Lecture 6

Multiple Sequence Alignment

January 20, 2000
Notes: Martin Tompa

While previous lectures discussed the problem of determining the similarity between two strings, this lecture turns to the problem of determining the similarity among multiple strings.

6.1. Biological Motivation for Multiple Sequence Alignment

6.1.1. Representing Protein Families

An important motivation for studying the similarity among multiple strings is the fact that protein databases are often categorized by protein families. A *protein family* is a collection of proteins with similar structure (i.e., three-dimensional shape), similar function, or similar evolutionary history. When we have a newly sequenced protein, we would like to know to which family it belongs, as this provides hypotheses about its structure, function, or evolutionary history. (See Section 3.2.1.) The new protein might not be particularly similar to a single protein in the database, yet might still share considerable similarity with the collective members of a family of proteins. One approach is to construct a representation for each protein family, for example a good multiple sequence alignment of all its members. Then, when we have a newly sequenced protein and want to find its family, we only have to compare it to the representation of each family.

Common structure, function, or origin of a molecule may only be weakly reflected in its sequence. For example, the three-dimensional structure of a protein is very difficult to infer from its sequence, and yet is very important to predict its function. Multiple sequence comparisons may help highlight weak sequence similarity, and shed light on structure, function, or origin.

6.1.2. Repetitive Sequences in DNA

In the DNA domain, a motivation for multiple sequence alignment arises in the study of *repetitive sequences*. These are sequences of DNA, often without clearly understood biological function, that are repeated many times throughout the genome. The repetitions are generally not exact, but differ from each other in a small number of insertions, deletions, and substitutions. As an example, the *Alu* repeat is approximately 300 bp long, and appears over 600,000 times in the human genome. It is believed that as much as 60% of the human genome may be attributable to repetitive sequences without known biological function. (See Jurka and Batzer [5].)

In order to highlight the similarities and differences among the instances of such a repeat family, one would like to display a good multiple sequence alignment of its constituent sequences.

6.2. Formulation of the Multiple String Alignment Problem

We now define the problem more precisely.

Definition 6.1: Given strings $S_1, S_2, \ldots, S_k$ a *multiple (global) alignment* maps them to strings $S'_1, S'_2, \ldots, S'_k$ that may contain spaces, where

1. $|S'_1| = |S'_2| = \cdots = |S'_k|$, and
2. the removal of spaces from S'_i leaves S_i, for $1 \leq i \leq k$.

The question that arises next is how to assign a value to such an alignment. In a pairwise alignment, we simply summed the similarity score of corresponding characters. In the case of multiple string alignment, there are various scoring methods, and controversy around the question of which is best. We focus here on a scoring method called the "sum-of-pairs" score. Other methods are explored in the homework.

Until now, we have been using a scoring function that assigns higher values to better alignments and lower values to worse alignments, and we have been trying to find alignments with maximum value. For the remainder of this lecture, we will switch to a function $\delta(x, y)$ that measures the *distance* between characters x and y. That is, it will assign higher values the more distant two strings are. In the case of two strings, we will thus be trying to *minimize*

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

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

Definition 6.2: The *sum-of-pairs (SP) value* for a multiple global alignment A of k strings is the sum of the values of all $\binom{k}{2}$ pairwise alignments induced by A.

In this definition we assume that the scoring function is symmetric. For simplicity, we will not discuss the issue of a separate gap penalty.

Example 6.3: Consider the following alignment:

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

Using the distance function $\delta(x, x) = 0$, and $\delta(x, y) = 1$ for $x \neq y$, this alignment has a sum-of-pairs value $3 + 5 + 4 = 12$.

Definition 6.4: An *optimal SP (global) alignment* of strings $S_1, S_2, \ldots, S_k$ is an alignment that has the minimum possible sum-of-pairs value for these k strings.

6.3. Computing an Optimal Multiple Alignment by Dynamic Programming

Given k strings each of length n, there is a generalization of the dynamic programming algorithm of Section 4.1 that finds an optimal SP alignment. Instead of a 2-dimensional table, it fills in a k-dimensional table. This table has dimensions

$$\underbrace{(n+1) \times (n+1) \cdots \times (n+1)}_{k},$$

that is, $(n+1)^k$ entries. Each entry depends on $2^k - 1$ adjacent entries, corresponding to the possibilities for the last match in an optimal alignment: any of the 2^k subsets of the k strings could participate in that match, except for the empty subset. The details of the algorithm itself and the recurrence are left as exercises for the reader.

Because each of the $(n+1)^k$ entries can be computed in time proportional to 2^k, the running time of the algorithm is $O((2n)^k)$. If $n \approx 350$ (as is typical for the length of proteins), it would be practical only for very small values of k, perhaps 3 or 4. However, typical protein families have hundreds of members, so this algorithm is of no use in the motivational problem posed in Section 6.1. We would like an algorithm that works for k in the hundreds too, which would be possible only if the running time were polynomial in both n and k. (In particular, k should not appear in the exponent as it does in the expression $(2n)^k$.) Unfortunately, we are very unlikely to find such an algorithm, which is a consequence of the following theorem:

Theorem 6.5 (Wang and Jiang [7]): The optimal SP alignment problem is *NP*-complete.

What *NP*-completeness means and what its consequences are will be discussed in the following section.

6.4. *NP*-completeness

In this section we give a brief introduction to *NP*-completeness, and how problems can be proved to be *NP*-complete.

Definition 6.6: A problem has a *polynomial time solution* if and only if there is some algorithm that solves it in time $O(n^c)$, where c is a constant and n is the size of the input.

Many familiar computational problems have polynomial time solutions:

1. two-string optimal alignment problem: $O(n^2)$ (Theorem 4.1) ,
2. sorting: $O(n \log n)$ [2],
3. two-string alignment with arbitrary gap penalty function: $O(n^3)$ (Section 5.3.1),
4. 100-string optimal alignment problem: $O(n^{100})$ (Section 6.3).

The last entry illustrates that having a polynomial time solution does not mean that the algorithm is practical. In most cases the converse, though, is true: an algorithm whose running time is not polynomial is likely to be impractical for all but the smallest size inputs.

NP-complete problems are equivalent in the sense that if any one of them has a polynomial time solution, then all of them do. One of the major open questions in computer science is whether there is a polynomial

time solution for any of the *NP*-complete problems. Almost all experts conjecture strongly that the answer to this question is "no". The bulk of the evidence supporting this conjecture, however, is only the failure to find such a polynomial time solution in thirty years.

In 1971, Cook defined the notion of *NP*-completeness and showed the *NP*-completeness of a small collection of problems, most of them from the domain of mathematical logic [1]. Roughly speaking, he defined *NP*-complete problems to be problems that have the property that we can verify in polynomial time whether a supplied solution is correct. For instance, if you did not have to *compute* an optimal SP alignment, but simply had to *verify* that a given alignment had SP value at most x, a given integer, it would be easy to write a polynomial time algorithm to do so.

Shortly after Cook's work, Karp recognized the wide applicability of the concept of *NP*-completeness. He showed that a diverse host of problems are each *NP*-complete [6]. Since then, many hundreds of natural problems from many areas of computer science and mathematics such as graph theory, combinatorial optimization, scheduling, and symbolic computation have been proven *NP*-complete; see Garey and Johnson [3] for details.

Proving a problem Q to be *NP*-complete proceeds in the following way. Choose a known *NP*-complete problem A. Show that A has a polynomial time algorithm if it is allowed to invoke a polynomial time subroutine for Q, and vice versa.

There are many computational biology problems that are *NP*-complete, yet in practice we still need to solve them somehow. There are different ways to deal with an *NP*-complete problem:

1. We might give up on the possibility of solving the problem on anything but small inputs, by using an exhaustive (nonpolynomial time) search algorithm. We can sometimes use dynamic programming or branch-and-bound techniques to cut down the running time of such a brute force exhaustive search.

2. We might give up guaranteed efficiency by settling for an algorithm that is sufficiently efficient on inputs that arise in practice, but is nonpolynomial on some worst-case inputs that (hopefully) do not arise in practice. There may be an algorithm that runs in polynomial time on average inputs, being careful to define the input distribution so that the practical inputs are highly probable.

3. We might give up guaranteed optimality of solution quality by settling for an approximate algorithm that gives a suboptimal solution, especially if the suboptimal solution is provably not much worse than the optimal solution. (An example is given in Section 6.5.)

4. Heuristics (local search, simulated annealing, "genetic" algorithms, and many others) can also be used to improve the quality of solution or running time in practice. We will see several examples throughout the remaining lectures. However, rigorous analysis of heuristic algorithms is generally unavailable.

5. The problem to be solved in practice may be more specialized than the general one that was proved *NP*-complete.

In the following section we will look at the approximation approach to find a solution for the multiple string alignment problem.

6.5. An Approximation Algorithm for Multiple String Alignment

In this section we will show that there is a polynomial time algorithm (called the *Center Star Alignment Algorithm*) that produces multiple string alignments whose SP values are less than twice that of the optimal solutions. This result is due to Gusfield [4]. Although the factor of 2 may be unacceptable in some applications, the result will serve to illustrate how approximation algorithms work.

In this section we will make the following assumptions about the distance function:

1. $\delta(x, x) = 0$, for all characters x.
2. Triangle Inequality: $\delta(x, z) \leq \delta(x, y) + \delta(y, z)$, for all characters x, y, and z, and

The triangle inequality says that the distance along one edge of a triangle is at most the sum of the distances along the other two edges. Although intuitively plausible, be aware that not all distance measures used in biology obey the triangle inequality.

6.5.1. Algorithm

Definition 6.7: For strings S and T, define $D(S, T)$ to be the value of the minimum (global) alignment distance of S and T.

The approximation algorithm is as follows. The input is a set $\mathcal{T}$ of k strings. First find $S_1 \in \mathcal{T}$ that minimizes

$$\sum_{S \in \mathcal{T} - \{S_1\}} D(S_1, S).$$

This can be done by running the dynamic programming algorithm of Section 4.1 on each of the $\binom{k}{2}$ pairs of strings in $\mathcal{T}$. Call the remaining strings in $\mathcal{T}$ $S_2, \ldots, S_k$. Add these strings $S_2, \ldots, S_k$ one at a time to a multiple alignment that initially contains only S_1, as follows.

Suppose $S_1, S_2, \ldots, S_{i-1}$ are already aligned as $S_1', S_2', \ldots, S_{i-1}'$. To add S_i, run the dynamic programming algorithm of Section 4.1 on S_1' and S_i to produce S_1'' and S_i'. Adjust $S_2', \ldots, S_{i-1}'$ by adding spaces to those columns where spaces were added to get S_1'' from S_1'. Replace S_1' by S_1''.

6.5.2. Time Analysis

Theorem 6.8: The approximation algorithm of Section 6.5.1 runs in time $O(k^2 n^2)$ when given k strings each of length at most n.

Proof: By Theorem 4.1, each of the $\binom{k}{2}$ values $D(S, T)$ required to compute S_1 can be computed in time $O(n^2)$, so the total time for this portion is $O(\binom{k}{2} n^2) = O(k^2 n^2)$. After adding S_i to the multiple string alignment, the length of S_1' is at most in, so the time to add all n strings to the multiple string alignment is

$$\sum_{i=1}^{k-1} O((in) \cdot n) = O(k^2 n^2).$$

□

6.5.3. Error Analysis

What remains to be shown is that the algorithm produces a solution that is less than a factor of 2 worse than the optimal solution. Let M be the alignment produced by this algorithm, let $d(i,j)$ be the distance M induces on the pair S_i, S_j, and let

$$v(M) = \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} d(i,j).$$

Note that $v(M)$ is exactly twice the SP score of M, since every pair of strings is counted twice.

Then $d(1,l) = D(S_1, S_l)$ for all l. This is because the algorithm used an optimal alignment of S'_1 and S_l, and $D(S'_1, S_l) = D(S_1, S_l)$, since $\delta(-,-) = 0$. If the algorithm later adds spaces to both S'_1 and S'_l, it does so in the same columns.

Let M^* be the optimal alignment, $d^*(i,j)$ be the distance M^* induces on the pair S_i, S_j, and

$$v(M^*) = \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} d^*(i,j).$$

Theorem 6.9: $\frac{v(M)}{v(M^*)} \leq \frac{2(k-1)}{k} < 2$. That is, the algorithm of Section 6.5.1 produces an alignment whose SP value is less than twice that of the optimal SP alignment.

Proof: We will derive an upper bound on $v(M)$ and a lower bound on $v(M^*)$, and then take their quotient.

$$\begin{aligned}
v(M) &= \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} d(i,j) \\
&\leq \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} (d(i,1) + d(1,j)) && \text{(triangle inequality)} \\
&= 2(k-1) \sum_{l=2}^{k} d(1,l) && \text{(explained below)} \\
&= 2(k-1) \sum_{l=2}^{k} D(S_1, S_l)
\end{aligned}$$

The third line follows because each $d(l,1) = d(1,l)$ occurs in $2(k-1)$ terms of the second line.

$$\begin{aligned}
v(M^*) &= \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} d^*(i,j) \\
&\geq \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} D(S_i, S_j) && \text{(Definition 6.7)}
\end{aligned}$$

$$\geq \sum_{i=1}^{k}\sum_{j=2}^{k} D(S_1, S_j) \qquad \text{(definition of } S_1\text{)}$$

$$= k\sum_{l=2}^{k} D(S_1, S_l)$$

Combining these inequalities,

$$\frac{v(M)}{v(M^*)} \leq \frac{2(k-1)}{k} < 2.$$

□

Note that for small values of k, the approximation is significantly better than a factor of 2. Furthermore, the error analysis does not mean that the approximation solution is always $2(k-1)/k$ times the optimal solution. It means that the quality of the solution is never worse than this, and may be better in practice.

6.5.4. Other Approaches

In the Center Star Algorithm discussed in Section 6.5.1, we always try to align the chosen center string S_1 with the unaligned strings. However, there might be cases in which some of the strings are very "near" to each other and form "clusters". It might be an advantage to align strings in the same cluster first, and then merge the clusters of strings. The problem with this is how to define "near" and how to define "clusters".

There are many variants on this idea, which sometimes are called *iterative pairwise alignment* methods. Here is one version: an unaligned string nearest to any aligned string is picked and aligned with the previously aligned group. (For those who have seen it before, note the similarity to Prim's minimum spanning tree algorithm [2].) The "nearest" string is chosen based on optimal pairwise alignments between individual strings in the multiple alignment and unaligned strings, without regard to spaces inserted in the multiple alignment. Now the problem is to specify how to align a string with a *group* of strings. One possible method is to mimic the technique that was used to add S_i to the center star alignment in Section 6.5.1.

6.6. The Consensus String

Given a multiple string alignment, it is sometimes useful to derive from it a "consensus string" that can be used to represent the entire set of strings in the alignment.

Definition 6.10: Given a multiple alignment M of strings $S_1, S_2, \ldots, S_k$, the *consensus character* of column i of M is the character c_i that minimizes the sum of distances to it from all the characters in column i; that is, it minimizes $\sum_{j=1}^{k} \delta(S'_j[i], c_i)$. Let $d(i)$ be this minimum sum. The *consensus string* is the concatenation $c_1 c_2 \cdots c_l$ of all the consensus characters, where $l = |S'_1| = \cdots = |S'_k|$. The *alignment error* of M is then defined to be $\sum_{i=1}^{l} d(i)$.

For instance, the consensus string for the multiple string alignment in Example 6.3 is `ac-cdbd`, and its alignment error is 6, the number of characters in the aligned strings that differ from the consensus character in the corresponding position.

6.7. Summary

Multiple sequence alignment is a very important problem in computational biology. It appears to be impossible to obtain exact solutions in polynomial time, even with very simple scoring functions. A variety of (provably) bounded approximation algorithms are known, and a number of heuristic algorithms have been suggested, but it still remains largely an open problem.

References

[1] S. A. Cook. The complexity of theorem proving procedures. In *Conference Record of Third Annual ACM Symposium on Theory of Computing*, pages 151–158, Shaker Heights, OH, May 1971.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[3] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[4] D. Gusfield. Efficient methods for multiple sequence alignment with guaranteed error bounds. *Bulletin of Mathematical Biology*, 55:141–154, 1993.

[5] J. Jurka and M. A. Batzer. Human repetitive elements. In R. A. Meyers, editor, *Encyclopedia of Molecular Biology and Molecular Medicine*, volume 3, pages 240–246. Weinheim, Germany, 1996.

[6] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–104. Plenum Press, New York, 1972.

[7] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1:337–348, 1994.