Efficiency Matters: Introduction to Sequence Alignment and Dynamic Programming (DP)

May 21 2010

Sequence Comparison

- Sequence comparison is at the heart of many tasks in computational biology.

Sequence Comparison

- Sequence comparison is at the heart of many tasks in computational biology.
- We want to know how *similar* two sequences are.

Sequence Comparison

- Sequence comparison is at the heart of many tasks in computational biology.
- We want to know how *similar* two sequences are.
- The *definition* of similarity should reflect biological (usually evolutionary) events, such as mutations and rearrangements, that cause related sequences to differ.

Sequence Comparison

- Sequence comparison is at the heart of many tasks in computational biology.
- We want to know how *similar* two sequences are.
- The *definition* of similarity should reflect biological (usually evolutionary) events, such as mutations and rearrangements, that cause related sequences to differ.
- Here we introduce the *simplest* definition of similarity, to illustrate how to *compute* a measure of similarity. This definition only crudely reflects real biology.

Sequence Comparison

- Sequence comparison is at the heart of many tasks in computational biology.
- We want to know how *similar* two sequences are.
- The *definition* of similarity should reflect biological (usually evolutionary) events, such as mutations and rearrangements, that cause related sequences to differ.
- Here we introduce the *simplest* definition of similarity, to illustrate how to *compute* a measure of similarity. This definition only crudely reflects real biology.
- The computational technique extends to more complex and realistic biological definitions of similarity.

Sequence Alignment

Definition: Given two sequences S_1 and S_2, an *alignment* of S_1 and S_2 is obtained by inserting spaces into, or before or after the ends of, S_1 and S_2, so that the resulting two strings S_1' and S_2' have the *same number* of characters (a space is considered a character).

Sequence Alignment

Definition: Given two sequences S_1 and S_2, an *alignment* of S_1 and S_2 is obtained by inserting spaces into, or before or after the ends of, S_1 and S_2, so that the resulting two strings S_1' and S_2' have the *same number* of characters (a space is considered a character).
Example:

S_1 =TACTAGGCATGAC
S_2 =ACAGGTCAGTC

S_1' =TACTAGG–CATGAC
S_2' =–AC–AGGTCA–GTC

An alignment of S_1, S_2 imposes well-defined positions, and some *aligned* pairs of characters from S_1, S_2.

Sequence Alignment

Definition: Given two sequences S_1 and S_2, an *alignment* of S_1 and S_2 is obtained by inserting spaces into, or before or after the ends of, S_1 and S_2, so that the resulting two strings S_1' and S_2' have the *same number* of characters (a space is considered a character).
Example:

S_1 =TACTAGGCATGAC
S_2 =ACAGGTCAGTC

S_1' =TACTAGG–CATGAC
S_2' =–AC–AGGTCA–GTC

An alignment of S_1, S_2 imposes well-defined positions, and some *aligned* pairs of characters from S_1, S_2.
In this example, there are ten aligned pairs of characters from S_1, S_2. Nine of those pairs consist of identical characters.

Maximum Common Subsequence

As one possible measure of the similarity of two sequences, we want an alignment that *Maximizes* the number of aligned (non-space) characters that are identical. That number is taken as a measure of the similarity of the two sequences. The alignment with that number of identical aligned characters is called optimal.

Maximum Common Subsequence

As one possible measure of the similarity of two sequences, we want an alignment that *Maximizes* the number of aligned (non-space) characters that are identical. That number is taken as a measure of the similarity of the two sequences. The alignment with that number of identical aligned characters is called optimal.

This is a simple, very crude, model of the biological similarity of two sequences. A richer model would better reflect the evolutionary history causing the divergence of the sequences.

Maximum Common Subsequence

As one possible measure of the similarity of two sequences, we want an alignment that *Maximizes* the number of aligned (non-space) characters that are identical. That number is taken as a measure of the similarity of the two sequences. The alignment with that number of identical aligned characters is called optimal.

This is a simple, very crude, model of the biological similarity of two sequences. A richer model would better reflect the evolutionary history causing the divergence of the sequences.

How can an optimal alignment be computed efficiently?

Maximum Common Subsequence

As one possible measure of the similarity of two sequences, we want an alignment that *Maximizes* the number of aligned (non-space) characters that are identical. That number is taken as a measure of the similarity of the two sequences. The alignment with that number of identical aligned characters is called optimal.

This is a simple, very crude, model of the biological similarity of two sequences. A richer model would better reflect the evolutionary history causing the divergence of the sequences.

How can an optimal alignment be computed efficiently?

Is an exhaustive examination of all possible alignments practical? How many alignments are there?

Time needed to examine each alignment

Assume that we can enumerate and examine one Billion alignments per second. Then the time required would be:

seq. length	no. alignments	secs	years	ages of universe
30	10^{22}	10^{13}	10^{6}	?
50	1.5×10^{37}	10^{28}	10^{19}	10^{9}
100	2×10^{75}	10^{66}	10^{59}	10^{50}
200	5×10^{150}	10^{141}	10^{137}	10^{127}
300	1.5×10^{228}	10^{219}	10^{213}	10^{203}

Clearly, exhaustive examination of all possible alignments is not practical, even for sequences of length 30. And, sometimes we want to align sequences that are vastly larger.

What to do: First, Think Recursively!

Recursive thinking requires that we describe the problem we want to solve in terms of smaller instances of the same problem. To do that, we first need the right notation.

For a string S, we define $S(k)$ as the character at position k of S, and $S[1..k]$ as the prefix of S consisting of the first k characters of S.

Think Recursively

We define $V(i, j)$ to be the maximum *number* of *identical* aligned characters in *any* alignment of substrings $S_1[1..i]$ and $S_2[1..j]$. Schematically:

$S_1[1..i]$ - - - - - - - - - - - - - i

$S_2[1..j]$ - - - - - - - - - - - - - - - - j

Ultimately, want to compute $V(n, m)$, where n and m are the lengths of S_1 and S_2.

Think Recursively!

To think about $V(i, j)$ recursively, we focus on the characters i of S_1 and j of S_2 and ask where these characters appear in the optimal alignment A of $S_1[1..i]$ and $S_2[1..j]$. There are three cases:

Case 1: Characters $S_1(i), S_2(j)$ align to each other in A.

Case 2: $S_1(i)$ appears to the left of $S_2(j)$ in A.

Case 3: $S_1(i)$ appears to the right of $S_2(j)$ in A.

You should convince yourselves that these three cases cover all the possibilities.

Case 1 in detail

When character $S_1(i)$ aligns with $S_2(j)$

$S_1[1..i]$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	i
$S_2[1..j]$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	j

Either characters $S(i)$ and $S(j)$ are identical or not.

If $S(i)$ and $S(j)$ are identical, then

$$V(i,j) = 1+??$$

Case 1 in detail

When character $S_1(i)$ aligns with $S_2(j)$

$S_1[1..i]$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	i
$S_2[1..j]$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	j

Either characters $S(i)$ and $S(j)$ are identical or not.

If $S(i)$ and $S(j)$ are identical, then

$$V(i,j) = 1+??$$

Comment: Here I wait for classroom participation - Hope for the baroque, "over-answered" reply

Case1 in detail

The point I want to emphasize is that *we* (the recursion designer) don't know how much will be added. That will be determined at execution time by the computer. But we know how to name what will be added.

Case1 in detail

The point I want to emphasize is that *we* (the recursion designer) don't know how much will be added. That will be determined at execution time by the computer. But we know how to name what will be added.
It is named

$$V(i-1, j-1)$$

Case1 in detail

The point I want to emphasize is that *we* (the recursion designer) don't know how much will be added. That will be determined at execution time by the computer. But we know how to name what will be added.
It is named

$$V(i-1, j-1)$$

So

$$V(i,j) = 1 + V(i-1, j-1) \text{ when } S_1(i) = S_2(j)$$

When characters $S_1(i)$ and $S_2(j)$ align

If $S_1(i)$ and $S_2(j)$ align to each other, but are not identical, then

$$V(i,j) = ??$$

When characters $S_1(i)$ and $S_2(j)$ align

If $S_1(i)$ and $S_2(j)$ align to each other, but are not identical, then

$$V(i,j) = ??$$

$V(i,j) = V(i-1,j-1)$ when $S_1(i) \neq S_2(j)$

Case 1 in detail

Recap: When character $S_1(i)$ aligns with $S_2(j)$

Then:

$$V(i,j) = 1 + V(i-1, j-1) \text{ if } S_1(i) = S_2(j)$$

and

$$V(i,j) = V(i-1, j-1) \text{ if } S_1(i) \neq S_2(j)$$

Cases 2 and 3 in detail

When $S_1(i)$ appears to the left of $S_2(j)$, then $S_2(j)$ must appear opposite a space in A. In that case,

$$V(i,j) = V(i,j-1).$$

Symmetrically, when $S_1(i)$ appears to the right of $S_2(j)$, then $S_1(i)$ must appear opposite a space in A. In that case,

$$V(i,j) = V(i-1,j).$$

Putting it all together

Putting the three cases together, the general recurrences are

$V(i,j)$ equals the Maximum of:

$1 + V(i-1, j-1)$ (if $S_1(i) = S_2(j)$)

$V(i-1, j-1)$ (if $S_1(i) \neq S_2(j)$)

$V(i, j-1)$.

$V(i-1, j)$.

Putting it all together

Putting the three cases together, the general recurrences are

$V(i,j)$ equals the Maximum of:

$1 + V(i-1, j-1)$ (if $S_1(i) = S_2(j)$)

$V(i-1, j-1)$ (if $S_1(i) \neq S_2(j)$)

$V(i, j-1)$.

$V(i-1, j)$.

We also need the Base Case:
$V(i, 0) = V(0, j) = 0$ for any i and j.

Evaluating the recurrences

At this point we have correct recurrences to express $V(i, j)$ in terms of smaller problem instances, and we can easily encode the recurrences in a program. The program would start with a call to compute $V(n, m)$, which would then recursively compute smaller instances. But this top-down use of the recurrences would be very inefficient, due to redundant recursive *calls*.

Instead, we use the recurrences in a bottom-up manner. We don't make explicit recursive calls to determine V values, but only do look-ups of previously computed V values.

First we take care of all of the Base-Cases, $V(i,0) = V(0,j) = 0$.

At this point, note that the value of $V(1,1)$ can be determined.

That is, the general recurrence for $V(i,j)$ requires that we have available $V(i-1,j-1)$, $V(i-1,j)$ and $V(i,j-1)$. For $i = j = 1$, we do have those values, so $V(1,1)$ can be determined, using the recurrences.

Evaluating the recurrences bottom up

After setting the value for $V(1,1)$, we can set the value for $V(1,2)$ or for $V(2,1)$.

Extending this observation, we can set the all the values $V(1,j)$ for increasing values of j, or set all the values of $V(i,1)$ for increasing values of i.

Then we can determine the values for $V(i,2)$ or $V(2,j)$ etc.

Finally, when we have determined $V(n,m)$, we have the value of the optimal alignment, but not the actual alignment itself.

Row-wise or Colum-wise evaluation of V - This is Dynamic Programming!

How efficient is the Dynamic Programming approach

We have $(n+1) \times (m+1)$ values of V that have to be determined. Using the recurrences, and evaluating V values by DP (not by recurrsive calls), each V value can be determined with a constant number of operations.

So the total number of operatations is $O(nm)$. Vastly, Vastly faster than exhaustive enumeration of all alignments, and vastly faster than a top-down use of the recurrences.

The traceback to actually determine the optimal alignment. That is for the next lecture.