$\log \frac{g^n_{i,j}}{f(j)}$.

For large n, the resulting PAM matrices often allow one to find related proteins even when there are practically no matches in the alignment. In this case, the underlying nucleotide sequences are so diverged that their comparison usually fails to find any statistically significant similarities. For example, the similarity between the cancer-causing ν-sis oncogene and the growth factor PDGF would probably have remained undetected had Russell Doolittle and colleagues not transformed the nucleotide sequences into amino acid sequences prior to performing the comparison.

6.8 Local Sequence Alignment

The Global Alignment problem seeks similarities between two entire strings. This is useful when the similarity between the strings extends over their entire length, for example, in protein sequences from the same protein family. These protein sequences are often very conserved and have almost the same length in organisms ranging from fruit flies to humans. However, in many biological applications, the score of an alignment between two substrings of **v** and **w** might actually be larger than the score of an alignment between the entireties of **v** and **w**.

For example, *homeobox* genes, which regulate embryonic development, are present in a large variety of species. Although homeobox genes are very different in different species, one region in each gene—called the *homeodomain*—is highly conserved. The question arises how to find this conserved area and ignore the areas that show little similarity. In 1981 Temple Smith and Michael Waterman proposed a clever modification of the global sequence alignment dynamic programming algorithm that solves the Local Alignment problem.

Figure 6.16 presents the comparison of two hypothetical genes **v** and **w** of the same length with a conserved domain present at the beginning of **v** and at the end of **w**. For simplicity, we will assume that the conserved domains in these two genes are identical and cover one third of the entire length, n, of these genes. In this case, the path from $source$ to $sink$ capturing the similarity between the homeodomains will include approximately $\frac{2}{3}n$ horizontal edges, $\frac{1}{3}n$ diagonal match edges (corresponding to homeodomains), and $\frac{2}{3}n$ vertical edges. Therefore, the score of this path is

$$-\frac{2}{3}n\sigma + \frac{1}{3}n - \frac{2}{3}n\sigma = n\left(\frac{1}{3} - \frac{4}{3}\sigma\right)$$

However, this path contains so many indels that it is unlikely to be the highest scoring alignment. In fact, biologically irrelevant diagonal paths from the source to the sink will likely have a higher score than the biologically relevant alignment, since mismatches are usually penalized less than indels. The expected score of such a diagonal path is $n(\frac{1}{4} - \frac{3}{4}\mu)$ since every diagonal edge corresponds to a match with probability $\frac{1}{4}$ and mismatch with probability $\frac{3}{4}$. Since $(\frac{1}{3} - \frac{4}{3}\sigma) < (\frac{1}{4} - \frac{3}{4}\mu)$ for many settings of indel and mismatch penalties, the global alignment algorithm will miss the correct solution of the real biological problem, and is likely to output a biologically irrelevant near-diagonal path. Indeed, figure 6.16 bears exactly this observation.

When biologically significant similarities are present in certain parts of DNA fragments and are not present in others, biologists attempt to maximize the alignment score $s(v_i \dots v_{i'}, w_j \dots w_{j'})$, over all substrings $v_i \dots v_{i'}$ of **v** and $w_j \dots w_{j'}$ of **w**. This is called the Local Alignment problem since the alignment does not necessarily extend over the entire string length as it does in the Global Alignment problem.

Local Alignment Problem:
Find the best local alignment between two strings.

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

Output: Substrings of **v** and **w** whose global alignment, as defined by δ, is maximal among all global alignments of all substrings of **v** and **w**.

The solution to this seemingly harder problem lies in the realization that the Global Alignment problem corresponds to finding the longest local path between vertices $(0, 0)$ and (n, m) in the edit graph, while the Local Alignment problem corresponds to finding the longest path among paths between *arbitrary vertices* (i, j) and (i', j') in the edit graph. A straightforward and inefficient approach to this problem is to find the longest path between every pair of vertices (i, j) and (i', j'), and then to select the longest of these computed paths.[10] Instead of finding the longest path from every vertex (i, j) to every other vertex (i', j'), the Local Alignment problem can be reduced to finding the longest paths from the *source* (0,0) to every other vertex by

10. This will result in a very slow algorithm with $O(n^4)$ running time: there are roughly n^2 pairs of vertices (i, j) and computing local alignments starting at each of them typically takes $O(n^2)$ time.

```
--T--CC-C-AGT--TATGT-CAGGGGACACG--A-GCATGCAGA-GAC
  |  || |  ||  | | | |||    || |  | |  | ||||  |
AATTGCCGCC-GTCGT-T-TTCAG----CA-GTTATG--T-CAGAT--C

                  tccCAGTTATGTCAGgggacacgagcatgcagagac
                    ||||||||||||
aattgccgccgtcgttttcagCAGTTATGTCAGatc
```

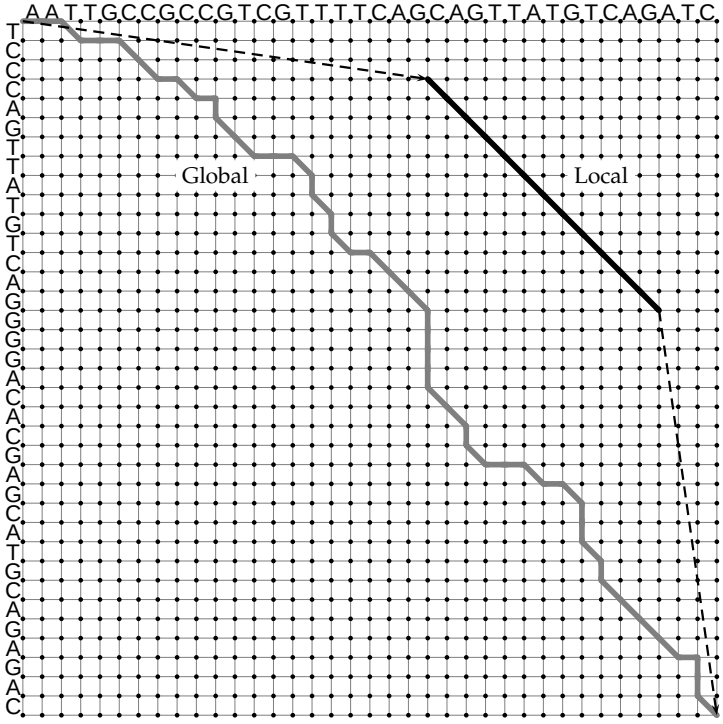


Figure 6.16 (a) Global and (b) local alignments of two hypothetical genes that each have a conserved domain. The local alignment has a much worse score according to the global scoring scheme, but it correctly locates the conserved domain.

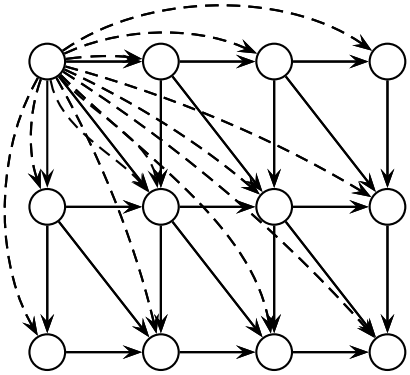


Figure 6.17 The Smith-Waterman local alignment algorithm introduces edges of weight 0 (here shown with dashed lines) from the source vertex $(0, 0)$ to every other vertex in the edit graph.

adding edges of weight 0 in the edit graph. These edges make the source vertex (0,0) a predecessor of every vertex in the graph and provide a "free ride" from the source to any other vertex (i, j). A small difference in the following recurrence reflects this transformation of the edit graph (shown in figure 6.17):

$$s_{i,j} = \max \begin{cases} 0 \\ s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \\ s_{i-1,j-1} + \delta(v_i, w_j) \end{cases}$$

The largest value of $s_{i,j}$ over the whole edit graph represents the score of the best local alignment of $\mathbf{v}$ and $\mathbf{w}$; recall that in the Global Alignment problem, we simply looked at $s_{n,m}$. The difference between local and global alignment is illustrated in figure 6.16 (top).

Optimal local alignment reports only the longest path in the edit graph. At the same time, several local alignments may have biological significance and methods have been developed to find the k best nonoverlapping local alignments. These methods are particularly important for comparison of multidomain proteins that share similar blocks that have been shuffled in one protein compared to another. In this case, a single local alignment representing all significant similarities may not exist.

6.9 Alignment with Gap Penalties

Mutations are usually caused by errors in DNA replication. Nature frequently deletes or inserts entire substrings as a unit, as opposed to deleting or inserting individual nucleotides. A *gap* in an alignment is defined as a contiguous sequence of spaces in one of the rows. Since insertions and deletions of substrings are common evolutionary events, penalizing a gap of length x as $-\sigma x$ is cruel and unusual punishment. Many practical alignment algorithms use a softer approach to gap penalties and penalize a gap of x spaces by a function that grows slower than the sum of penalties for x indels.

To this end, we define *affine gap penalties* to be a linearly weighted score for large gaps. We can set the score for a gap of length x to be $-(\rho + \sigma x)$, where $\rho > 0$ is the penalty for the introduction of the gap and $\sigma > 0$ is the penalty for each symbol in the gap (ρ is typically large while σ is typically small). Though this may seem to be complicating our alignment approach, it turns out that the edit graph representation of the problem is robust enough to accommodate it.

Affine gap penalties can be accommodated by adding "long" vertical and horizontal edges in the edit graph (e.g., an edge from (i, j) to $(i + x, j)$ of length $-(\rho + \sigma x)$ and an edge from (i, j) to $(i, j + x)$ of the same length) from each vertex to every other vertex that is either east or south of it. We can then apply the same algorithm as before to compute the longest path in this graph. Since the number of edges in the edit graph for affine gap penalties increases, at first glance it looks as though the running time for the alignment algorithm also increases from $O(n^2)$ to $O(n^3)$, where n is the longer of the two string lengths.[11] However, the following three recurrences keep the running time down:

$$\overset{\downarrow}{s}_{i,j} = \max \begin{cases} \overset{\downarrow}{s}_{i-1,j} - \sigma \\ s_{i-1,j} - (\rho + \sigma) \end{cases}$$

$$\vec{s}_{i,j} = \max \begin{cases} \vec{s}_{i,j-1} - \sigma \\ s_{i,j-1} - (\rho + \sigma) \end{cases}$$

11. The complexity of the corresponding Longest Path in a DAG problem is defined by the number of edges in the graph. Adding long horizontal and vertical edges imposed by affine gap penalties increases the number of edges by a factor of n.

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

The variable $\overset{\downarrow}{s}_{i,j}$ computes the score for alignment between the i-prefix of $\mathbf{v}$ and the j-prefix of $\mathbf{w}$ ending with a deletion (i.e., a gap in $\mathbf{w}$), while the variable $\overrightarrow{s}_{i,j}$ computes the score for alignment ending with an insertion (i.e., a gap in $\mathbf{v}$). The first term in the recurrences for $\overset{\downarrow}{s}_{i,j}$ and $\overrightarrow{s}_{i,j}$ corresponds to extending the gap, while the second term corresponds to initiating the gap. Essentially, $\overset{\downarrow}{s}_{i,j}$ and $\overrightarrow{s}_{i,j}$ are the scores of optimal paths that arrive at vertex (i, j) via vertical and horizontal edges correspondingly.

Figure 6.18 further explains how alignment with affine gap penalties can be reduced to the Manhattan Tourist problem in the appropriate city grid. In this case the city is built on three levels: the bottom level built solely with vertical $\downarrow$ edges with weight $-\sigma$; the middle level built with diagonal edges of weight $\delta(v_i, w_j)$; and the upper level, which is built from horizontal edges $\rightarrow$ with weight $-\sigma$. The lower level corresponds to gaps in sequence $\mathbf{w}$, the middle level corresponds to matches and mismatches, and the upper level corresponds to gaps in sequence $\mathbf{v}$. Also, in this graph there are two edges from each vertex $(i, j)_{middle}$ at the middle level that connect this vertex with vertex $(i + 1, j)_{lower}$ at the lower level and with vertex $(i, j + 1)_{upper}$ at the upper level. These edges model a start of the gap and have weight $-(\rho + \sigma)$. Finally, one has to introduce zero-weight edges connecting vertices $(i, j)_{lower}$ and $(i, j)_{upper}$ with vertex $(i, j)_{middle}$ at the middle level (these edges model the end of the gap). In effect, we have created a rather complicated graph, but the same algorithm works with it.

We have now introduced a number of pairwise sequence comparison problems and shown that they can all be solved by what is essentially the same dynamic programming algorithm applied to a suitably built Manhattan-style city. We will now consider other applications of dynamic programming in bioinformatics.

6.10 Multiple Alignment

The goal of protein sequence comparison is to discover structural or functional similarities among proteins. Biologically similar proteins may not exhibit a strong sequence similarity, but we would still like to recognize resem-

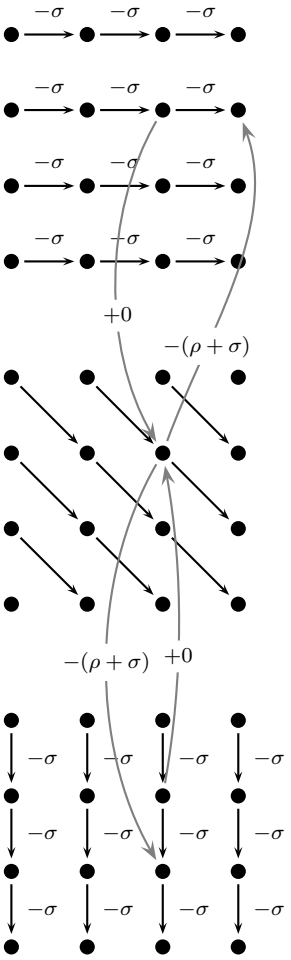
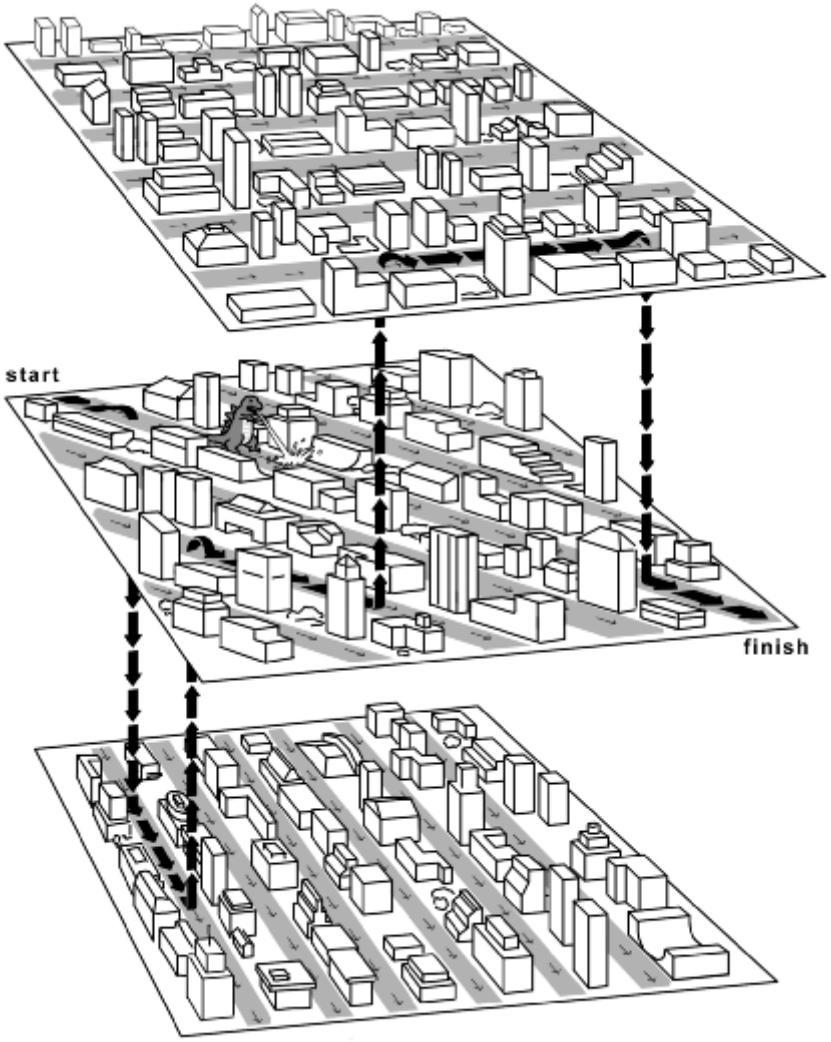


Figure 6.18 A three-level edit graph for alignment with affine gap penalties. Every vertex (i, j) in the middle level has one outgoing edge to the upper level, one outgoing edge to the lower level, and one incoming edge each from the upper and lower levels.


start
finish

```
--T--CC-C-AGT--TATGT-CAGGGGACACG--A-GCATGCAGA-GAC
  |  || |  ||  | | | |||    || |  | |  | ||||  |
AATTGCCGCC-GTCGT-T-TTCAG----CA-GTTATG--T-CAGAT--C
 ||||| |   X|||| |          || XXX|||  | ||| |  |
-ATTGC-G--ATTCGTAT------GGGACA-TGGATGCATGCAG-TGAC
```

Figure 6.19 Multiple alignment of three sequences.

blance even when the sequences share only weak similarities.[12] If sequence similarity is weak, pairwise alignment can fail to identify biologically related sequences because weak pairwise similarities may fail statistical tests for significance. However, simultaneous comparison of many sequences often allows one to find similarities that are invisible in pairwise sequence comparison.

Let $\mathbf{v}_1, \ldots, \mathbf{v}_k$ be k strings of length $n_1, \ldots, n_k$ over an alphabet $\mathcal{A}$. Let $\mathcal{A}'$ denote the extended alphabet $\mathcal{A} \bigcup \{-\}$, where '$-$' denotes the space character (reserved for insertions and deletions). A *multiple alignment* of strings $\mathbf{v_1}, \ldots, \mathbf{v_k}$ is specified by a $k \times n$ matrix A, where $n \geq \max_{1 \leq i \leq k} n_i$. Each element of the matrix is a member of $\mathcal{A}'$, and each row i contains the characters of $\mathbf{v_i}$ in order, interspersed with $n - n_i$ spaces (figure 6.19). We also assume that every column of the multiple alignment matrix contains at least one symbol from $\mathcal{A}$, that is, no column in a multiple alignment contains only spaces. The multiple alignment matrix we have constructed is a generalization of the pairwise alignment matrix to $k > 2$ sequences. The score of a multiple alignment is defined to be the sum of scores of the columns, with the optimal alignment being the one that maximizes the score. Just as it was in section 4.5, the consensus of an alignment is a string of the most common characters in each column of the multiple alignment. At this point, we will use a very general scoring function that is defined by a k-dimensional matrix δ of size $|\mathcal{A}'| \times \ldots \times |\mathcal{A}'|$ that describes the scores of all possible combinations of k symbols from $\mathcal{A}'$.[13]

A straightforward dynamic programming algorithm in the k-dimensional edit graph formed from k strings solves the Multiple Alignment problem.

12. Sequences that code for proteins that perform the same function are likely to be somehow related but it may be difficult to decide whether this similarity is significant or happens just by chance.

13. This is a k-dimensional scoring matrix rather than the two-dimensional $|\mathcal{A}'| \times |\mathcal{A}'|$ matrix for pairwise alignment (which is a multiple alignment with $k = 2$).

For example, suppose that we have three sequences $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$, and that we want to find the "best" alignment of all three. Every multiple alignment of three sequences corresponds to a path in the three-dimensional Manhattan-like edit graph. In this case, one can apply the same logic as we did for two dimensions to arrive at a dynamic programming recurrence, this time with more terms to consider. To get to vertex (i, j, k) in a three-dimensional edit graph, you could come from any of the following predecessors (note that $\delta(x, y, z)$ denotes the score of a column with letters x, y, and z, as in figure 6.20):

1. $(i-1, j, k)$ for score $\delta(u_i, -, -)$
2. $(i, j-1, k)$ for score $\delta(-, v_j, -)$
3. $(i, j, k-1)$ for score $\delta(-, -, w_k)$
4. $(i-1, j-1, k)$ for score $\delta(u_i, v_j, -)$
5. $(i-1, j, k-1)$ for score $\delta(u_i, -, w_k)$
6. $(i, j-1, k-1)$ for score $\delta(-, v_j, w_k)$
7. $(i-1, j-1, k-1)$ for score $\delta(u_i, v_j, w_k)$

We create a three-dimensional dynamic programming array $\mathbf{s}$ and it is easy to see that the recurrence for $s_{i,j,k}$ in the three-dimensional case is similar to the recurrence in the two-dimensional case (fig. 6.21). Namely,

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

Unfortunately, in the case of k sequences, the running time of this approach is $O((2n)^k)$, so some improvements of the exact algorithm, and many heuristics for suboptimal multiple alignments, have been proposed. A good heuristic would be to compute all $\binom{k}{2}$ optimal pairwise alignments between every pair of strings and then combine them together in such a way that pairwise alignments induced by the multiple alignment are close to the optimal

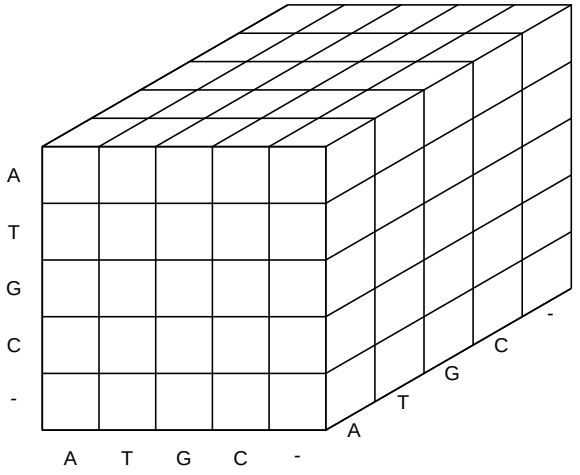


Figure 6.20 The scoring matrix, δ, used in a three-sequence alignment.

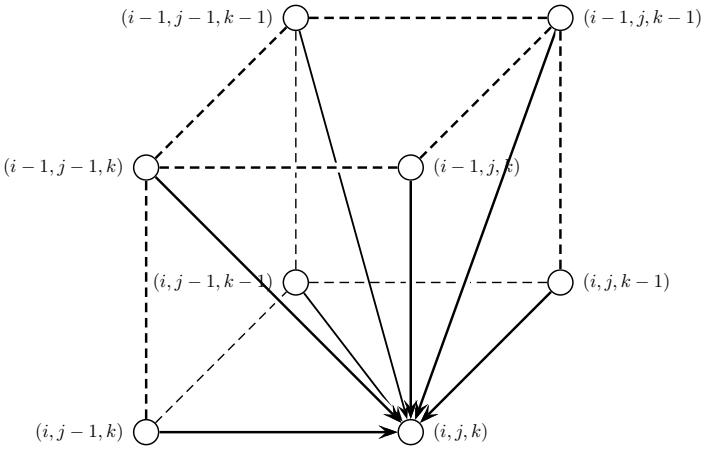


Figure 6.21 A cell in the alignment graph between three sequences.

ones. Unfortunately, it is not always possible to combine optimal pairwise alignments into a multiple alignment since some pairwise alignments may be incompatible. For example, figure 6.22 (a) shows three sequences whose optimal pairwise alignment can be combined into a multiple alignment, whereas (b) shows three sequences that cannot be combined. As a result, some multiple alignment algorithms attempt to combine some compatible subset of optimal pairwise alignments into a multiple alignment.

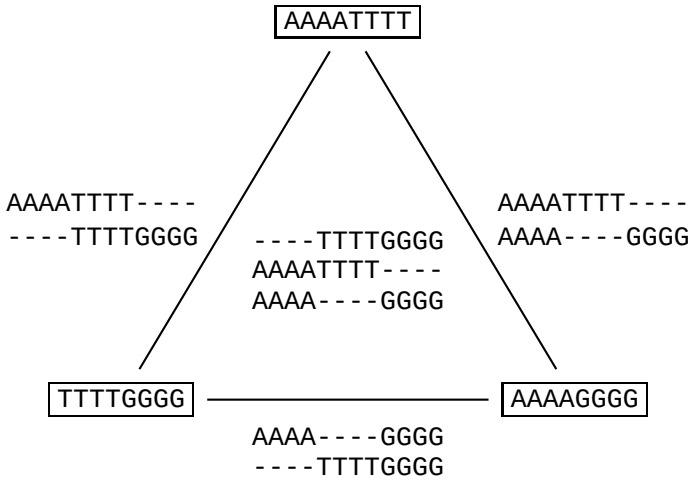
Another approach to do this uses one particularly strong pairwise alignment as a building block for the multiple k-way alignment, and iteratively adds one string to the growing multiple alignment. This greedy *progressive multiple alignment* heuristic selects the pair of strings with greatest similarity and merges them together into a new string following the principle "once a gap, always a gap."[14] As a result, the multiple alignment of k sequences is reduced to the multiple alignment of $k-1$ sequences. The motivation for the choice of the closest strings at the early steps of the algorithm is that close strings often provide the most reliable information about a real alignment. Many popular iterative multiple alignment algorithms, including the tool CLUSTAL, use similar strategies.

Although progressive multiple alignment algorithms work well for very close sequences, there are no performance guarantees for this approach. The problem with progressive multiple alignment algorithms like CLUSTAL is that they may be misled by some spuriously strong pairwise alignment, in effect, a bad seed. If the very first two sequences picked for building multiple alignment are aligned in a way that is incompatible with the optimal multiple alignment, the error in this initial pairwise alignment will propagate all the way through to the whole multiple alignment. Many multiple alignment algorithms have been proposed, and even with systematic deficiencies such as the above they remain quite useful in computational biology.

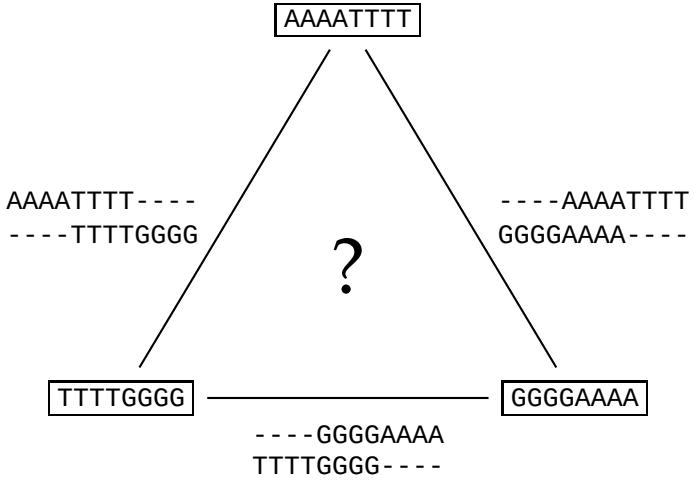
We have described multiple alignment for k sequences as a generalization of the Pairwise Alignment problem, which assumed the existence of a k-dimensional scoring matrix δ. Since such k-dimensional scoring matrices are not very practical, we briefly describe two other scoring approaches that are more biologically relevant. The choice of the scoring function can drastically affect the quality of the resulting alignment, and no single scoring approach is perfect in all circumstances.

The columns of a multiple alignment of k sequences describe a path of

14. Essentially, this principle states that once a gap has been introduced into the alignment it will never close, even if that would lead to a better overall score.



(a) Compatible pairwise alignments



(b) Incompatible pairwise alignments

Figure 6.22 Given three sequences, it might be possible to combine their pairwise alignment into a multiple alignment (a), but it might not be (b).