String Alignment

CSE 521: Design and Analysis of Algorithms
April 8, 1998

1 Introduction to Sequence Similarity

These notes deal with the topic of "sequence similarity", where the sequence under consideration might be a DNA, RNA, or amino acid sequence. Our interest is predicated on the assumption that a high degree of similarity between two sequences *usually* implies similar function and/or structure. Most of the content is from a forthcoming book by Dan Gusfield on strings and trees in computer science and biology.

We start with two motivating case studies in which sequence similarity was utilized.

1.1 Hypothesizing the function of a new sequence

In 1995, Fleischmann *et al.* were the first to succeed in finding the DNA sequence for an entire genome of a free-living organism, when they announced the complete sequence for the bacterium *Haemophilus influenza rd.* The researchers identified 1743 regions of this sequence likely to be sites of genes (a regions of DNA that encodes the amino acid sequence of some protein), and set out to discover those genes' functions.

In order to hypothesize the functions of the genes, the researchers translated the coding regions into their corresponding amino acid sequences, and searched for similar sequences in a protein database. This search identified 1007 close matches. Because the database contained both sequenced proteins and their functions, these close matches allowed them to make conjectures about the function of each matched gene. These conjectures could then be tested in the laboratory.

1.2 Researching the effects of multiple sclerosis

Multiple sclerosis is an autoimmune disease in which the immune system attacks nerve cells in the patient. More specifically, the immune system's T-cells, which normally identify foreign bodies for immune system attacks, mistakenly identify proteins in the nerves' myelin sheaths as foreign.

It was conjectured that the myelin sheath proteins identified by the T-cells were similar to viral and/or bacterial sheath proteins from an earlier infection. In order to test this hypothesis, the following steps were carried out:

- the sequence of amino acids comprising the myelin sheath proteins were determined,

- a protein database was searched for similar bacterial and viral sequences, and
- laboratory tests were performed to determine if the T-cells attacked these same proteins.

The result was the identification of certain bacterial and viral proteins that were confused with the myelin sheath proteins. Hopefully further study of this confusion can shed light on the common features of the proteins involved that cause the T-cells to misidentify the myelin sheath proteins.

2 The String Alignment Problem

We now introduce the problem of string similarity. Given two strings `acbcdb` and `cadbd`, how should we measure how similar they are? Similarity is witnessed by finding a good "alignment" between two strings. Here is one possible alignment of these two strings.

a	c	-	-	b	c	d	b
-	c	a	d	b	-	d	-

The special character "-" represents the insertion of a space. We can evaluate the goodness of such an alignment using a scoring function. For example, if an exact match between two characters scores +2, and every mismatch or insertion of a space scores −1, then the alignment above has score

$$3 \cdot (2) + 5 \cdot (-1) = 1.$$

This example shows only one possible alignment for the given strings and scoring function. For any pair of strings and scoring function, there are many possible alignments.

The following definitions generalize this example.

Definition: If x and y are each a single character or space, then $\sigma(x, y)$ denotes the **score** of aligning x and y.

In the example above, for any two distinct characters a and c, $\sigma(c, c) = +2$, and $\sigma(c, a) = \sigma(c, -) = \sigma(-, c) = -1$.

Definition: If S is a string, then $|S|$ denotes the length of S and $S[i]$ denotes the ith character of S (where the first character is $S[1]$ rather than, say, $S[0]$).

For example, if $S =$ `acbcdb`, then $|S| = 6$ and $S[3] =$ `b`.

Definition: Let S and T be strings. An **alignment** A maps S and T into strings S' and T' that may contain space characters, where

1. $|S'| = |T'|$, and

2. the removal of all spaces from S' and T' leaves S and T, respectively.

The **value** of the alignment A is

$$\sum_{i=1}^{l} \sigma\left(S'[i], T'[i]\right),$$

where $l = |S'| = |T'|$.

In the example alignment above, if $S =$ `acbcdb` and $T =$ `cadbd`, then $S' =$ `ac--bcdb` and $T' =$ `-cadb-d-`.

Definition: An **optimal alignment** of S and T is one that has the maximum possible value for these two strings.

For the two strings given in the example above, is the alignment shown optimal? We will next present an algorithm for computing optimal alignments, which will allow us to answer that question.

3 Computing an Optimal Alignment by Dynamic Programming

Given strings S and T, with $|S| = n$ and $|T| = m$, our goal is to compute an optimal alignment of S and T. To reach this goal, define $V(i, j)$ as the value of an optimal alignment of the strings $S[1] \cdots S[i]$ and $T[1] \cdots T[j]$.

The value of an optimal alignment of S and T is then $V(n, m)$. The crux of dynamic programming is to solve the more general problems of computing *all* values $V(i, j)$ with $0 \leq i \leq n$ and $0 \leq j \leq m$, in order of increasing i and j. Each of these will be relatively simple to compute, given the values already computed for smaller i and/or j, using a recurrence relation. To start the process, we need a basis for $i = 0$ and/or $j = 0$.

BASIS :

$$\begin{aligned} V(i,0) &= \sum_{k=1}^{i} \sigma(S[k], -) \\ V(0,j) &= \sum_{k=1}^{j} \sigma(-, T[k]) \end{aligned}$$

The basis for $V(i, 0)$ says that if i characters of S are to be aligned with 0 characters of T, then they must be matched with spaces. The basis for $V(0, j)$ is analogous.

$$
\begin{array}{llll}
V(i,j) = \max(& V(i-1,j-1) & + \ \sigma(S[i],T[j]) & , \\
 & V(i-1,j) & + \ \sigma(S[i],-) & , \\
 & V(i,j-1) & + \ \sigma(-,T[j]) &)
\end{array}
$$

This formula can be understood by considering an optimal alignment of the first i characters from S and the first j characters from T. In particular, consider the last pair of characters in such an alignment. This last pair must be one of the following:

1. $(S[i], T[j])$, in which case the alignment to the left of this pair must be an optimal alignment of $S[1]\cdots S[i-1]$ and $T[1]\cdots T[j-1]$ (i.e., must have value $V(i-1, j-1)$), or
2. $(S[i], -)$, in which case the alignment to the left must have value $V(i-1, j)$, or
3. $(-, T[j])$, in which case the alignment to the left must have value $V(i, j-1)$.

The optimal alignment chooses the one among these three possibilities with the greatest value.

3.1 Example

In aligning `acbcdb` and `cadbd`, the dynamic programming algorithm fills in the following values for $V(i,j)$ from top to bottom and left to right, simply applying the basis and recurrence formulas. (Recall from the example of Section 2 that matches score $+2$, mismatches and spaces -1.) For instance, in the table below, the entry in row 4 and column 1 is obtained by computing $\max(-3+2, 0-1, -4-1) = -1$.

	j	0	1	2	3	4	5
i			c	a	d	b	d
0		0	-1	-2	-3	-4	-5
1	a	-1	-1	1	0	-1	-2
2	c	-2	1	0	0	-1	-2
3	b	-3	0	0	-1	2	1
4	c	-4	-1	-1	-1	1	1
5	d	-5	-2	-2	1	0	3
6	b	-6	-3	-3	0	3	2

The value of the optimal alignment is $V(n, m)$, and so can be read from the entry in the last row and last column. Thus, there is an alignment of `acbcdb` and `cadbd` that has value 2, so the alignment proposed in Section 2 with value 1 is not optimal. But how can we determine the optimal alignment itself, and not just its value?

3.2 Recovering the Alignments

The solution is to retrace our steps back from the (n, m) entry, determining which preceding entries were responsible for the current one. For instance, in the table below, the (4,2) entry could have followed from either the (3,1) or (3,2) entry; this is denoted by the two arrows pointing to those entries. We can then follow any of these paths from (n, m) to $(0, 0)$, tracing out an optimal alignment:

	j	0	1	2	3	4	5
i			c	a	d	b	d
0		0	$\leftarrow -1$	-2	-3	-4	-5
1	a	$\uparrow -1$	-1	$\nwarrow$ 1	0	-1	-2
2	c	-2	$\nwarrow$ 1	0	$\nwarrow$ 0	-1	-2
3	b	-3	$\uparrow$ 0	$\nwarrow$ 0	-1	$\nwarrow$ 2	1
4	c	-4	-1	$\nwarrow\uparrow -1$	-1	$\uparrow$ 1	1
5	d	-5	-2	-2	$\nwarrow$ 1	0	$\nwarrow$ 3
6	b	-6	-3	-3	0	$\nwarrow$ 3	$\hookleftarrow\uparrow$ 2

The optimal alignments corresponding to these three paths are

a	c	b	c	d	b	-
-	c	-	a	d	b	d

,

a	c	b	c	d	b	-
-	c	a	-	d	b	d

, and

-	a	c	b	c	d	b
c	a	d	b	-	d	-

.

Each of these has three matches, one mismatch, and three spaces, for a value of $3 \cdot (2) + 4 \cdot (-1) = 2$, the optimal alignment value.

3.3 Time Analysis

Theorem 1 *The dynamic programming algorithm computes an optimal alignment in time $O(nm)$.*

Proof: This algorithm requires an $(n + 1) \times (m + 1)$ table to be completed. Any particular entry is computed with a maximum of 6 table lookups, 3 additions, and a three-way maximum, that is, in time c, a constant. Thus, the complexity of the algorithm is at most $c(n+1)(m+1) = O(nm)$. Reconstructing a single alignment can then be done in time $O(n + m)$. $\square$