SEQUENCE COMPARISON

Andrzej Lingas
Lund University, Sweden

Examples of applications

1. Comparison of different results of sequencing the same gene
2. Fragment assembly in large scale DNA sequencing
3. Search for local similarities in large biosequence databases

Comparing Two Sequences

An example of sequence alignment

$$\begin{array}{ccccccccc} A & T & - & C & - & T & G & A & T \\ - & T & G & C & A & T & C & A & - \end{array}$$

An example of column scores

1. match (the same characters) +1
2. mismatch (different characters) -1
3. space -2

$$4 \times (-2) + 4 \times (+1) + 1 \times (-1) = -5$$

Similarity problem: Given two sequences, find the maximum of the scores of their alignment.

Optimal alignment problem: Given two sequences, find their optimal alignment, i.e., an alignment achieving the highest score.

A Dynamic Programming Method

Input: two sequences s and t
Output: the similarity of s and t, i.e., the score of their optimal alignment

$$SI(s[1..i], t[1..j]) = \max \begin{cases} SI(s[1..i], t[1..j-1]) - 2 \\ SI(s[1..i-1], t[1..j-1]) + p(i,j) \\ SI(s[1..i-1], t[1..j]) - 2 \end{cases}$$

where $p(i,j)$ is 1 if $s[i] = t[j]$ and -1 if $s[i] \neq t[j]$

Theorem 0.1 *The similarity between two sequences s and t can be computed in time $O(|s||t|)$.*

To compute an optimal alignment backtrack.

Theorem 0.2 *An optimal alignment between two sequences s and t can be computed in time $O(|s||t|)$.*

Procedure $Align(s[1..i],\ t[1..j])$

Output: alignment in $align_s$, $align_t$, and the length l of the alignment

if $s[1..i]$ **and** $t[1..j]$ are empty **then** $l \leftarrow 0$

else if $s[1..i]$ is not empty **and** $SI(s[1..i], t[1..j]) = SI(s[1..i], t[1..j-1]) - 2$ **then**

$Align(s[1..i], t[1..j-1], l);$

$l \leftarrow l + 1;$

$align_s[l] \leftarrow -;$

$align_t[l] \leftarrow t[j]$

else if both $s[1..i]$ and $t[1..j]$ are not empty **and** $SI(s[1..i], t[1..j]) = SI(s[1..i-1], t[1..j-1]) + p(i, j)$ **then**

$Align(s[1..i-1], t[1..j-1], l);$

$l \leftarrow l + 1;$

$align_s[l] \leftarrow s[i];$

$align_t[l] \leftarrow t[j]$

else comment($t[1..j]$ is not empty and $SI(s[1..i], t[1..j]) = SI(s[1..i-1], t[1..j]) - 2$)

$Align(s[1..i-1], t[1..j], l);$

$l \leftarrow l + 1;$

$align_s[l] \leftarrow s[i];$

$align_t[l] \leftarrow -$

Local Alignment

A **local alignment** between s and t is an alignment between a substring of s and a substring of t.

The **problem of optimal local alignment** for two sequences s and t is to find an alignment between a substring of s and a substring of t achieving the highest possible score.

Almost the same dynamic programming method

Input: two sequences s and t
Output: for each i, j, the highest score of an alignment between suffixes of $s[1..i]$ and $t[j]$

$$SI(s[1..i], t[1..j]) = \max \begin{cases} SI(s[1..i], t[1..j-1]) - 2 \\ SI(s[1..i-1], t[1..j-1]) + p(i,j) \\ SI(s[1..i-1], t[1..j]) - 2 \\ 0 \end{cases}$$

where $p(i,j)$ is 1 if $s[i] = t[j]$ and -1 if $s[i] \neq t[j]$

Theorem 0.3 *An optimal local alignment between two sequences s and t can be computed in time $O(|s||t|)$.*

Space efficient algorithm

Similarity between s and t can be easily computed in linear space and time $O(|s||t|)$ by implementing the algorithm for similarity in a row or column fashion.
If we proceed in a row fashion up to the i-th row then we obtain the similarities between $s[1..i]$ and all prefixes of t.

$$Optalign\begin{pmatrix} s[1..i-1] \\ t[1..j-1] \end{pmatrix} + \begin{matrix} s[i] \\ t[j] \end{matrix} + Optalign\begin{pmatrix} s[i+1..m] \\ t[j+1..n] \end{pmatrix}$$

$$Optalign\begin{pmatrix} s[1..i-1] \\ t[1..j] \end{pmatrix} + \begin{matrix} s[i] \\ - \end{matrix} + Optalign\begin{pmatrix} s[i+1..m] \\ t[j+1..n] \end{pmatrix}$$

Set i to $\lceil m/2 \rceil$.
Compute the similarities between $s[1..i-1]$ and prefixes of t and the similarities between $s[i+1..m]$ and suffixes of t. Next, pick j yielding a maximum total score.

Complexity analysis

The linear space complexity of the method follows immediately from that of computing similarities between $s[1..i-1]$ and prefixes of t and similarities between $s[i+1..m]$ and suffixes of t. Note that both groups of similarities and the optimal index j can be computed in time $(c/2) \times mn$ for some constant c.

For $m = |s|$ and $n = |t|$, let $T(m, n)$ be the time complexity of the method.
We prove $T(m, n) \leq cmn$ by induction on m.
For $m = 1$, the thesis is obvious if we pick enough large c.

$$T(m, n) \leq (c/2)mn + T(m/2, j) + t(m/2, n - j)$$

$$\leq c(m/2)n + c(m/2)j + c(m/2)(n - j) = cmn$$

Theorem 0.4 *An optimal (local) alignment between two sequences s and t can be computed in space $O(|s| + |t|)$ and time $O(|s||t|)$.*

Gap Penalty Functions

For $k > 1$, a *k-gap* is a sequence of k consecutive spaces. A k-gap is more likely than k isolated spaces. Let $w(k)$ be the penalty associated with a k-gap.

Computing similarities with respect to a general gap penalty by block dynamic programming.

Three kinds of blocks:

1. Two aligned characters different from -
2. A maximal series of consecutive characters in t aligned with spaces in s;
3. A maximal series of consecutive characters in s aligned with spaces in t.

$$\begin{array}{cccccccccccccccc} A & A & C & - & - & - & A & T & A & T & C & C & G & C & A & C \\ A & C & T & A & C & C & T & - & - & - & - & - & - & G & - & - \end{array}$$

$$\left| \begin{array}{c} A \\ A \end{array} \right| \begin{array}{c} A \\ C \end{array} \left| \begin{array}{c} C \\ T \end{array} \right| \begin{array}{c} \ldots\ldots\ldots \\ ACC \end{array} \left| \begin{array}{c} A \\ T \end{array} \right| \begin{array}{c} TATCCG \\ \ldots\ldots\ldots\ldots\ldots\ldots \end{array} \left| \begin{array}{c} C \\ G \end{array} \right| \begin{array}{c} AC \\ \ldots\ldots \end{array} \right|$$

The block method

Three arrays of size $(m+1) \times (n+1)$

Array a for alignments ending with character-character blocks

Array b for alignments ending with spaces in s

Array c for alignments ending with spaces in t

Initialization

$a[0,0] = 0$

$b[0,j] = -w(j)$

$c[i,0] = -w(i)$

All other entries where $i = 0$ or $j = 0$ are set to $-\infty$

The recurrence relations

$$a[i,j] = p(i,j) + \max \begin{cases} a[i-1, j-1] \\ b[i-1, j-1] \\ c[i-1, j-1] \end{cases}$$

$$b[i,j] = \max \begin{cases} a[i, j-k] - w(k), 1 \leq k \leq j \\ c[i, j-k] - w(k), 1 \leq k \leq j \end{cases}$$

$$c[i,j] = \max \begin{cases} a[i-k, j] - w(k), 1 \leq k \leq i \\ b[i-k, j] - w(k), 1 \leq k \leq i \end{cases}$$

$$S'(s,t) = \max\{a[m,n], b[m,n], c[m,n]\}$$

Time complexity

The time complexity of the block method is proportional to the number of times the entries of the arrays $a,\ b,\ c$ are accessed which is

$$\sum_{i=1}^{m}\sum_{j=1}^{n}(3+2j+2i) = O(mn^2+m^2n)$$

An optimal alignment under the general penalty function w can be obtained again by backtracking.

Theorem 0.5 *An optimal (local) alignment between two sequences s and t under a general penalty gap function can be computed in time $O(|s||t|^2+|s|^2|t|)$.*

Affine gap penalty function

A penalty function w which satisfies $w(k) = h + gk$ is called *affine*
We have also to assume $h, g \geq 0$ in order to keep the sub-additivity property $w(k_1 + k_2) \leq w(k_1) + w(k_2)$
Similarly as in the block method, we use the three arrays $a,\ b,\ c$ of size $(m+1) \times (n+1)$

Initialization

$$a[0,0] = 0,\ a[i,0] = -\infty, a[0,j] = -\infty,$$

$$b[i,0] = -\infty,\ b[0,j] = -(h+gj),$$

$$c[i,0] = -(h+gi),\ c[0,j] = -\infty.$$

The recurrence relations

$$a[i,j] = p(i,j) + \max \begin{cases} a[i-1,j-1] \\ b[i-1,j-1] \\ c[i-1,j-1] \end{cases}$$

$$b[i,j] = \max \begin{cases} -(h+g) + a[i,j-1] \\ -g + b[i,j-1] \\ -(h+g) + c[i,j-1] \end{cases}$$

$$c[i,j] = \max \begin{cases} -(h+g) + a[i-1,j] \\ -(h+g) + b[i-1,j] \\ -g + c[i-1,j] \end{cases}$$

Theorem 0.6 *An optimal alignment between two sequences s and t under a sub-additive affine penalty gap function can be computed in time $O(|s||t|)$.*

Homework 1

Design a polynomial-time algorithm for counting the number of optimal pairwise alignments.