COS 597c: Topics in Computational Molecular Biology
Lecture 2: September 22, 1999
Lecturer: Mona Singh
Scribe: Robert Phillips [1]

Sequence Comparison: Global Sequence Alignments

Introduction

In comparisons of biomolecular sequences (i.e., those of DNA, RNA, and protein), regions of high sequence similarity often indicate significant functional or structural similarity as well. The same and related molecular structures and mechanisms are reused and modified during evolution, and thus show up repeatedly within either a single genome or across the genomes of a wide variety of species. As a result, sequence comparison is the most commonly used method for inferring structure and biological function. Of course, sequences can have similar structure and function without exhibiting sequence similarity.

Sequence comparison is also the first step for many problems in computational biology, including fragment assembly, evolutionary tree reconstruction and genome analysis.

We will describe a method for detecting sequence similarity based on dynamic programming. Dynamic programming is a very useful technique that has many applications in computer science. Moreover, dynamic programming is used in other areas of computational biology as well, including in: prediction of RNA secondary structure; hidden Markov model applications (which are used for building profiles, for example); and homology-based methods for gene finding.

Before we can proceed further, we must make some definitions.

- *alphabet* - set of allowable symbols. Examples of biosequence alphabets:
  Σ={A,C,G,T} (DNA)
  Σ={A,C,G,U} (RNA)
  Σ={A,C,D,E,F,G,H,I,K,L,M,N,P,Q,R,S,T,V,W,Y} (proteins)

[1] This lecture is adapted from one given by Bonnie Berger at MIT, and is based on the text of Setubal and Meidanis [2]. Scribe notes are adapted from notes taken by Janet Marques and Russell Schwartz, also at MIT.

- *sequence* (*or string*) - finite succession of characters chosen from an alphabet
  e.g., `ATCCGAACTTG` from the DNA alphabet Σ=`{A,C,G,T}`

- *subsequence* - sequence obtained from a sequence by removal of characters
  e.g., `TTT` is a subsequence of `ATATAT`
  `AAAA` is not a subsequence of `ATATAT`

- *substring* - subsequence of *consecutive* characters
  e.g., `TAC` is a substring of `ACTACA`
  `TAC` is not a substring of `ATGAC`

- *prefix* - substring containing the first character of a string, including the empty string
  e.g., in the string `ACT`, a prefix may be the empty string, `A`, `AC`, or `ACT`

The Alignment Problem

Given two strings of different length, an *alignment* makes these sequences the same length through the insertion of gaps. Gaps may be added anywhere within the sequences, including at the beginning or the end, but cannot be aligned together.

As an example, take two strings, `CGACCTA` and `CGCCTA`. One alignment is as follows:

```
CGACCTA
CG-CCTA
```

Here we inserted a space (or gap) to generate a good alignment.

The global alignment problem is to find the "best" alignment according to some scoring function that measures similarity. A simple scoring function follows; later, we'll talk about where more realistic scoring functions for protein sequences come from.

Column	Score
Match	+1
Mismatch	-1
Gap	-2

Table 1: A simple scoring function

In this simple scoring function, a column containing two identical characters (a match) would receive a score of 1, a column containing two different characters (mismatch)

would receive a score of -1, and a column containing a gap would receive a score of -2. (Typically, for protein sequences, the scoring functions are symmetric 20 by 20 substitution matrices, where there is a similarity measure for each pair of amino acids.)

Using the simple scoring function, in the following example the total score is: $1+1-2-1+1+1=1$.

C	G	A	C	C	T
C	G	-	T	C	T
1	1	-2	-1	1	1

How do we find the best global alignment (i.e., the one with the highest score)? Let's first consider a brute-force approach:
1. Enumerate all possible alignments.
2. Score each alignment.
3. Choose the alignment with the highest score.

The problem with this approach is that the number of possible alignments is prohibitively large (exponential in the length of the sequences).

The Basic Algorithm

Fortunately, there exists an algorithm that computes the best alignment in $O(mn)$ time, where m and n are the lengths of the two sequences.[2]

We will first demonstrate the algorithm through an example. Suppose we want to know the score of the best alignment of $s =$ `AAAC` and $t =$ `AGC` using our simple scoring function.

Notation: Let $s(i)$ and $t(i)$ denote the ith and jth characters of s and t, respectively.

Considering just the last column of the alignment, we have only three possibilities:

- The last character of s (`C`) is aligned with the last character of t (`C`). In this case, the score of the best alignment of s and t is equal to the the score of the best alignment of `AAA` (the remaining portion of s) and `AG` (the remaining portion of t), plus 1 for matching the last character.

[2]Roughly speaking, we evaluate running time by examining the growth rate of higher order terms, ignoring constants of proportionality and any lower order terms. For example, $100n^2 + 100n + 100$ and $10000n^2 + 5n + 100$ are both $O(n^2)$.

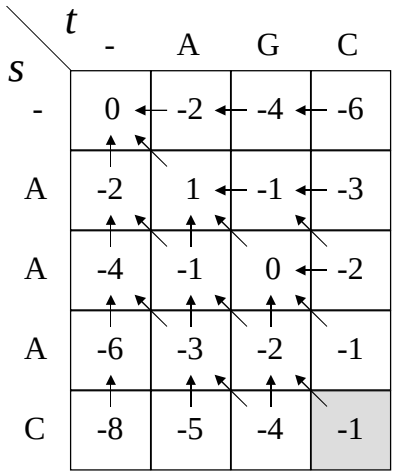

Figure 1: Matrix *sim* for scoring alignments of s=`AAAC` and t=`AGC`. The lower right element ($sim(4,3)$) is the most important; it contains the score of the best alignment.

- The last character of s (`C`) is aligned with a gap. In this case, the score of the best alignment of s and t is equal to the the score of the best alignment of `AAA` (the remaining portion of s) and `AGC` (all of t), minus 2 for inserting a gap.
- The last letter of t (`C`) is aligned with a gap. In this case, the score of the best alignment of s and t is equal to the the score of the best alignment of `AAAC` (all of s) and `AG` (the remaining portion of t), minus 2 for inserting a gap.

If we know the answers to the three subproblems mentioned above, then we will know the score of the best alignment between s and t. Note that the subproblems consist of aligning prefixes of s and t. We will find and save optimal solutions for all prefixes of s and t, building up from shorter ones to longer ones. There are five prefixes for s: empty, `A`, `AA`, `AAA`, and `AAAC`, and we will refer to these prefixes as the 0th, 1st, 2nd, 3rd and 4th prefixes of s. Likewise there are four prefixes for t: empty, `A`, `AG`, and `AGC`. The algorithm uses a *matrix representation* (in this case a 5×4 matrix), with characters of s along the rows and characters of t along the columns (shown in the Figure 1). We will define $sim(i, j)$ to correspond to the optimal alignment score (the "similarity") of the ith prefix of s with the jth prefix of t. Thus, the matrix reflects the similarity scores for all prefixes of s and t.

Looking at the matrix, $sim(0, 3)$ contains the score of the best alignment of the empty string (0th prefix of s) with `AGC` (3rd prefix of t), $sim(2, 2)$ contains the best alignment

score for `AA` (2nd prefix of s) and `AG` (2nd prefix of t). The element $sim(4,3)$ contains what we're looking for: the best alignment score for `AAAC` and `AGC`. In general, when we are aligning sequence s of length m and sequence t of length n, $sim(m,n)$ has the answer we're looking for, and we will fill out the entire matrix in order to get this last score. Each element is determined by our sim function, which takes the following form (again derived by considering what happens in the last column of the alignment of the ith prefix of s and the jth prefix of t):

$$sim(i,j) = max \begin{cases} sim(i-1,j-1) \pm 1, & \text{align } s(i) \text{ with } t(j), \\ & \quad +1 \text{ for a match, -1 for mismatch} \\ sim(i-1,j) - 2, & \text{align } s(i) \text{ with a gap} \\ sim(i,j-1) - 2, & \text{align } t(j) \text{ with a gap} \end{cases}$$

In our array then, in order to compute $sim(i,j)$, we need to have three entries pre-computed: $sim(i-1,j-1)$, $sim(i-1,j)$, and $sim(i,j-1)$. If we compute entries row by row left to right, we will always have things computed when we need them.

We start by filling out the the 0^{th} row and column: using our scoring function, an alignment of a single-letter string with a gap results in a score of -2; similarly, alignment of a two-letter string with a gap results in a score of -4. In general, the alignment of a string of i letters with a gap gives a score of $-2i$, and the 0^{th} row and column may thus be filled in accordingly (see Figure 1).

Now we have all the information we need to evaluate array element (1,1): $sim(1,1)$ is the alignment of `A` and `A` according to the function:

$$sim(1,1) = max \begin{cases} sim(0,0) \pm 1, & \text{align } s(1) \text{ with } t(1), \ (= 0 + 1 = 1) \\ sim(0,1) - 2, & \text{align } s(1) \text{ with a gap } (= -2 - 2 = -4) \\ sim(1,0) - 2, & \text{align } t(1) \text{ with a gap } (= -2 - 2 = -4) \end{cases}$$

The maximum is found at $sim(0,0)$ and evaluates to 1. We place 1 in our array element and, since the maximum came from element (0,0), we keep track of this (by "drawing" an arrow pointing to that array element). See Figure 1. We now have the information required to evaluate $sim(1,2)$ in the same manner:

$$sim(1,2) = max \begin{cases} sim(0,1) \pm 1, & \text{align } s(1) \text{ with } t(2), \ (= -2 - 1 = -3) \\ sim(0,2) - 2, & \text{align } s(1) \text{ with a gap } (= -4 - 2 = -6) \\ sim(1,1) - 2, & \text{align } t(2) \text{ with a gap } (= 1 - 2 = -1) \end{cases}$$

The maximum is -1 (fill this value in on the matrix) and it came from (1,1), so draw an arrow pointing to the left, toward element (1,1). Continuing with this process

we obtain the full matrix (with arrows) depicted in the previous figure. The final alignment score is in element (4,3) and is -1. But how do we reconstruct the alignment itself? This is where the arrows come in. Start with the final array element and follow the arrows back. An arrow from (i, j) pointing to element $(i-1, j-1)$ (to the diagonally upper left) means to align $s(i)$ and $t(j)$ with each other. An arrow pointing upward to $(i-1, j)$ means to align $s(i)$ with a gap, and an arrow pointing to the left to $(i, j-1)$ means to align $t(j)$ with a gap. Continuing through the three possible arrow paths, we are able to build three possible alignments (with the same scores):

```
AAAC
-AGC

AAAC
A-GC

AAAC
AG-C
```

The algorithm described takes $O(mn)$ time and $O(mn)$ space. There is also a space-saving version of the algorithm that takes $O(m+n)$ space, but still works in $O(mn)$ time. See the textbook of Setubal and Meidanis [2] for a description of the space-saving trick.

References

[1] Needleman, S. B. and Wunsch, C. D. (1970) A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.* 48:443–453.

[2] Setubal, J. and Meidanis, J. *Introduction to Computational Molecular Biology.* PWS Publishing Company, 1997.