Lecture 3

Introduction to Sequence Similarity

January 11, 2000
Notes: Martin Tompa

3.1. Sequence Similarity

The next few lectures will deal with the topic of "sequence similarity", where the sequences under consideration might be DNA, RNA, or amino acid sequences. This is likely the most frequently performed task in computational biology. Its usefulness is predicated on the assumption that a high degree of similarity between two sequences often implies similar function and/or three-dimensional structure. Most of the content of these lectures on sequence similarity is from Gusfield [1].

Why are we starting here, rather than with a discussion of how biologists determine the sequence in the first place? The reason is that the problems and algorithms of sequence similarity are reasonably simple to state. This makes it a good context in which to ensure that we agree on the language we will be using to discuss computing and algorithms.

To begin that process, the word *algorithm* simply means an unambiguously specified method for solving a problem. In this context, an algorithm may be thought of as a computer program, although algorithms are usually expressed in a somewhat more abstract language than real programming languages.

3.2. Biological Motivation for Studying Sequence Similarity

We start with two motivating applications in which sequence similarity is utilized.

3.2.1. Hypothesizing the Function of a New Sequence

When a new genome is sequenced, the usual first analysis performed is to identify the genes and hypothesize their functions. Hypothesizing their functions is most often done using sequence similarity algorithms, as follows. One first translates the coding regions into their corresponding amino acid sequences, using the genetic code of Table 1.1. One then searches for similar sequences in a protein database that contains sequenced proteins (from related organisms) and their functions. Close matches allow one to make strong conjectures about the function of each matched gene. In a similar way, sequence similarity can be used to predict the three-dimensional structure of a newly sequenced protein, given a database of known protein sequences and their structures.

3.2.2. 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 myelin sheath proteins were sequenced,
- 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.

3.3. The String Alignment Problem

The first task is to make the problem of sequence similarity more precise. A *string* is a sequence of characters from some alphabet. 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*, representing a deletion from its sequence (or, equivalently, an insertion in the other sequence). 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 deletion (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. For any pair of strings, there are many possible alignments.

The following definitions generalize this example.

Definition 3.1: If x and y are each a single character or space, then $\sigma(x, y)$ denotes the *score* of aligning x and y. σ is called the *scoring function*.

In the example above, for any two distinct characters a and c, $\sigma(c, c) = +2$, and $\sigma(c, a) = \sigma(c, -) = \sigma(-, c) = -1$. If one were designing a scoring function for comparing amino acid sequences, one would certainly want to incorporate into it the physico-chemical similarities and differences among the amino acids, such as those described in Section 1.1.1.

Definition 3.2: 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 $S = \texttt{acbcdb}$, then $|S| = 6$ and $S[3] = \texttt{b}$.

Definition 3.3: 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 spaces from S' and T' (without changing the order of the remaining characters) 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 $S = \texttt{acbcdb}$ and $T = \texttt{cadbd}$, then $S' = \texttt{ac--bcdb}$ and $T' = \texttt{-cadb-d-}$.

Definition 3.4: An *optimal alignment* of S and T is one that has the maximum possible value for these two strings.

Finding an optimal alignment of S and T is the way in which we will measure their similarity. For the two strings given in the example above, is the alignment shown optimal? We will next present some algorithms for computing optimal alignments, which will allow us to answer that question.

3.4. An Obvious Algorithm for Optimal Alignment

The most obvious algorithm is to try all possible alignments, and output any alignment with maximum value. We will examine this approach in more detail.

A *subsequence* of a string S means a sequence of characters of S that need not be consecutive in S, but do retain their order as given in S. For instance, $\texttt{acd}$ is a subsequence of $\texttt{acbcdb}$.

Suppose we are given strings S and T, and assume for the moment that $|S| = |T| = n$. Also, consider an arbitrary scoring function $\sigma(x, y)$, subject to the reasonable restriction that $\sigma(-,-) \leq 0$. With this restriction, there is never a reason to align a pair of spaces.

The obvious algorithm for optimal alignment is given in Figure 3.1. This algorithm works correctly, but is it a good algorithm? If you tried running this algorithm on a pair of strings each of length 20 (which is ridiculously modest by biology standards), you would find it much too slow to be practical. The program would run for an hour on such inputs, even if the computer can perform a billion basic operations per second.

```
for all $i$, $0 \leq i \leq n$, do
    for all subsequences $A$ of $S$ with $|A| = i$ do
        for all subsequences $B$ of $T$ with $|B| = i$ do
            Form an alignment that matches $A[k]$ with $B[k]$, $1 \leq k \leq i$,
                and matches all other characters with spaces;
            Determine the value of this alignment;
            Retain the alignment with maximum value;
        end ;
    end ;
end ;
```

Figure 3.1: Enumerating all Alignments to Find the Optimal

The running time analysis of this algorithm proceeds as follows. A string of length n has $\binom{n}{i}$ subsequences of length i. [1] Thus, there are $\binom{n}{i}^2$ pairs (A, B) of subsequences each of length i. Consider one such pair. Since there are n characters in S, only i of which are matched with characters in T, there will be $n - i$ characters in S unmatched to characters in T. Thus, the alignment has length $n + (n - i) = 2n - i$. We must look up and add the score of each pair in the alignment, so the total number of basic operations is at least

$$\sum_{i=0}^{n} \binom{n}{i}^2 (2n - i) \geq n \sum_{i=0}^{n} \binom{n}{i}^2 = n \binom{2n}{n} > 2^{2n}, \text{ for } n > 3.$$

(The equality has a pretty combinatorial explanation that is worth discovering. The last inequality follows from Stirling's approximation [2].) Thus, for $n = 20$, this algorithm requires more than $2^{2n} = 2^{40}$ basic operations.

3.5. Asymptotic Analysis of Algorithms

In Section 4.1 we will see a cleverer algorithm that runs in time proportional to n^2. For large n, it is clear that 2^{2n} is greater than n^2. As a demonstration that an algorithm that requires time proportional to n^2 is far more desirable than one that requires time 2^{2n}, consider at what value of n these two functions cross. Suppose the actual running time of the cleverer algorithm is $100n^2 + 100n + 100$. The value of 2^{2n} already exceeds this quadratic at $n = 7$. Suppose instead that the running time is $10,000n^2 + 100n + 100$. Despite the fact that we increased the constant of proportionality by a factor of 100, 2^{2n} already exceeds this quadratic at $n = 10$. This demonstration should make it clear that the rate of growth of the high order term is the most important determinant of how long an algorithm runs on large inputs, independent of the constant of proportionality and any lower order terms.

To formalize this notion, we introduce "big O" notation.

Definition 3.5: Let $f(n)$ and $g(n)$ be functions. Then $f(n) = O(g(n))$ if and only if there is a constant c such that, for all n sufficiently large, $|f(n)| \leq cg(n)$.

[1] The notation $\binom{n}{i}$ denotes the number of combinations of n distinguishable objects taken i at a time. See any textbook on combinatorial mathematics, for instance Roberts [2].

For example, $100n^2 + 100n + 100$ and $10,000n^2 + 100n + 100$ are both $O(n^2)$. For the former, $c = 101$ works, and for the latter, $c = 10,001$ works.

References

[1] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.

[2] F. S. Roberts. *Applied Combinatorics*. Prentice-Hall, 1984.