Chapter 1

Sequence Alignment

See also Chapter 3 of [12], Sequence Comparison and Database Search.

1.1 Introduction

The similarity (or rather dissimilarity) between two strings can be measured in the number of operations needed to transform one into the other. There are three basic operations we consider here: changing one character into another, inserting a character, or deleting a character. In the context of molecular biology these operations correspond to mutations (point mutations or insertions and deletions) in the genome.

When we assume that no two operations take place at the same position (like changing a character, then removing it) the operations used to transfer one string into another can be represented by an *alignment* of the two strings. Corresponding symbols are written in columns, marking positions where a symbol was deleted or inserted with a dash in the proper position.

1.1 Example. We have recreated an example of alignment given at wikipedia. It consists of sequences AAB24882 and AAB24881, and was generated using the ClustalW2 tool at the European Bioinformatics Institute, where all settings were left as default. The symbol * in the bottom row indicates that the two sequences are equal at that position, whereas : and . indicate decreasing similarity of the amino acids at that position.

```
AAB24882 TYHMCQFHCRYVNNHSGEKLYECNERSKAFSCPSHLQCHKRRQIGEKTHEHNQCGKAFPT 60
AAB24881 --------------------YECNQCGKAFAQHSSLKCHYRTHIGEKPYECNQCGKAFSK 40
                            ****: .***:  * *:** * :****.:* *******..

AAB24882 PSHLQYHERTHTGEKPYECHQCGQAFKKCSLLQRHKRTHTGEKPYE-CNQCGKAFAQ- 116
AAB24881 HSHLQCHKRTHTGEKPYECNQCGKAFSQHGLLQRHKRTHTGEKPYMNVINMVKPLHNS 98
          **** *:***********:***:**.: .***************    :  *.: :
```

Formally an *alignment* of strings x and y over alphabet Σ is a sequence of letter vectors, $\binom{x_1}{y_1}\binom{x_2}{y_2}\dots\binom{x_\ell}{y_\ell}$, with $x_i, y_i \in \Sigma \cup \{\varepsilon\}$ and $\binom{x_i}{y_i} \neq \binom{\varepsilon}{\varepsilon}$ such that $x = x_1x_2\dots x_\ell$ and $y = y_1y_2\dots y_\ell$. Note that $\ell \leq |x| + |y|$. Usually the empty string ε is represented by a dash $-$.

Given two sequences x and y it is an algorithmic task to determine the alignment where the number of operations involved has been minimal, counting the number of positions in the alignment where the two rows have unequal content. This value is called the *edit distance* of x and y.

In general one considers a *weighted* version of this problem by adding a *scoring system*. This in general consists of a similarity matrix σ (or substitution matrix) specifying a value $\sigma(a, b)$ for all a, b in the alphabet (representing the cost of changing a into b) and gap-penalty $\sigma(a, -)$ and $\sigma(-, b)$ for deleting a or inserting b. Thus the score for the general alignment above is given by $\sum_{1=1}^{\ell} \sigma(x_i, y_i)$, where the empty string ε is equated with the dash $-$.

Given a scoring system, the *similarity* of strings x and y is defined to be the maximal score taken over all alignments of x and y. An alignment that has this score is called an *optimal* alignment of x and y.

In simple examples the distances are given by just three values, one fixed value (typically positive) for matches $\sigma(a, a)$, one (typically negative) for mismatches $\sigma(a, b)$, $a \neq b$, and one (also negative) for the 'insdels' (insertions and deletions) $\sigma(a, -)$ and $\sigma(-, b)$. This latter is sometimes referred to as the *gap penalty*.

1.2 Example. The scoring system on the alphabet $\{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$ of nucleotides is defined here by the values $+2$ and -1 for match and mismatch, and -1 for gaps.

For the strings TCAGACGATTG and TCGGAGCTG a possible alignment is

TCAG - ACG - ATTG
TC - GGA - GC - T - G

It consists 7 matches and 6 insdels, so its score is $14 - 6 = 8$.

Similarly, the alignment

TCAGACGATTG
TCGGA - GCT - G

consists of 6 matches, 2 mismatches, and 2 insdels. Consequently the score is $12 - 2 - 2 = 8$. Both alignments have the same score, and the similarity of the strings is at least 8.

As stated above, one usually has $\sigma(a, a) > 0$. In some applications also $\sigma(a, b)$ may be positive when a and b are different (but have some similarity). Section 1.7 for the BLOSUM62 scoring system, used for amino acids, which has this feature. In general σ will be symmetric: $\sigma(a, b) = \sigma(b, a)$. In the sequel

the value for insdels is assumed to be given by a fixed gap penalty $g \leq 0$, which does not depend on the symbol that is deleted or introduced.

As noted above, an alignment gives at most a single operation at each position, which seems reasonable in general. Consider the case where deleting A and C costs -5 and -2, respectively, whereas substituting A by C costs -2. Now the two operations $A \to C \to -$ have total cost -4 which is better than the direct deletion of A. In such a case the algorithms of this section will compute the optimal alignment, however this might not correspond to the optimal score set of operations from one string to the other. Usually the scoring system avoids this kind of problems.

1.2 Global Alignment

Given a pair of strings x and y over an alphabet Σ and a scoring system for Σ, we want to compute the similarity of x and y, and an optimal alignment for the strings.

We use a dynamic programming approach for this problem. The algorithm computes the similarity for each pair of prefixes of the two strings starting with short prefixes, storing the values in a table, and reusing them for the longer prefixes. When the scores of the partial alignments are determined, the second phase starts. The alignment itself is computed from the numbers stored, working backwards. This is called a *traceback*. In the context of molecular biology this method is known as the *Needleman-Wunsch* algorithm [8].

A recursive implementation of the problem is easily given. Consider the last position of an optimal alignment of strings xa and yb. We have only three possibilities: $\binom{-}{b}$, $\binom{a}{b}$, or $\binom{a}{-}$. Hence the similarity of x and y, the value $\mathrm{sim}(xa, yb)$ of an optimal alignment is found by recursively computing

$$\mathrm{sim}(xa, yb) = \max \begin{cases} \mathrm{sim}(xa, y) & + & g \\ \mathrm{sim}(x, y) & + & \sigma(a, b) \\ \mathrm{sim}(x, yb) & + & g \end{cases}$$

Boundary values (when one of the sequences is empty) can be obtained from the identities $\mathrm{sim}(xa, \varepsilon) = \mathrm{sim}(x, \varepsilon) + g$, and $\mathrm{sim}(\varepsilon, \varepsilon) = 0$.

Let $x = x_1 \ldots x_m$ and $y = y_1 \ldots y_n$ be two strings that we want to align. Denote the value of the optimal alignment of the prefixes $x_1 \ldots x_i$ and $y_1 \ldots y_j$ by $A[i, j]$, to stay close to programming style. (In our program strings start at position 1, index $i = 0$ or $j = 0$ corresponds to the empty string.)

The first phase of the algorithm computes the values $A[i, j]$ as follows. The value of the optimal alignment, the similarity of x and y, can be found as $A[m, n]$.

$$\begin{array}{rcll} A[i,0] & = & i \cdot g & 0 \leq i \leq m \\ A[0,j] & = & j \cdot g & 0 \leq j \leq n \\ A[i,j] & = & \max \begin{cases} A[i, j-1] & + & g \\ A[i-1, j-1] & + & \sigma(x_i, y_j) \\ A[i-1, j] & + & g \end{cases} & 1 \leq i \leq m, 1 \leq j \leq n \end{array}$$

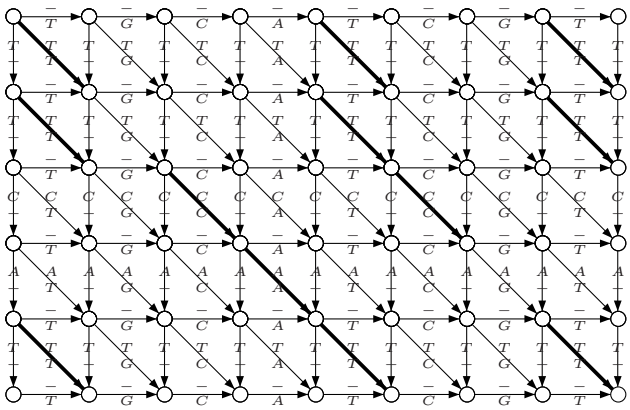
For the second phase, traceback, we assume that for each position (i, j) in the matrix A the cases were stored for which the value of that element was obtained, either $(i, j-1)$, $(i-1, j-1)$, or $(i-1, j)$ – as the three cases in the maximum, representing $\binom{-}{y_j}$, $\binom{x_i}{y_j}$, or $\binom{x_i}{-}$. Now start at the bottom-right position (m, n), and return to the cell of the matrix that resulted in that value. This is repeated until the first cell $(0, 0)$ is reached. In many cases the maximum was obtained not for one of the arguments, but for two or even three arguments. In that case we can choose to store just a single of these, or to store all of them, and trace all alignments rather than single one.

Alternatively, the trace is not followed from stored values, but is recomputed from the values in the matrix at the current position $A[i, j]$ and three neighbouring positions $A[i-1, j]$, $A[i-1, j-1]$, and $A[i, j-1]$.

Complexity. We assume the given strings have length m and n, respectively. The matrix takes space $\mathcal{O}(mn)$, and computing all its elements takes time $\mathcal{O}(mn)$. The traceback is computed in time $\mathcal{O}(m+n)$. If we do not need the alignment itself, but only its score, it is not necessary to store all elements of the matrix but only the last column (or last row). This reduces the space complexity to $\mathcal{O}(m)$, but time complexity still is $\mathcal{O}(mn)$.

1.3 Example. Global alignment of TTCAT and TGCATCGT with scoring system match 5, mismatch -2, and insdel -6.

A graph represention the problem is as below. The task is to find the optimal path from top-left to bottom-right, where the costs of traversing an edge are related to the label of the edge (negative values represent costs, while positive values can be seen as rewards). Bold diagonal edges represent matches, horizontal and vertical edges represent insdels.

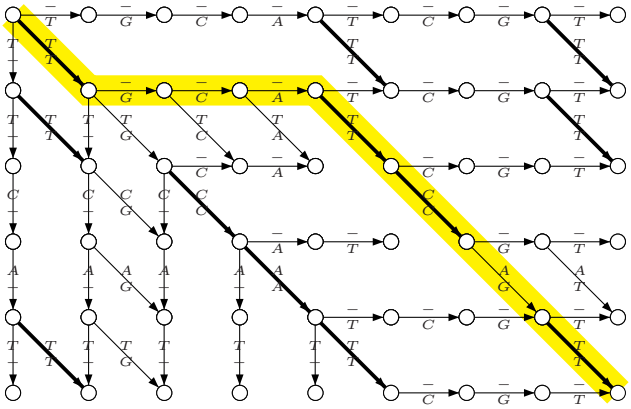


The following matrix is computed by the dynamic programming algorithm. It indicates that the score of the optimal global alignment equals 0.

	–	T	G	C	A	T	C	G	T
–	0	−6	−12	−18	−24	−30	−36	−42	−48
T	−6	5	−1	−7	−13	−19	−25	−31	−37
T	−12	−1	3	−3	−9	−8	−14	−20	−26
C	−18	−7	−3	8	2	−4	−3	−9	−15
A	−24	−13	−9	2	13	7	1	−5	−11
T	−30	−19	−15	−4	7	18	12	6	0

The alignment itself can be traced back from the final position, following incoming edges that represent the direction over which the maximal score was obtained. These edges and a possible traceback are as follows, giving the alignment

T	-	-	-	T	C	A	T
T	G	C	A	T	C	G	T
+5	-6	-6	-6	+5	+5	-2	+5



Two other alignments with optimal score can be read from the diagram.

TTCA - - - T	TTCAT - - -
TGCATCGT	TGCATCGT

Edit Distance. The edit distance between two strings, also called Levenshtein distance[6], counts the minimum number of operations to change one string into the other. This corresponds with alignment with match score 0, while mismatch and insdel are both −1.

LCS. A string z is a *subsequence* of string x if z can be obtained by deleting symbols from x. Formally z is a subsequence of $x = x_1 \dots x_m$ if we can write $z = x_{i_1} x_{i_2} \dots x_{i_k}$ for $i_1 < i_2 < \cdots < i_k$. A *longest common subsequence* of

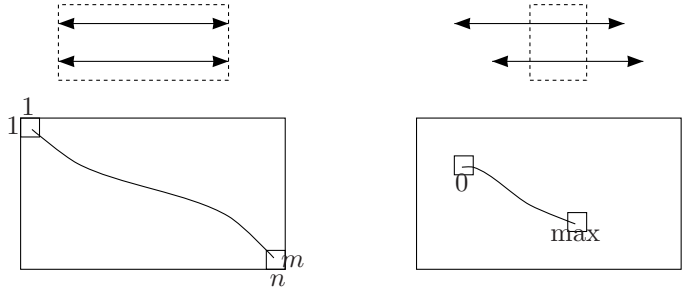


Figure 1.1: Global versus local alignment.

strings x and y is a string z of maximal length such that z is subsequence of both x and y. Although z itself may not be unique, the *length* of a longest common subsequence of given strings is.

The problem of finding a longest common subsequence can be answered by computing the alignment where match is rewarded by $+1$ while mismatch and insdel penalty are both 0.

1.3 Local Alignment

Global alignments attempts to align every character in both sequences. This is useful when the sequences are similar and of roughly equal size. In some cases one expects only parts of the strings to be similar, e.g., when both strings contain a common motif. In such cases one tries to find segments of both strings that are similar, using local alignment (known as Smith-Waterman [13]). This uses a simple adaptation of the global approach, and is the main topic of this section.

Another variant is when one wants to determine whether one string can be seen as extending the other, following a partial overlap. This is motivated by sequence reconstruction based on a set of substrings. Also this can be solved by an adaptation of the dynamic programming technique.

Local alignment. Given two strings x and y, and a scoring system, we want to find substrings x' and y' (of x and y respectively) such that the similarity of x' and y' is maximal.

The main difference with the global version of the algorithm is that we can forget negative values. Whenever a partial alignment reaches a negative value it is reset to zero. As we want to find substrings with maximal alignment we can drop the pieces that give negative contribution. In the same vein the value of the local alignment is not found in the bottom-right corner of the matrix, but rather it is the maximal value found in the matrix. Indeed, extending the

maximum alignment will add negative contribution to the value so far obtained, and this part is skipped when stopping at the maximum.

This means we obtain the following algorithm for the computation of local alignment. It differs in two aspects from global alignment. The initialization sets the borders to zero (not the multiples of the gap penalty), and the maximum for the computation of the non-border cells now includes zero, to avoid negative values.

$$
\begin{array}{rcll}
A[i,0] & = & 0 & 0 \le i \le m \\
A[0,j] & = & 0 & 0 \le j \le n \\
A[i,j] & = & \max \left\{ \begin{array}{lll} A[i,j-1] & + & g \\ A[i-1,j-1] & + & \sigma(x_i, y_j) \\ A[i-1,j] & + & g \\ 0 & & \end{array} \right. & 1 \le i \le m, 1 \le j \le n
\end{array}
$$

1.4 Example. Local alignment of ATTCAT and TGCATCGT with scoring system match 2, mismatch -1, and insdel -1.

The following matrix is computed by the dynamic programming algorithm. It indicates that the score of the optimal local alignment equals 7, which is the maximal value in the matrix.

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

Tracing back the matrix from the maximal position until a zero is reached one finds one of the following alignments.

```
TTCAT      T - CAT
TGCAT      TGCAT
```

Semi-Global Alignment. In some contexts we are interested in specific overlap between the strings x and y. For instance, when we have a set of (overlapping) random segments of a long string we may reconstruct the original long string using the segments, after we have determined their order using the overlap between the strings. Hence we are interested in determining the maximal overlap consisting of a suffix of x and a prefix of y. As another example when y is much shorter than x it is not very useful to consider the global alignment of

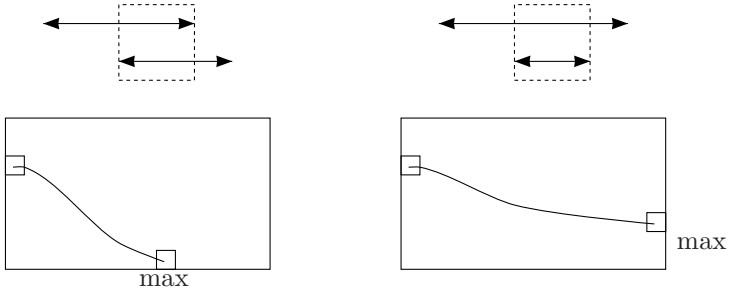


Figure 1.2: Semi-global alignment: finding overlap or containment.

x and y to find the possible position of y within x. See Fig. 1.2 for a pictorial representation of these two cases.

Both these cases share a property with local alignment that gaps at the beginning or the end of one of the strings should not be penalized. As a consequence these problems can be solved in a similar manner. Where initial gaps are free we can include this in the initialization phase of the algorithm (see local alignment), not counting the gap panelty in the first row or column of the matrix. Where final gaps are free we solve this in the final phase of the algoritm. The solution then is not found in the bottom-right cell, but rather is the maximal value on either bottom row or rightmost column.

1.4 More Variants of the Basic Algorithm

Complexity Revisited: Linear Space. The time complexity of the algorithm for determining the optimal alignment equals $\mathcal{O}(mn)$, where m and n are the lengths of the given strings. The space complexity also equals $\mathcal{O}(mn)$, but as we have seen, it can be reduced to $\mathcal{O}(m)$ if we are interested only in the value of the alignment, and not in its traceback.

There is however a clever recursive implementation [5] that computes the optimal alignment in space $\mathcal{O}(m)$, including the traceback. Let us write $\mathrm{sim}(x, y)$ for the similarity of x and y, the value of the optimal alignment.

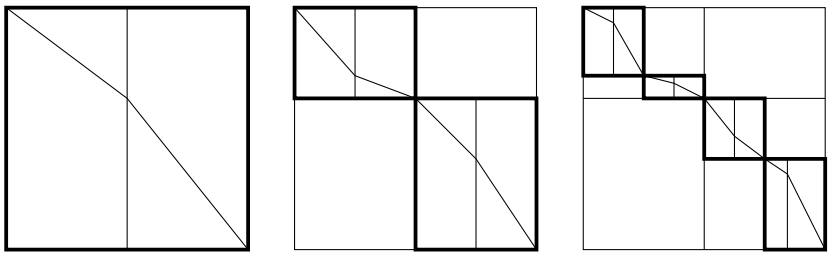
Start by dividing y into two (almost) equal parts $y[1 \cdots \frac{n}{2}]$ and $y[\frac{n}{2} + 1 \cdots n]$. This division in the alignment will also cut the first string into two parts, and (this is important for the recursion) the total score of the alignment can be computed by adding the scores of the two parts. As we do not know where the first string was cut when cutting the alignment in the middle of y we reconstruct that position by looking for the maximal sum of two partial scores.

$$\mathrm{sim}(x, y) = \max_{0 \le k \le m} \left[\mathrm{sim}(x[1 \cdots k], y[1 \cdots \frac{n}{2}]) + \mathrm{sim}(x[k+1 \cdots m], y[\frac{n}{2} + 1 \cdots n]) \right]$$

The partial scores $\mathrm{sim}(x[1 \cdots k], y[1 \cdots \frac{n}{2}])$ $(k = 0, \ldots, m)$ do not have to be computed one-by-one. In fact they form the last column of the alignment matrix

for x and $y[1 \cdots \frac{n}{2}]$. In the same way the partial scores $\mathrm{sim}(x[k+1 \cdots m], y[\frac{n}{2}+1 \cdots n)$ $(k = 0, \ldots, m)$ fill the first column of the alignment matrix for x en $y[\cdots]$, provided it is computed in reverse, from right-to-left, bottom-to-top.

In each step the amount of work halves (as the total matrix size halves). In the end we thus have used $1 + \frac{1}{2} + \frac{1}{4} + \cdots \approx 2$ times the usual amount of work, which again is $\mathcal{O}(mn)$, while in each computation we only need two columns of length at most m.



1.5 Example. Starting with TTCAT and TGCATCGT (as in previous example) we compute the last column of the alignment of TTCAT and prefix TGCA and the first column of the alignment of TTCAT and suffix TCGT (in reverse). The matrix below contains all values, but in reality we keep only two columns in the middle as stated. The amount of work is (roughly) the same as for the ordinary algorithm, $m \cdot n$.

Combining the values on the various positions we get the following scores for the full alignment: $-24+0, -13+13, -9-5, 2-5, 13-13, 7-24$. The best value is 0 which can be obtained at two positions.

From this we conclude that an optimal alignment of the strings can be written as the composition of optimal alignments of T and TGCA, and of TCAT and TCGT. These are computed recursively.

	−	T	G	C	A						
−	0	−6	−12	−18	−24						
T	−6	5	−1	−7	−13	0	−4	−15	−19	−36	T
T	−12	−1	3	−3	−9	13	2	−9	−13	−24	T
C	−18	−7	−3	8	2	−5	8	−3	−7	−18	C
A	−24	−13	−9	2	13	−9	−3	3	−1	−12	A
T	−30	−19	−15	−4	7	−13	−7	−1	5	−6	T
						−24	−18	−12	−6	0	−
						T	C	G	T	−	

Affine Gap Penalty. Usually alignments are considered more relevant if they contain as little gaps as possible. A single long insertion is preferred over a set of small ones.

This can be captured by making the gap cost dependent on the gap length n, a relatively large cost G for starting a gap (*opening* it) and a smaller cost L for each additional position (*extending* it). Such a value is of the form $G + nL$, where n is the length of the gap, and is called *affine*. (A gap of length one has cost $G + L$.)

The basic algorithm can be extended to handle affine costs by computing three values in each position of the matrix, two additional values indicating the score when the gap opening cost has already been accounted for, corresponding to alignments that end with gaps in either top or bottom strand.

Profile Alignment. The familiar alignment procedures are easily modified to accomodate sequences of character frequency vectors, PSSM – position specific scoring matrix, where each vector specifies the distribution of characters at a position in an alignment of several sequences.

In a sense the character at a position is fuzzy rather than exactly determined. The point that we have to adapt in the basic algorithm is the scoring of a letter against such a fuzzy position (or column in the PSSM). The new score is simply the mean of the separate scores for all the different characters, weighted by their frequencies.

Let α be a frequency vector, mapping alphabet Σ into the interval $[0, 1]$. Then for a scoring system σ the score for vector α and symbol a equals $\sigma(\alpha, a) = \sum_{b\in\Sigma} \sigma(b, a)$.

This method is used in Section 1.6 where a given alignment is extended to include several other strings. In fact, also the alignment of profiles against one another is needed for such a multiple alignment. The score then consists of weighted mean of all letter-letter combinations.

1.6 Example. Consider the profile defined by five sequences.

TGGGGGA
CGAGACA
TGGGG - A
TGAGA - A
TGAGGGA

It defines the following PSSM, in absolute numbers, and frequencies.

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

	1	2	3	4	5	6	7
A	0	0	.6	0	.4	0	1
C	.2	0	0	0	0	.2	0
G	0	1	.4	1	.6	.4	0
T	.8	0	0	0	0	0	0
-	0	0	0	0	0	.4	0

The score of symbol A against the fifth column equals $\frac{4}{10}\sigma(A, A) + \frac{6}{10}\sigma(A, G)$

Another approach for alinging sequences against a profile is considered later: *profile hidden Markov models*.

1.5 Heuristics

Dot matrix. The dot plot is a widely used technique to visualize similarities between two sequences. Along the axes of a matrix we put the two sequences, and we place a dot in the cells where both sequences agree. Then the dots are filtered, and only those dots are shown that are part of a longer run.

Banded Alignment. For sequences that are quite similar the alignment traces a path close to the main diagonal of the alignment matrix. This implies that only values near this diagonal contribute to the score computed in the matrix. This can be used as the basis for a heuristic approach. In computing the score of an alignment consider only the cells that lie in a small band along the diagonal. This turns the quadratic algorithm into linear time.

Database Search. Searching a string in a sequence database that is similar to a given query string forces one to build an alignment for the query against the full database. Although the algorithm we have discussed is poynomial, the sheer size of present databases makes this approach unfeasible.

Heuristic techniques have been proposed that improve the speed of the search. Basically the assumption is that a good alignment should contain at least a small sequence where the similarity is exeptionally strong.

For **FASTA**[7] and **BLAST**[1] see the lecture notes of APG [4, Section 2.2].

1.6 Multiple Alignment

Multiple sequence alignment is an extension of pairwise alignment to incorporate more than two sequences at a time. It is often used in identifying conserved sequence regions across a group of sequences hypothesized to be (evolutionarily) related.

Formally the alignment of ℓ sequences over alphabet Σ consists of a sequence of ℓ-dimensional vectors with elements from $\Sigma \cup \{\varepsilon\}$ such that the concatenation of the elements on the i-th row concatenate into the i-th sequence. As in the two-dimensonal case we assume that each of the vectors contains at least one element from Σ.

Sum of pairs. The usual way of scoring a multiple alignment is *sum of pairs*. Each multiple alignment induces alignments between each pair of sequences, by considering only the two corresponding components, discarding a vector when both these components are ε.

A multiple alignment defines pairwise alignments for each pair of involved strings, by restricting to the two rows involved. The sum of pairs score of the multiple alignment is defined as the sum of the $\frac{n(n-1)}{2}$ pairwise alignments defined by the multiple alignment.

Note that the induced pairwise alignments need not to be optimal. Hence the sum of pair score can be bounded using the similarity scores $\mathrm{sim}(x, y)$ of the optimal alignments: $sop(x_1, x_2, \ldots, x_\ell) \leq \sum_{1 \leq i < j \leq \ell} \mathrm{sim}(x_i, x_j)$.

The sum of pair score can also be computed directly from the vectors (the columns) in the multiple alignment. Given a scoring system σ for elements of $\Sigma \cup \{\varepsilon\}$, and setting $\sigma(\varepsilon, \varepsilon) = 0$, the score of vector $(a_1, a_2, \ldots, a_\ell)$ equals $\sum_{1 \leq i < j \leq \ell} \sigma(a_i, a_j)$.

Dynamic Programming. The technique of dynamic programming is theoretically applicable to any number of sequences. However, because it is computationally expensive in both time and memory, it is rarely used for more than three or four sequences in its most basic form. This method requires constructing the ℓ-dimensional equivalent of matrix constructed for two sequences, where ℓ is the number of sequences in the query. For ℓ strings of length m the hypermatrix contains m^ℓ cells, which implies that the problem takes $\mathcal{O}(m^\ell)$ time to compute, for input of size $\mathcal{O}(m \cdot \ell)$.

Star Alignment. For a star alignment we start with a central string s, and we construct an alignment with s for every other string in the collection we want to align. Then these pairwise alignments are joined into one multiple alignment taking s as a guide. For each other string its characters are placed in the columns as determined by the pairwise aligment with s, inserting gaps in s as prescribed by the alignment. This method of joining is sometimes called 'once a gap, always a gap'. Indeed, if one of the pairwise alignments with s contains a gap (in s) this gap will be added to all the other strings (unless s also had a gap in that other alignment). Hence a single ocurrence of a gap might be multiplied, and consequently its negative score.

The central string s is usually taken to be the string that is most similar to the other strings. This can be determined by computing the pairwise alignment for all pairs, and adding all scores for each string. The central string is chosen to be the one that is most similar, i.e., with maximal total similarity to the other strings.

1.7 Example. Assume we start with 5 strings, s_1 = ATTGCCATT, s_2 = ATGGCCATT, s_3 = ATCCAATTTT, s_4 = ATCTTCTT, and s_5 = ACTGACC, and consider the scoring system +1 (match), −1 (mismatch), −2 (gap).

The similarity scores between the strings are given by the following table. We can verify that s_1 has the best (total) score.

	s_1	s_2	s_3	s_4	s_5
s_1		7	−2	0	−3
s_2	7		−2	0	−4
s_3	−2	−2		0	−7
s_4	0	0	0		−3
s_5	−3	−4	−7	−3	

We consider the pairwise alignments with s_1 and the other strings. These have optimal score, but are not necessarily unique.

s_1	ATTGCCATT	s_1	ATTGCCATT - -
s_2	ATGGCCATT	s_3	ATC - CAATTTT

These are easily combined into a common alignment.

s_1	ATTGCCATT - -
s_2	ATGGCCATT - -
s_3	ATC - CAATTTT

Finally we add the two other alignments

s_1	ATTGCCATT	s_1	ATTGCCATT
s_4	ATCTTC - TT	s_5	ACTGACC - -

to obtain

s_1	ATTGCCATT - -
s_2	ATGGCCATT - -
s_3	ATC - CAATTTT
s_4	ATCTTC - TT - -
s_5	ACTGACC - - - -

Example taken from Setubal & Meidanis [12].

Progressive Alignment. A slightly more sophisticated heuristic than star alignment does not simply compute a central string, but determines a tree that captures the similarities between the strings. In this tree the leaves correspond to the strings, and the path length from one leaf to another is assumed to be proportional to the distance from one string to the other. Methods to obtain such a tree are *clustering algorithms*, like Neighbour Joining or UPGMA, that are discussed in Chapter 2.

See the lecture notes of APG [4, Section 2.4] for an account of this method, including the introduction of sequence weights corresponding to the branching structure of the tree.

1.7 BLOSUM62

ftp://ftp.ncbi.nih.gov/blast/matrices/BLOSUM62

See the APG notes [4] for an account how matrices like this are obtained.

```
#  Matrix made by matblas from blosum62.iij
#  * column uses minimum score
#  BLOSUM Clustered Scoring Matrix in 1/2 Bit Units
#  Blocks Database = /data/blocks_5.0/blocks.dat
#  Cluster Percentage: >= 62
#  Entropy =   0.6979, Expected =  -0.5209
   A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V  B  Z  X  *
A  4 -1 -2 -2  0 -1 -1  0 -2 -1 -1 -1 -1 -2 -1  1  0 -3 -2  0 -2 -1  0 -4
R -1  5  0 -2 -3  1  0 -2  0 -3 -2  2 -1 -3 -2 -1 -1 -3 -2 -3 -1  0 -1 -4
N -2  0  6  1 -3  0  0  0  1 -3 -3  0 -2 -3 -2  1  0 -4 -2 -3  3  0 -1 -4
D -2 -2  1  6 -3  0  2 -1 -1 -3 -4 -1 -3 -3 -1  0 -1 -4 -3 -3  4  1 -1 -4
C  0 -3 -3 -3  9 -3 -4 -3 -3 -1 -1 -3 -1 -2 -3 -1 -1 -2 -2 -1 -3 -3 -2 -4
Q -1  1  0  0 -3  5  2 -2  0 -3 -2  1  0 -3 -1  0 -1 -2 -1 -2  0  3 -1 -4
E -1  0  0  2 -4  2  5 -2  0 -3 -3  1 -2 -3 -1  0 -1 -3 -2 -2  1  4 -1 -4
G  0 -2  0 -1 -3 -2 -2  6 -2 -4 -4 -2 -3 -3 -2  0 -2 -2 -3 -3 -1 -2 -1 -4
H -2  0  1 -1 -3  0  0 -2  8 -3 -3 -1 -2 -1 -2 -1 -2 -2  2 -3  0  0 -1 -4
I -1 -3 -3 -3 -1 -3 -3 -4 -3  4  2 -3  1  0 -3 -2 -1 -3 -1  3 -3 -3 -1 -4
L -1 -2 -3 -4 -1 -2 -3 -4 -3  2  4 -2  2  0 -3 -2 -1 -2 -1  1 -4 -3 -1 -4
K -1  2  0 -1 -3  1  1 -2 -1 -3 -2  5 -1 -3 -1  0 -1 -3 -2 -2  0  1 -1 -4
M -1 -1 -2 -3 -1  0 -2 -3 -2  1  2 -1  5  0 -2 -1 -1 -1 -1  1 -3 -1 -1 -4
F -2 -3 -3 -3 -2 -3 -3 -3 -1  0  0 -3  0  6 -4 -2 -2  1  3 -1 -3 -3 -1 -4
P -1 -2 -2 -1 -3 -1 -1 -2 -2 -3 -3 -1 -2 -4  7 -1 -1 -4 -3 -2 -2 -1 -2 -4
S  1 -1  1  0 -1  0  0  0 -1 -2 -2  0 -1 -2 -1  4  1 -3 -2 -2  0  0  0 -4
T  0 -1  0 -1 -1 -1 -1 -2 -2 -1 -1 -1 -1 -2 -1  1  5 -2 -2  0 -1 -1  0 -4
W -3 -3 -4 -4 -2 -2 -3 -2 -2 -3 -2 -3 -1  1 -4 -3 -2 11  2 -3 -4 -3 -2 -4
Y -2 -2 -2 -3 -2 -1 -2 -3  2 -1 -1 -2 -1  3 -3 -2 -2  2  7 -1 -3 -2 -1 -4
V  0 -3 -3 -3 -1 -2 -2 -3 -3  3  1 -2  1 -1 -2 -2  0 -3 -1  4 -3 -2 -1 -4
B -2 -1  3  4 -3  0  1 -1  0 -3 -4  0 -3 -3 -2  0 -1 -4 -3 -3  4  1 -1 -4
Z -1  0  0  1 -3  3  4 -2  0 -3 -3  1 -1 -3 -1  0 -1 -3 -2 -2  1  4 -1 -4
X  0 -1 -1 -1 -2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2  0  0 -2 -1 -1 -1 -1 -1 -4
* -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4  1
```

Bibliography

[1] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, D.J. Lipman (1990). Basic local alignment search tool. Journal of Molecular Biology 215 (3): 403–410. doi:10.1006/jmbi.1990.9999
▷ alignment heuristics, BLAST, Section 1.5

[2] K.S. Booth, G.S. Lueker: Testing for the consecutive ones property, interval graphs, and planarity using PQ-tree algorithms, Journal of Computational Systems Science, Vol. 13 (1976), pp. 335-379.
▷ physical mapping, Section 3.5

[3] Fitch and Margoliash, Construction of Phylogenetic Trees, Science Vol. 155, 20 Jan. 1967.

[4] A.P. Gultyaev, Computational Molecular Biology, Application-oriented view, Leiden University, 2009.

[5] D.S. Hirschberg. Algorithms for the Longest Common Subsequence Problem, Journal of the ACM, 24 (1977) 664–675.
▷ linear space alignment, Section 1.4

[6] V.I. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady 10 (1966):707–710.
▷ global alignment, edit distance, Section 1.2

[7] D.J. Lipman, W.R. Pearson, Rapid and sensitive protein similarity searches. Science. 1985 Mar 22;227(4693):1435-41.
▷ alignment heuristics, FASTA, Section 1.5

[8] S.B. Needleman, C.D. Wunsch. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. J Mol Biol 48 (3): 443–53. doi:10.1016/0022-2836(70)90057-4
▷ global alignment, Section 1.2

[9] N. Saitou and M. Nei, (1987). The neighbor-joining method: a new method for reconstructing phylogenetic trees. Mol. Biol. Evol. 4(4):406-425
▷ phylogeny, unrooted trees, Section 2.5

[10] D. Sankoff (1975). Minimal mutation trees of sequences. SIAM Journal of Applied Mathematics 28: 35-42.
▷ character based, small parsimony, Sankoff algorithm, Section 2.3

[11] R. Shamir, Algorithms in Molecular Biology, lecture notes, 2001-2002, Tel Aviv University School of Computer Science. www.cs.tau.ac.il/ rshamir/algmb/01/algmb01.html

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

[13] T.F. Smith, M.S. Waterman (1981). Identification of Common Molecular Subsequences. Journal of Molecular Biology 147: 195–197. doi:10.1016/0022-2836(81)90087-5
▷ local alignment, Section 1.3