## Longest Common Subsequences

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

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

and a sequence of positions in **w**,

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

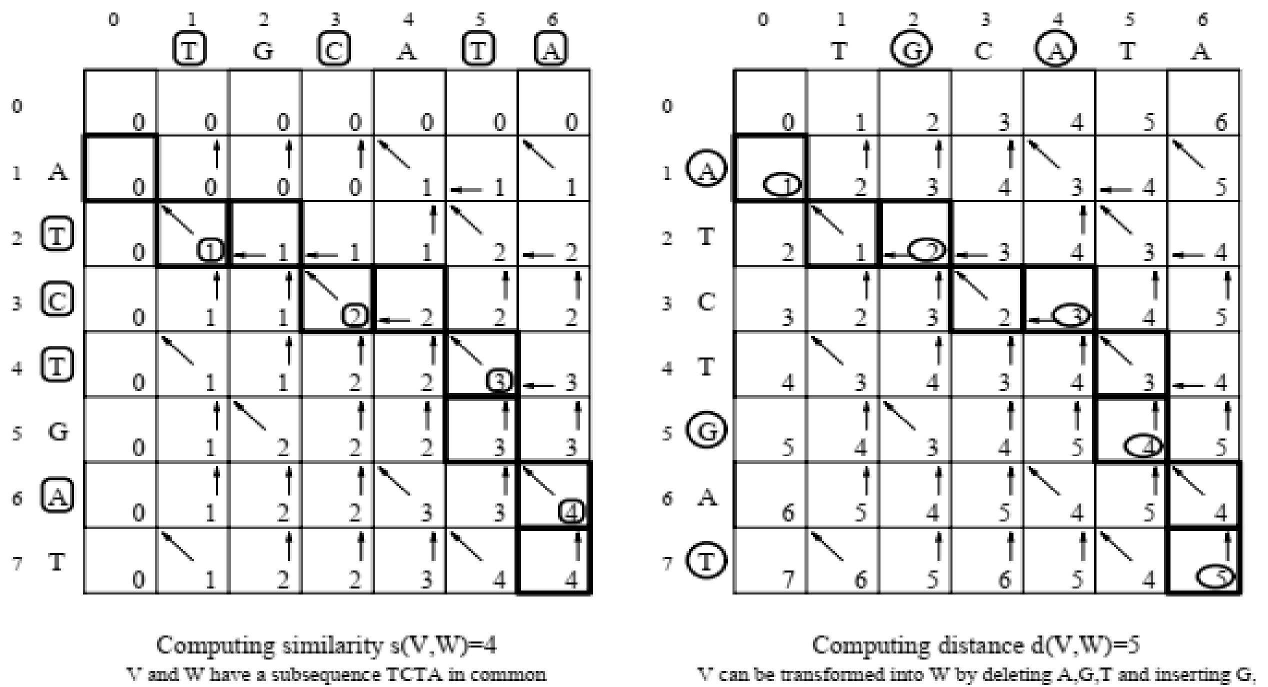
such that the symbols at the corresponding positions in **v** and **w** coincide:

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

For example, TCTA is a common to both A**TCT**GA**T** and **T**G**C**A**TA**.

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

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




Alignment:

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

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

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

---

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

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

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

---

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

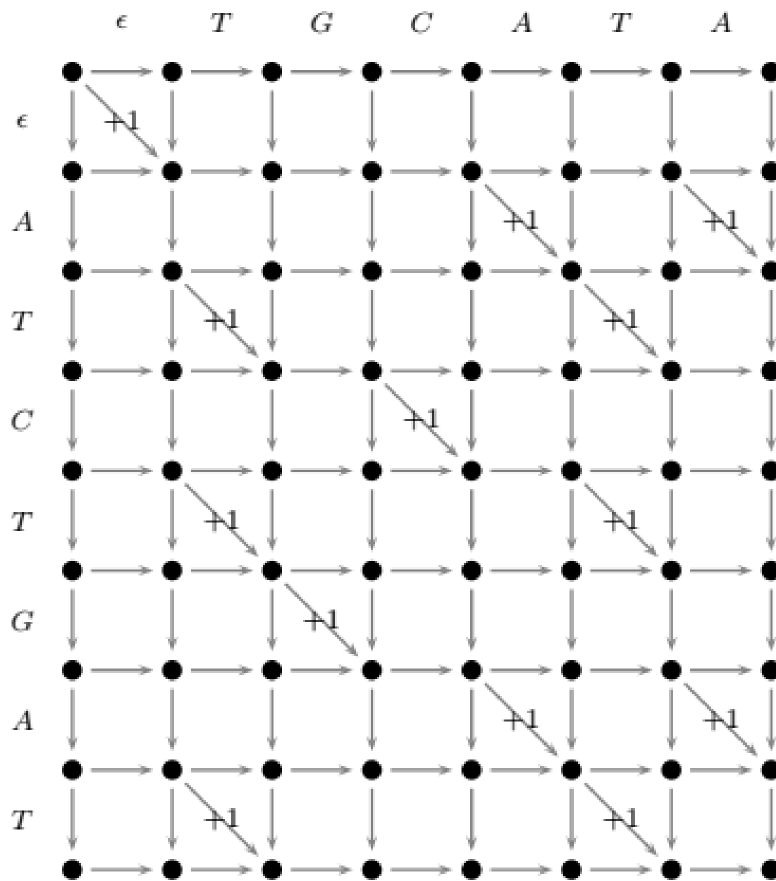



**Figure 6.15** An LCS edit graph.

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

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

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

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

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

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

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

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

```
LCS(v, w)
1  for i ← 0 to n
2      s_{i,0} ← 0
3  for j ← 1 to m
4      s_{0,j} ← 0
5  for i ← 1 to n
6      for j ← 1 to m
```

$$7 \qquad s_{i,j} \leftarrow \max \begin{cases} s_{i-1,j} \\ s_{i,j-1} \\ s_{i-1,j-1}+1, & \text{if } v_i = w_j \end{cases}$$

$$8 \qquad b_{i,j} \leftarrow \begin{cases} \text{“}\uparrow\text{”} & \text{if } s_{i,j} = s_{i-1,j} \\ \text{“}\leftarrow\text{”} & \text{if } s_{i,j} = s_{i,j-1} \\ \text{“}\nwarrow\text{”}, & \text{if } s_{i,j} = s_{i-1,j-1}+1 \end{cases}$$

```
9  return (s_{n,m}, b)
```

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

```
PRINTLCS(b, v, i, j)
 1  if i = 0 or j = 0
 2      return
 3  if b_{i,j} = "↖"
 4      PRINTLCS(b, v, i − 1, j − 1)
 5      print v_i
 6  else
 7      if b_{i,j} = "↑"
 8          PRINTLCS(b, v, i − 1, j)
 9      else
10          PRINTLCS(b, v, i, j − 1)
```

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

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

## Global Sequence Alignment

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

---

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

**Input:** Strings **v**, **w** and a scoring matrix $\delta$.

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

---

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

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

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

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

The corresponding recurrence can be rewritten as

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

We can again store similar "backtracking pointer" information while calculating the dynamic programming table, and from this reconstruct the alignment. We remark that the LCS problem is the Global Alignment problem with the parameters $\mu = 0$, $\sigma = 0$ (or, equivalently, $\mu = \infty$, $\sigma = 0$).

## Scoring Alignments

While the scoring matrices for DNA sequence comparison are usually defined only by the parameters $\mu$ (mismatch penalty) and $\sigma$ (indel penalty), scoring matrices for sequences in the amino acid alphabet of proteins are quite involved. The common matrices for protein sequence comparison, *point accepted mutations (PAM)* and *block substitution (BLOSUM)*, reflect the frequency with which amino acid $x$ replaces amino acid $y$ in evolutionarily related sequences.

Random mutations of the nucleotide sequence within a gene may change the amino acid sequence of the corresponding protein. Some of these mutations do not drastically alter the protein's structure, but others do and impair the protein's ability to function. While the former mutations usually do not affect the fitness of the organism, the latter often do. Therefore some amino acid substitutions are commonly found throughout the process of molecular evolution and others are rare: `Asn`, `Asp`, `Glu`, and `Ser` are the most "mutable" amino acids while `Cys` and `Trp` are the least mutable. For example, the probability that `Ser` mutates into `Phe` is roughly three times greater than the probability that `Trp` mutates into `Phe`. Knowledge of the types of changes that are most and least common in molecular evolution allows biologists to construct the amino acid scoring matrices and to produce biologically adequate sequence alignments. As a result, in contrast to nucleotide sequence comparison, the optimal alignments of amino acid sequences may have very few matches (if any) but still represent biologically adequate alignments. The entry of amino acid scoring matrix $\delta(i, j)$ usually reflects how often the amino acid $i$ substitutes the amino acid $j$ in the alignments of related protein sequences. If one is provided with a large set of alignments of

related sequences, then computing $\delta(i,j)$ simply amounts to counting how many times the amino acid $i$ is aligned with amino acid $j$. A "minor" complication is that to build this set of biologically adequate alignments one needs to know the scoring matrix! Fortunately, in many cases the alignment of very similar sequences is so obvious that it can be constructed even without a scoring matrix, thus resolving this predicament. For example, if proteins are 90% identical, even a naive scoring matrix (e.g., a matrix that gives premium $+1$ for matches and penalties $-1$ for mismatches and indels) would do the job. After these "obvious" alignments are constructed they can be used to compute a scoring matrix $\delta$ that can be used iteratively to construct less obvious alignments.

This simplified description hides subtle details that are important in the construction of scoring matrices. The probability of `Ser` mutating into `Phe` in proteins that diverged 15 million years ago (e.g., related proteins in mouse and rat) is smaller than the probability of the `Ser` $\rightarrow$ `Phe` mutation in proteins that diverged 80 million years ago (e.g., related proteins in mouse and human). This observation implies that the best scoring matrices to compare two proteins depends on how similar these organisms are.

Biologists get around this problem by first analyzing extremely similar proteins, for example, proteins that have, on average, only one mutation per 100 amino acids. Many proteins in human and chimpanzee fulfill this requirement. Such sequences are defined as being *one PAM unit diverged* and to a first approximation one can think of a PAM unit as the amount of time in which an "average" protein mutates 1% of its amino acids. The *PAM 1* scoring matrix is defined from many alignments of extremely similar proteins as follows.

Given a set of base alignments, define $f(i,j)$ as the total number of times amino acids $i$ and $j$ are aligned against each other, divided by the total number of aligned positions. We also define $g(i,j)$ as $\frac{f(i,j)}{f(i)}$, where $f(i)$ is the frequency of amino acid $i$ in all proteins from the data set. $g(i,j)$ defines the probability that an amino acid $i$ mutates into amino acid $j$ within 1 PAM unit. The $(i,j)$ entry of the *PAM 1* matrix is defined as $\delta(i,j) = \log \frac{f(i,j)}{f(i) \cdot f(j)} = \log \frac{g(i,j)}{f(j)}$ $(f(i) \cdot f(j)$ stands for the frequency of aligning amino acid $i$ against amino acid $j$ that one expects simply by chance). The *PAM n* matrix can be defined as the result of applying the PAM 1 matrix $n$ times. If $\mathbf{g}$ is the $20 \times 20$ matrix of frequencies $g(i,j)$, then $\mathbf{g}^n$ (multiplying this matrix by itself $n$ times) gives the probability that amino acid $i$ mutates into amino acid $j$ during $n$ PAM units. The $(i,j)$ entry of the PAM $n$ matrix is defined as

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

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 $\nu$-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.

## 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 $s(v_i \ldots v_{i'}, w_j \ldots w_{j'})$, over all substrings $v_i \ldots v_{i'}$ of $\mathbf{v}$ and $w_j \ldots w_{j'}$ of $\mathbf{w}$. 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 $\mathbf{v}$ and $\mathbf{w}$ and a scoring matrix $\delta$.

**Output:** Substrings of $\mathbf{v}$ and $\mathbf{w}$ whose global alignment, as defined by $\delta$, is maximal among all global alignments of all substrings of $\mathbf{v}$ and $\mathbf{w}$.

---

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
```

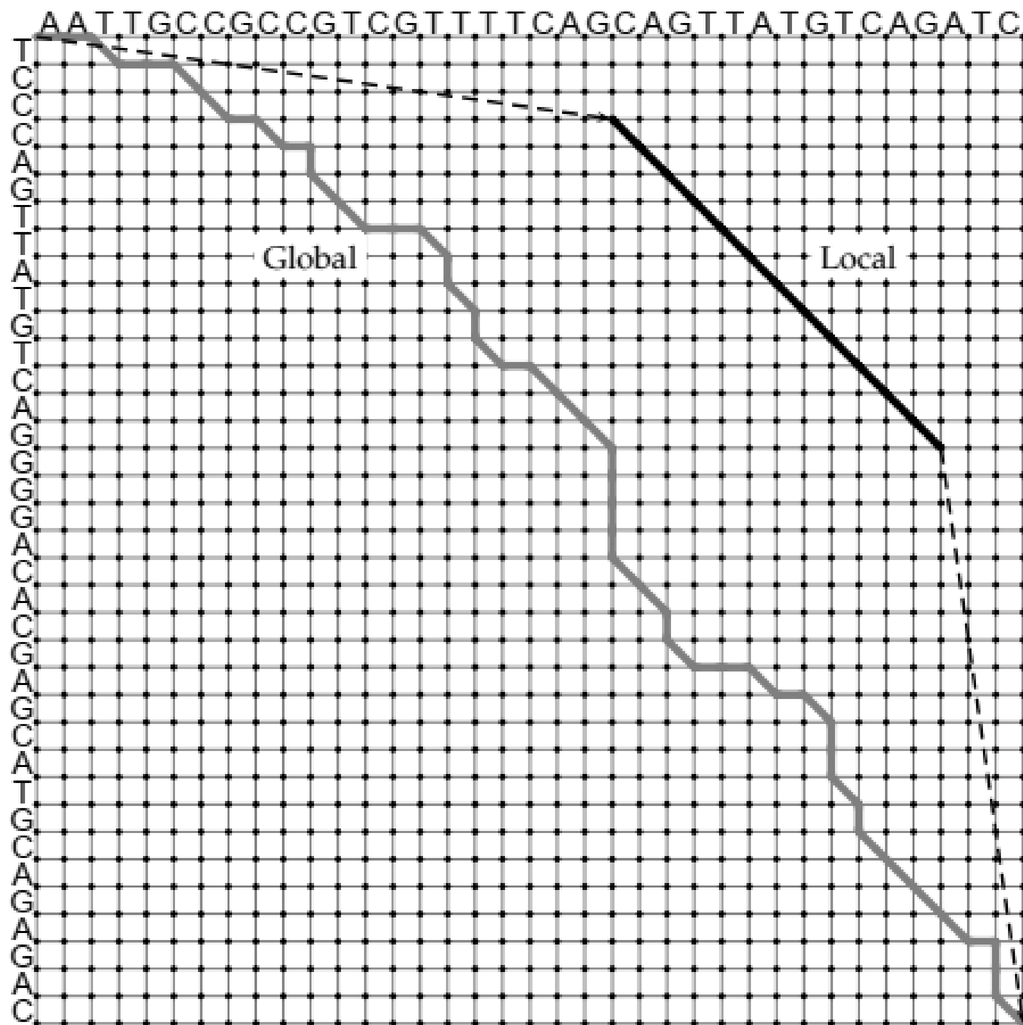



**Figure 6.16** (a) Global and (b) local alignments of two hypothetical genes that each have a conserved domain. The local alignment has a much worse score according to the global scoring scheme, but it correctly locates the conserved domain.

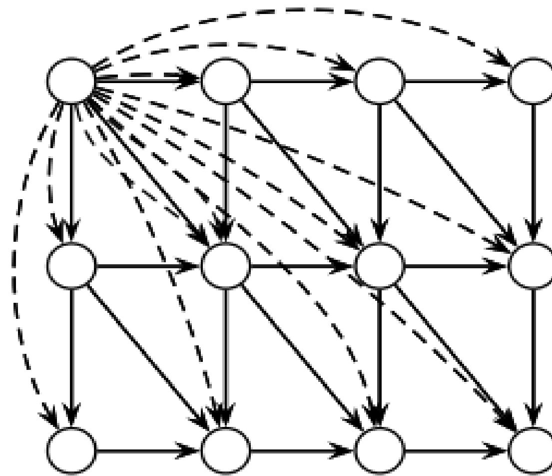


**Figure 6.17** The Smith-Waterman local alignment algorithm introduces edges of weight 0 (here shown with dashed lines) from the source vertex $(0,0)$ to every other vertex in the edit graph.

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 **v** and **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.