Global alignment with general and affine gapcost
Global alignment

Match / sub columns

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>c</th>
<th>g</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>g</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>t</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

gap / indel columns

gapcost: 0

Note about gap cost

In general: cost of “gap block” = g(k), where k is the gap length

Our examples:

\[ g(k) = 0 \cdot k \]  “zero gap cost”
\[ g(k) = -1 \cdot k \]  “linear gap cost”

Many programs: \[ g(k) = a + b \cdot k \]  “affine gap cost”
Biological observation: longer insertions and deletions (indels) are preferred to shorter indels, i.e. a "good" alignment tends to few long indels rather than many short indels ...

Can the simple algorithm for pairwise alignment be adapted to reflect this additional biological insight, i.e. a better model of biology?

Yes, we introduce the concept of a gapcost-function $g(k)$ which gives the cost/penalty for a block of $k$ consecutive insertions or deletions ...

Example

<table>
<thead>
<tr>
<th>A</th>
<th>T</th>
<th>A</th>
<th>C</th>
<th>A</th>
<th>-</th>
<th>-</th>
<th>C</th>
<th>G</th>
<th>C</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>T</td>
<td>-</td>
<td>C</td>
<td>T</td>
<td>C</td>
<td>C</td>
<td>A</td>
<td>C</td>
<td>-</td>
<td>C</td>
</tr>
</tbody>
</table>

$s(A,A) + s(T,T) + g(1) + s(C,C) + s(A,T) + g(3) + s(C,C) + g(1) + s(C,C) + s(A,T)$
Modeling gapcost

**Biological observation:** longer insertions and deletions (indels) are more common than shorter indels, i.e. a “good” alignment tends to few long indels rather than many short indels ...

Can the simple algorithm for pairwise alignment be adapted to reflect this additional biological insight, i.e. a better model of biology?

Yes, we introduce the concept of a gapcost-function $g(k)$ which gives the cost/penalty for a block of $k$ consecutive insertions or deletions ...

**Computational challenge**

Can we compute an optimal global alignment with general (affine) gapcost efficiently?

Yes, we introduce the concept of a gapcost-function $g(k)$ which gives the cost/penalty for a block of $k$ consecutive insertions or deletions ...

**Example**

<table>
<thead>
<tr>
<th>A</th>
<th>T</th>
<th>A</th>
<th>C</th>
<th>A</th>
<th>--</th>
<th>--</th>
<th>C</th>
<th>G</th>
<th>C</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>T</td>
<td>--</td>
<td>C</td>
<td>T</td>
<td>C</td>
<td>C</td>
<td>A</td>
<td>C</td>
<td>--</td>
<td>C</td>
</tr>
</tbody>
</table>

$$s(A,A) + s(T,T) + g(1) + s(C,C) + s(A,T) + g(3) + s(C,C) + g(1) + s(C,C) + s(A,T)$$
Global alignment of two strings

\[ \mathcal{A} = \begin{bmatrix} A & C & T & C & G \end{bmatrix} \]

General cost:

\[ \text{cost}(\mathcal{A}) = \sum_{\text{blocks}} \text{"cost of block"} \]

Substitution cost: \[ s : \Sigma \times \Sigma \rightarrow \mathbb{R} \]

Gap cost: \[ g : \mathbb{N}^+ \rightarrow \mathbb{R} \]
Computing optimal cost with general gap cost

Intuition

\[ S(i, j) = \max_{\text{last block}} \left\{ \text{"cost of last block" + "opt. cost of rest."} \right\} \]

Formalization

\[ S(i, j) = \max \begin{cases} 0, & M(i, j) \\ s(A[i], B[j]) + S(i-1, j-1) - D(i, j) \end{cases} \]

\[ \max_{0 \leq k \leq i} \left[ S(i-k, j) - g(k) \right] \]

\[ \max_{0 \leq k \leq j} \left[ S(i, j-k) - g(k) \right] \]
Computing optimal cost with general gap cost

Can be implemented using dynamic programming

Running time: \( \left( \frac{4}{9} \right)^2 \cdot \frac{n}{2} \leq T(n) \leq n^2 \cdot n \; \mathcal{O}(n^3) \)

Space consumption: \( \mathcal{O}(n^2) \)

\[
S(e, j) = \max \begin{cases} 
0 & \text{MC}(i, j) \\
S(e, j) + S(e-1, j-1) & \text{O}(i, j) \\
\max_{0 \leq h \leq i} \left[ S(e-h, j) - g(h) \right] & \text{I}(i, j) \\
\max_{0 \leq k \leq j-i} \left[ S(e, j-k) - g(k) \right] & \text{B}(j-i, j) 
\end{cases}
\]
Computing optimal cost with affine gap cost

\[ g(k) = \alpha \cdot k + \beta \ , \ \alpha, \beta > 0 \]

Intuition

... as general gap cost but reduce time to compute \( D(i, j) \) and \( I(i, j) \). Let us consider \( D(i, j) \)...

Trick

Consider the best deletion block, two possibilities:

\[
\begin{align*}
M(i-1,j) - (d+\beta) & \quad I(i-1,j) - (d+\beta) \\
\underbrace{\overbrace{AL(i-1)}_{=AL(i-1)} | \overbrace{AE}^{=AE}}_{=BE(j)} - SE(j) & \quad \overbrace{AL(i-1)}_{=AL(i-1)} | \overbrace{AE}^{=AE} - SE(j)
\end{align*}
\]

1. A new block

\[
D(i, j) - \alpha
\]

2. Continuation of existing block

\[
\underbrace{AL(i-1)}_{=AL(i-1)} | \overbrace{AE}^{=AE} - SE(j)
\]
**Computing optimal cost with affine gap cost**

Because

\[ S(i-1,j) = \max\{M(i-1,j), I(i-1,j), D(i-1,j)\} \]

and

\[ D(i-1,j)-(\alpha+\beta) \leq D(i-1,j)-\alpha \]

Hence,

\[ D(i,j) = \max \{ M(i-1,j)-(\alpha+\beta), I(i-1,j)-(\alpha+\beta), D(i-1,j)-\alpha \} \]

\[ = \max \{ S(i-1,j)-(\alpha+\beta), D(i-1,j)-\alpha \} \]

**Trick**

Consider the best deletion block, two possibilities:

1. A new block
   \[
   \begin{bmatrix}
   \text{AL}\ldots\text{AL} \\
   \text{BE}\ldots\text{BE}
   \end{bmatrix}
   \]

2. Continuation of existing block
   \[
   \begin{bmatrix}
   \text{AL}\ldots\text{AL} \\
   \text{BE}\ldots\text{BE}
   \end{bmatrix}
   \]

\[ D(i-1,j) - \alpha \]
Computing optimal cost with affine gap cost

\[
S(i, j) = \max \begin{cases} 
0 & \text{i = 0 and j = 0} \\
S(i-1, j-1) + s(A[i], B[j]) & \text{i > 0 and j > 0} \\
D(i, j) & \text{i > 0 and j >= 0} \\
I(i, j) & \text{i >= 0 and j >= 0} 
\end{cases}
\]

\[
D(i, j) = \max \begin{cases} 
S(i-1, j) - (e+\beta) & \text{i > 0 and j >= 0} \\
D(i-1, j) - \alpha & \text{i > 1 and j >= 0} 
\end{cases}
\]

\[
I(i, j) = \max \begin{cases} 
S(i, j-1) - (e+\beta) & \text{i >= 0 and j > 0} \\
I(i, j-1) - \alpha & \text{i >= 0 and j > 1} 
\end{cases}
\]

Can be implemented using dynamic programming/memorization using 3 tables of size \(O(n \cdot m)\)
Computing optimal cost with affine gap cost

\[ S(i, j) = \max \begin{cases} 
0 & i = 0 \text{ and } j = 0 \\
S(i-1, j-1) + s(A[i], B[j]) & i > 0 \text{ and } j > 0 \\
D(i, j) & i > 0 \text{ and } j \geq 0 \\
I(i, j) & i \geq 0 \text{ and } j \geq 0 
\end{cases} \]

\[ D(i, j) = \max \begin{cases} 
S(i-1, j) - (\alpha + \beta) & i > 0 \text{ and } j \geq 1 \\
D(i-1, j) - \alpha & i > 1 \text{ and } j \geq 0 
\end{cases} \]

\[ I(i, j) = \max \begin{cases} 
S(i, j-1) - (\alpha + \beta) & i \geq 0 \text{ and } j > 0 \\
I(i, j-1) - \alpha & i \geq 0 \text{ and } j \geq 1 
\end{cases} \]

Can be implemented using dynamic programming/memorization using 3 tables of size \( \Theta(n_m) \).
Computing optimal cost with affine gap cost

Time and space is $O(nm)$. An optimal alignment can be retrieved by backtracking in time $O(n)$.

\[
S(i,j) = \max \begin{cases} 
0 & \text{i = 0 and j = 0} \\
S(i-1,j-1) + \delta(A[i],B[j]) & \text{i > 0 and j > 0} \\
D(i,j) & \text{i > 0 and j >= 0} \\
I(i,j) & \text{i >= 0 and j >= 0} 
\end{cases}
\]

\[
D(i,j) = \max \begin{cases} 
S(i-1,j) - (d+\beta) & \text{i > 0 and j >= 0} \\
D(i-1,j) - \alpha & \text{i > 1 and j >= 0} 
\end{cases}
\]

\[
I(i,j) = \max \begin{cases} 
S(i,j-1) - (d+\beta) & \text{i >= 0 and j > 0} \\
I(i,j-1) - \alpha & \text{i >= 0 and j > 1} 
\end{cases}
\]

Can be implemented using dyn-prog/memorization using 3 tables of size $\Theta(nm)$. 
Backtracking an optimal alignment

```
func backtrack_iterative( A[1..n], B[1..m], S[0..n][..m] )
    // A and B are the input strings, S is the dynamic programming table, where S[i,j]
    // is the cost of an optimal alignment with affine gapcost of A[1..i] and B[1..m].
    i = n, j = m
    while ( i>0 or j>0 ):
        if (i > 0 and j > 0) and (S[i,j] == S[i-1, j-1] + subcost(A[i], B[j])) then
            // optimal alignment of A[1..i] and B[1..j] ends in a sub-column
            "output column (A[i], B[j])"
            i = i – 1
            j = j - 1
        else
            // optimal alignment of A[1..i] and B[1..j] ends in a del- or in-block
            k = 1
            while True:
                if (i >= k) and S[i,j] == S[i-k, j] - g(k) then
                    // optimal alignment of A[1..i] and B[1..j] ends in del-block of length k
                    "output columns (A[i] A[i-1] … A[i-k+1], -- … --)"
                    i = i – k
                    break // exit while-loop
                else if (j >= k) and S[i,j] == S[i, j-k] - g(k) then
                    // optimal alignment of A[1..i] and B[1..j] ends in an in-block of length k
                    j = j – k
                    break // exit while-loop
                else
                    k = k + 1
            endif
        endwhile
   endif
endfunc
```
Backtracking an optimal alignment

```python
func backtrack_iterative( A[1..n], B[1..m], S[0..n][..m] )

    // A and B are the input strings, S is the dynamic programming table, where S[i,j]
    // is the cost of an optimal alignment with affine gapcost of A[1..i] and B[1..m].
    i = n, j = m
    while ( i>0 or j>0 ):
        if (i > 0 and j > 0) and (S[i,j] == S[i-1, j-1] + subcost(A[i], B[j])) then
            // optimal alignment of A[1..i] and B[1..j] ends in a sub-column
            "output column (A[i], B[j])"
            i = i – 1
            j = j - 1
        else
            // optimal alignment of A[1..i] and B[1..j] ends in a del- or in-block
            k = 1
            while True:
                if (i >= k) and S[i,j] == S[i-k, j] - g(k) then
                    // optimal alignment of A[1..i] and B[1..j] ends in del-block of length k
                    "output columns (A[i] A[i-1] ... A[i-k+1], -- ... --)"
                    i = i – k
                    break // exit while-loop
                else if (j >= k) and S[i,j] == S[i, j-k] - g(k) then
                    // optimal alignment of A[1..i] and B[1..j] ends in a in-block of length k
                    "output columns (-- ... --, B[j]B[j-1] ... B[j-k+1])"
                    j = j – k
                    break // exit while-loop
            else
                k = k + 1
        endif
    endwhile
endfunc
```

Does it work?

Running time?
Backtracking an optimal alignment

```python
func backtrack_iterative( A[1..n], B[1..m], S[0..n][..m] )
    // A and B are the input strings, S is the dynamic programming table, where S[i,j]
    // is the cost of an optimal alignment with affine gapcost of A[1..i] and B[1..m].
    i = n, j = m
    while ( i>0 or j>0 ):
        if (i > 0 and j > 0) and (S[i,j] == S[i-1, j-1] + subcost(A[i], B[j])) then
            // optimal alignment of A[1..i] and B[1..j] ends in a sub-column
            "output column (A[i], B[j])"
            i = i – 1
            j = j - 1
        else
            // optimal alignment of A[1..i] and B[1..j] ends in a del- or in-block
            k = 1
            while True:
                if (i >= k) and S[i,j] == S[i-k, j] - g(k) then
                    // optimal alignment of A[1..i] and B[1..j] ends in del-block of length k
                    "output columns (A[i] A[i-1] ... A[i-k+1], -- ... --)"
                    i = i – k
                    break // exit while-loop
                else if (j >= k) and S[i,j] == S[i, j-k] - g(k) then
                    // optimal alignment of A[1..i] and B[1..j] ends in a in-block of length k
                    "output columns (-- ... --, B[j]B[j-1] ... B[j-k+1])"
                    j = j – k
                    break // exit while-loop
                else
                    k = k + 1

    endwhile
endfunc
```

Running time? O(n) because we only use O(k) time to backtrack a gap of length k (why?). Note that if our implementation use O(n) time to backtrack a gap of length k, the running time would be O(n²).
Global alignment with convex gap cost

\[ g(k) = \text{“a convex function, \( f_x \log(k) \) or \( \sqrt{k} \)”} \]

In cubic time using above recursion, but it can be done in \( O(nm\log(n)) \)