navarro & Raffinot (2002).

The first sparse dynamic programming algorithm for edit distances is the Hunt–Szymanski algorithm for the LCS problem (Hunt & Szymanski 1977). We explained another algorithm in order to demonstrate the invariant technique and the connection to range search data structures (Mäkinen *et al.* 2003). Observe that, using a more efficient

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

range search structure for semi-infinite ranges as in Exercise 3.6, the $O(|M|\log m)$ time algorithm can be improved to $O(|M|\log\log m)$.

There also exist sparse dynamic programming algorithms for other variants of edit distances (Eppstein *et al.* 1992a,b).

The global and local alignment algorithms are from Needleman & Wunsch (1970) and Smith & Waterman (1981), respectively. For affine gap scores, the first algorithm is from Gotoh (1982) and the second one (invariant technique, range maximum) was developed for this book. The distribution on the number of introns is taken from Sakharkar *et al.* (2004). The derivation of substitution scores in Insight 6.1 follows the presentation of Durbin *et al.* (1998).

The small parsimony problem we studied in the context of multiple alignment scores is from Sankoff (1975). The NP-hardness of the longest common subsequence problem on multiple sequences was first examined in Maier (1978); that reduction (from vertex cover) is stronger and more complex than the one we described here (from an independent set), insofar as it shows that the problem is NP-hard even on binary sequences. The basic version (on a large alphabet) of that reduction is, however, not much more difficult than ours (see Exercise 6.24).

The literature on multiple alignments is vast; we refer the reader to another textbook (Durbin *et al.* 1998) for further references. The idea of using DAGs to replace profiles appears in Lee *et al.* (2002) and in Löytynoja *et al.* (2012); we loosely followed the latter.

Jumping alignments are from Spang *et al.* (2002); we followed the same idea here, but gave a quite different solution.

Exercises

6.1 Show that the edit distance is a *metric*, namely that it satisfies

i. $D(A,B) \geq 0$,
ii. $D(A,B) = 0$, if and only if $A = B$,
iii. $D(A,B) = D(B,A)$,
iv. $D(A,B) \leq D(A,C) + D(C,B)$.

6.2 Modify Algorithm 6.1 on page 74 to use only one vector of length $m+1$ in the computation.

6.3 The shortest-detour algorithm is able to compute the dynamic programming matrix only partially. However, just allocating space for the rectangular matrix of size $m \times n$ requires $O(mn)$ time. Explain how one can allocate just the cells that are to be computed. Can you do the evaluation in $O(d)$ space, where $d = D(A,B)$?

6.4 Modify Myers' bitparallel algorithm to solve the k-errors problem on short patterns.

6.5 Optimize Myers' bitparallel algorithm to use fewer basic operations.

6.6 Write a program to automatically verify each row in the truth tables in the lemmas proving the correctness of Myers' bitparallel algorithm.

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

6.7 Write a program to automatically discover the four "easy" operations (those of four the first lemmas) in Myers' bitparallel algorithm.

6.8 Describe how to trace back an optimal alignment with Myers' bitparallel algorithm.

6.9 Consider Exercise 6.16. Show that the same technique can be combined with Myers' bitparallel algorithm to obtain linear space traceback together with the bitparallel speedup. *Hint.* Compute the algorithm also for the reverse of the strings. Then you can deduce $j_{\texttt{mid}}$ from forward and reverse computations. See Section 7.4 for an analogous computation.

6.10 The four Russians technique we studied in Section 3.2 leads to an alternative way to speed up edit distance computation. The idea is to encode each column as in Myers' bitparallel algorithm with two bitvectors R_j^+ and R_j^- and partition these bitvectors to blocks of length b. Each such pair of bitvector blocks representing an *output encoding* of a block $d_{i-b+1..i,j}$ in the original dynamic programming matrix is uniquely defined by the *input parameters* $A_{i-b+1..i}$, b_j, $d_{i-b,j-1}$, and $d_{i-b,j}$ and the bitvector blocks representing the block $d_{i-b+1..i,j-1}$. Develop the details and choose b so that one can afford to precompute a table storing, for all possible input parameters, the output encoding of a block. Show that this leads to an $O(mn/\log_\sigma n)$ time algorithm for computing the Levenshtein edit distance.

6.11 Prove Theorem 6.10.

6.12 Show that the set $M = M(A,B) = \{(i,j) \mid a_i = b_j\}$ which is needed for sparse dynamic programming LCS computation, sorted in reverse column-order, can be constructed in $O(\sigma + |A| + |B| + |M|)$ time on a constant alphabet and in $O((|A| + |B|)\log|A| + |M|)$ time on an ordered alphabet. Observe also that this construction can be run in parallel with Algorithm 6.3 to improve the space requirement of Theorem 6.12 to $O(m)$.

6.13 The sparse dynamic algorithm for distance $D_{\texttt{id}}(A,B)$ can be simplified significantly if derived directly for computing $|\mathsf{LCS}(A,B)|$. Derive this algorithm. *Hint.* The search tree can be modified to answer range maximum queries instead of range minimum queries (see Section 6.4.5).

6.14 Give a pseudocode for local alignment using space $O(m)$.

6.15 Give a pseudocode for tracing an optimal path for the maximum-scoring local alignment, using space quadratic in the alignment length.

6.16 Develop an algorithm for tracing an optimal path for the maximum-scoring local alignment, using space linear in the alignment length. *Hint.* Let $[i'..i] \times [j'..j]$ define the rectangle containing a local alignment. Assume you know $j_{\texttt{mid}}$ for row $(i - i')/2$ where the optimal alignment goes through. Then you can independently recursively consider rectangles defined by $[i'..(i - i')/2] \times [j'..j_{\texttt{mid}}]$ and $[(i - i')/2] \times [j_{\texttt{mid}}..j]$. To find $j_{\texttt{mid}}$ you may, for example, store, along with the optimum value, the index j where that path crosses the middle row.

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

6.17 Prove the correctness of Algorithms 6.5 and 6.6 on pages 98 and 100.

6.18 The sequencing technology SOLiD™ from Applied Biosystems produces short reads of DNA in *color-space* with a two-base encoding defined by the following matrix (row = first base, column = second base):

	A	C	G	T
A	0	1	2	3
C	1	0	3	2
G	2	3	0	1
T	3	2	1	0

For example, T012023211202102 equals TTGAAGCTGTCCTGGA (the first base is always given). Modify the overlap alignment algorithm to work properly in the case where one of the sequences is a SOLiD read and the other is a normal sequence.

6.19 Show that the relative entropy used in Equation (6.15) on page 91 is non-negative.

6.20 Give a pseudocode for prokaryote gene alignment.

6.21 Give a pseudocode for eukaryote gene alignment with affine gaps.

6.22 Give a pseudocode for the $O(mn \cdot \texttt{maxin})$ time algorithm for eukaryote gene alignment with limited introns.

6.23 Modify the recurrence for gene alignment with limited introns so that the alignment must start with a start codon, end with a stop codon, and contain GT and AG dinucleotides at intron boundaries. Can you still solve the problem in $O(mn \cdot \texttt{maxin})$ time? Since there are rare exceptions when dinucleotides at intron boundaries are something else, how can you make the requirement softer?

6.24 Give an alternative proof for the NP-hardness of the longest common subsequence problem on multiple sequences, by using a reduction from the vertex cover problem defined on page 16. *Hint.* From a graph $G = (V = \{1, \ldots, |V|\}, E)$ and integer k, construct $|E| + 1$ sequences having LCS of length $|V| - k$ if and only if there is vertex cover of size k in G. Recall that vertex cover V' is a subset of V such that all edges in E are incident to at least one vertex in V.

6.25 Develop a sparse dynamic programming solution for computing the longest common subsequence of multiple sequences. What is the expected size of the match set on random sequences from an alphabet of size σ? *Hint.* The search tree can be extended to higher-dimensional range queries using recursion (see the range trees from the computational geometry literature).

6.26 Reformulate the DAG-path alignment problem (Problem 6.19 on page 105) as a local alignment problem. Show that the Smith–Waterman approach extends to this, but is there a polynomial algorithm if one wishes to report alignments with negative score? Can the affine gap cost alignment be extended to the DAG-path alignment? Can one obtain a polynomial-time algorithm for that case?

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press

6.27 Show how to use DAG-path alignment to align a protein sequence to DNA. *Hint*. Represent the protein sequence as a *codon-DAG*, replacing each amino acid by a sub-DAG representing its codons.

6.28 Consider a DAG with predicted exons as vertices and arcs formed by pairs of exons predicted to appear inside a transcript: this is the *splicing graph* considered in Chapter 15. Detail how DAG-path alignment on the splicing graph and the codon-DAG of the previous assignment can be used to solve the eukaryote gene alignment problem.

6.29 Reformulate jumping alignment so that A is just aligned to the sequences forming the multiple alignment and not to the existing gaps inside the multiple alignment, as in Example 6.21. Show that the reformulated problem can be solved as fast as the original one.

https://doi.org/10.1017/CBO9781139940023.008 Published online by Cambridge University Press