Informatica per le biotecnologie

8. Confronto tra sequenze - 2

Estratto dal testo di J. Setubal e J. Meidanis:
Introduction to Computational Molecular Biology

NOTE. Dopo una breve motivazione sulla rilevanza degli algoritmi di confronto tra sequenze in biologia molecolare, riportiamo due problemi più avanzati rispetto a quelli della dispensa 7 che utilizzano lo stesso schema algoritmico e gli stessi pesi della global comparison.

Si noti solo che qui la matrice *M* si chiama *a*, e che *g* indica il peso dell'allineamento tra un carattere e uno spazio.

CHAPTER 3

SEQUENCE COMPARISON AND DATABASE SEARCH

In this chapter we present some of the most practical and widely used methods for sequence comparison and database search. Sequence comparison is undoubtedly the most basic operation in computational biology. It appears explicitly or implicitly in all subsequent chapters of this book. It has also many applications in other subareas of computer science.

3.1 BIOLOGICAL BACKGROUND

Why compare sequences? Why use a computer to do that? How to compare sequences using computers? These are the main questions that will concern us in this chapter. We will answer the first two in this section, and devote the rest of the chapter to the last one.

Sequence comparison is the most important primitive operation in computational biology, serving as a basis for many other, more complex, manipulations. Roughly speaking, this operation consists of finding which parts of the sequences are alike and which parts differ. However, behind this apparently simple concept, a great variety of truly distinct problems exist, with diverse formalizations and sometimes requiring completely different data structures and algorithms for an efficient solution.

As examples, we will give a list of problems that often appear in computational biology. In these examples we use two notions that will be precisely defined in later sections. One is the *similarity* of two sequences, which gives a measure of how similar the sequences are. The other is the *alignment* of two sequences, which is a way of placing one sequence above the other in order to make clear the correspondence between similar characters or substrings from the sequences. The examples (and the whole chapter) also use basic concepts about strings, which are defined in Section 2.1. Here are the examples.

1. We have two sequences over the same alphabet, both about the same length (tens of thousands of characters). We know that the sequences are almost equal, with

only a few isolated differences such as insertions, deletions, and substitutions of characters. The average frequency of these differences is low, say, one each hundred characters. We want to find the places where the differences occur.

2. We have two sequences over the same alphabet with a few hundred characters each. We want to know whether there is a prefix of one which is similar to a suffix of the other. If the answer is yes, the prefix and the suffix involved must be produced.
3. We have the same problem as in (2), but now we have several hundred sequences that must be compared (each one against all). In addition, we know that the great majority of sequence pairs are unrelated, that is, they will not have the required degree of similarity.
4. We have two sequences over the same alphabet with a few hundred characters each. We want to know whether there are two substrings, one from each sequence, that are similar.
5. We have the same problem as in (4), but instead of two sequences we have one sequence that must be compared to thousands of others.

Problems like (1) appear when, for instance, the same gene is sequenced by two different labs and they want to compare the results; or even when the same long sequence is typed twice into the computer and we are looking for typing errors. Problems like (2) and (3) appear in the context of fragment assembly in programs to help large-scale DNA sequencing. Problems like (4) and (5) occur in the context of searches for local similarities using large biosequence databases.

We will see in this chapter that a single basic algorithmic idea can be used to solve all of the above problems. However, this may not be the most efficient solution. Sometimes less general but faster methods are better suited to each task.

3.3.4 COMPARING SIMILAR SEQUENCES

Two sequences are similar when they "look alike" in some sense. We have already developed our intuition about the concept of similarity and have formalized it through alignments and scores. So, for us, two sequences are similar when the scores of the optimal alignments between them are very close to the maximum possible (where "maximum possible" will be made precise). This section is about faster algorithms to find good alignments in this case. We analyze global alignments only.

Let us first treat the case where the two sequences of interest, s and t, have the same length n. This is a fair assumption, given that we are focusing on similar sequences. If s and t have the same length, the dynamic programming matrix is a square matrix and its main diagonal runs from position $(0, 0)$ to (n, n). Following this diagonal corresponds to the unique alignment without spaces between s and t. If this is not an optimal alignment, we need to insert some spaces in the sequences to obtain a better score. Notice that spaces will always be inserted in pairs, one in s and one in t.

As we insert these space pairs, the alignment is thrown off the main diagonal. Consider for a moment the following two sequences:

$$s = \text{GCGCATGGATTGAGCGA}$$
$$t = \text{TGCGCCATGGATGAGCA}.$$

The optimal alignments correspond to paths that reach up to the diagonal twice removed from the main one, as in Figure 3.8. The best alignments include two pairs of spaces. In this case, the number of space pairs is equal to how far the alignment departed from the main diagonal, but this does not always happen. One thing is certain, though: The number of space pairs is greater than or equal to the maximum departure.

The basic idea then goes as follows. If the sequences are similar, the best alignments have their paths near the main diagonal. To compute the optimal score and alignments, it is not necessary to fill in the entire matrix. A narrow band around the main diagonal should suffice.

The algorithm *KBand* in Figure 3.9 performs the matrix fill-in in a band of horizontal (or vertical) width $2k + 1$ around the main diagonal. At the end, the entry $a[n, n]$ contains the highest score of an alignment confined to that band. This algorithm runs in time $O(kn)$, which is a big win over the usual $O(n^2)$ if k is small compared to n.


```
-GCGC-ATGGATTGAGCGA
TGCGCCATGGAT-GAGC-A
```

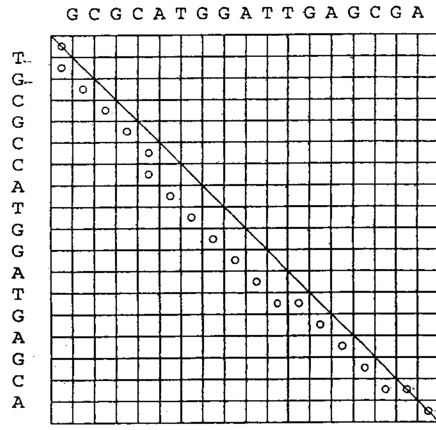

FIGURE 3.8

An optimal alignment and its corresponding path in the dynamic programming matrix. A line is drawn along the main diagonal.

Observe that we do not use entries outside of the k-strip at all. We do not initialize them, and we do not consult them for maximum computations. To test whether a certain position (i, j) is inside the k-strip, we use the following criterion:

$$\mathit{InsideStrip}(i, j, k) = (-k \le i - j \le k).$$

As in the other dynamic programming algorithms, each entry $a[i, j]$ depends on $a[i-1, j]$, $a[i-1, j-1]$, and $a[i, j-1]$. In the code, we need not test position $(i-1, j-1)$, because it is in the same diagonal as (i, j), so it will always be inside the strip. Tests must be performed for $(i-1, j)$ and $(i, j-1)$, which may be outside the strip when (i, j) is in the border.

How can we use algorithm *KBand*? We choose a value for k and run the algorithm. If $a[n, n]$ is greater than or equal to the best score that could possibly come from an alignment with $k+1$ or more space pairs, we are lucky: We have found an optimal alignment with just $O(kn)$ steps. The best possible score, given that we have at least $k+1$ space pairs, is

$$M(n - k - 1) + 2(k + 1)g, \tag{3.9}$$

which is computed assuming that there are exactly $k + 1$ space pairs and that the other pairs are matches. Here M is the score of a match, and g is added for each space. We will assume that $M > 0$ and $g \le 0$.

```
Algorithm KBand
    input: sequences s and t of equal length n, integer k
    output: best score of alignment at most k diagonals away from main diagonal
    n ← |s|
    for i ← 0 to k do
        a[i, 0] ← i × g
    for j ← 0 to k do
        a[0, j] ← j × g
    for i ← 1 to n do
        for d ← −k to k do
            j ← i + d
            if 1 ≤ j ≤ n do then
                // compute maximum among predecessors
                a[i, j] ← a[i − 1, j − 1] + p(i, j)
                if InsideStrip(i − 1, j, k) then
                    a[i, j] ← max(a[i, j], a[i − 1, j] + g)
                if InsideStrip(i, j − 1, k) then
                    a[i, j] ← max(a[i, j], a[i, j − 1] + g)
    return a[n, n]
```

FIGURE 3.9

Algorithm for k-strip around main diagonal.

If $a[n, n]$ is smaller than the quantity (3.9), we double k and run the algorithm again. If the initial value of k is 1, further values will be powers of two. The stop condition becomes

$$a_k[n, n] \geq M(n - k - 1) + 2(k + 1)g,$$

which is equivalent to

$$k \geq \frac{Mn - a_k[n, n]}{M - 2g} - 1.$$

At this point, we have already run the algorithm many times, each time with a larger k, so that the total complexity is

$$n + 2n + 4n + \cdots + kn \leq 2kn,$$

assuming we use powers of two. To bound this total complexity, we need an upper bound on k. Now, we did not stop earlier, so

$$\frac{k}{2} < \frac{Mn - a_{k/2}[n, n]}{M - 2g} - 1.$$

If $a_k[n, n] = a_{k/2}[n, n]$, then this is the optimum score $\mathrm{sim}(s, t)$, and our bound is

$$k < 2\left(\frac{Mn - \mathrm{sim}(s, t)}{M - 2g} - 1\right).$$

If $a_k[n, n] > a_{k/2}[n, n]$, then optimal alignments have more than $k/2$ pairs of spaces,

and therefore

$$\mathrm{sim}(s, t) \leq M\left(n - \frac{k}{2} - 1\right) + 2\left(\frac{k}{2} + 1\right)g,$$

which leads to

$$k \leq 2\left(\frac{Mn - \mathrm{sim}(s, t)}{M - 2g} - 1\right),$$

almost the same bound as above.

Observe that $M - 2g$ is a constant. The time complexity is then $O(dn)$, where d is the difference between the maximum possible score Mn — the score of two identical sequences — and the optimal score. Thus, the higher the similarity, the faster the answer.

It is straightforward to extend this method to general sequences, not necessarily with the same length. Space-saving versions can also be easily derived.

3.4 COMPARING MULTIPLE SEQUENCES

So far in this chapter we have concentrated on the comparison between a pair of sequences. However we often are given several sequences that we have to align simultaneously in the best possible way. This happens, for example, when we have the sequences for certain proteins that have similar function in a number of different species. We may want to know which parts of these sequences are similar and which parts are different. To get this information, we need to build a **multiple alignment** for these sequences, and that is the topic of this section.

The notion of multiple alignment is a natural generalization of the two-sequence case. Let $s_1, \ldots, s_k$ be a set of sequences over the same alphabet. A multiple alignment involving $s_1, \ldots, s_k$ is obtained by inserting spaces in the sequences in such a way as to make them all of the same size. It is customary to place the extended sequences in a vertical list so that characters — or spaces — in corresponding positions occupy the same column. We further require that no column be made exclusively of spaces. Figure 3.10 shows a multiple alignment involving four short amino acid sequences. (Multiple alignments are more common with proteins, so in some of our examples in this section we use sequences of amino acids.)

One important issue to decide in multiple alignment is the precise definition of the

```
MQPILLL
MLR-LL-
MK-ILLL
MPPVLIL
```

FIGURE 3.10

Multiple alignment of four amino acid sequences

quality of an alignment. We will next study one way of scoring a multiple alignment based on pairwise alignments. In addition, scientists also look at multiple alignments by placing the sequences in a tree structure rather than piling them up. This leads to different measures of quality that we discuss in subsequent sections.

3.4.1 THE SP MEASURE

Scoring a multiple alignment is more complex than its pairwise counterpart. We restrict ourselves to purely additive functions here; that is, the alignment score is the sum of column scores. Therefore we need a way to assign a score to each column and then add them up to get the alignment score. However, to score a column, we want a function with k arguments, where k is the number of sequences. Each one of these arguments is a character or a space. One way to do that would be to have a k-dimensional array that could be indexed with the arguments and return the value. The problem with this approach is that it is necessary to specify a value for each possible combination of arguments, and there could be as many as $2^k - 1$ such combinations. A typical value for k is 10, and that results in more than 1000 possibilities. Some more manageable methods to define such a function are necessary.

Such methods can be obtained by determining "reasonable" properties that such a function should have. First, the function must be independent of the order of arguments. For instance, if a column has `I, -, I, V` and another has `V, I, I, -`, they should both receive the same score. Second, it should be a function that rewards the presence of many equal or strongly related residues and penalizes unrelated residues and spaces. A solution that satisfies these properties is the so-called **sum-of-pairs** (SP) function. It is defined as the sum of pairwise scores of all pairs of symbols in the column. For instance, the score of a column with the above content is

$$SP\text{-}score(\mathtt{I}, -, \mathtt{I}, \mathtt{V}) = p(\mathtt{I}, -) + p(\mathtt{I}, \mathtt{I}) + p(\mathtt{I}, \mathtt{V}) + p(-, \mathtt{I}) + p(-, \mathtt{V}) + p(\mathtt{I}, \mathtt{V}),$$

where $p(a, b)$ is the pairwise score of symbols a and b. Notice that this may include a space penalty specification when either a or b is a space. This is a very convenient scheme, because it relies on pairwise scores like the ones we use for two-sequence comparison. The SP scoring system is widely used due to its simplicity and effectiveness.

A small but important detail needs to be addressed to complete the definition. Although no column can be composed exclusively by spaces, it is possible to have two or more spaces in a given column. When computing the SP score, we need a value for $p(-, -)$. This value is not specified in two-sequence comparison because it never appears there. However, it is necessary for SP-based multiple comparison. The common practice is to set $p(-, -) = 0$. This may seem strange, given that spaces are generally penalized (i.e., when there is a space, the pairwise score is negative), so two spaces should be even more so. Nevertheless, there are good reasons to define $p(-, -)$ as zero. One of them is related to pairwise alignments again. We often draw conclusions about a multiple alignment by looking at the pairwise alignments it induces. Indeed, in any multiple alignment, we may select two of the sequences and just look at the way they are aligned to each other, forgetting about all the rest. It is not difficult to see that this produces a pairwise

alignment, except for the fact that we may have columns with two spaces in them. But then we just remove these columns and derive a true pairwise alignment. An example of this procedure is presented in Figure 3.11. This is what we call the *induced* pairwise alignment, or the *projection* of a multiple alignment.

A very useful fact, which is true only if we have $p(-, -) = 0$, is the following formula for the SP score of a multiple alignment α:

$$SP\text{-}score(\alpha) = \sum_{i<j} score(\alpha_{ij}), \qquad (3.10)$$

where α_{ij} is the pairwise alignment induced by α on sequences s_i and s_j. This is true because it reflects two ways of doing the same thing. We may compute the score of each column and then add all column scores, or compute the score for each induced pairwise alignment and then add these scores. In any case, we are adding, for each column c and for each pair (i, j), the score $p(s'_i[c], s'_j[c])$, where s' indicates the extended sequence (with spaces inserted). But this is true only if $p(-, -) = 0$, because these quantities appear in the first computation only.

Multiple alignment

```
1 PEAALYGRFT---IKSDVW
2 PEAALYGRFT---IKSDVW
3 PESLAYNKF---SIKSDVW
4 PEALNYGRY---SSESDVW
5 PEALNYGWY---SSESDVW
6 PEVIRMQDDNPFSFQSDVY
```

Get only sequences 2 and 4

```
PEAALYGRFT---IKSDVW
PEALNYGRY---SSESDVW
```

Remove columns with two spaces

```
PEAALYGRFT-IKSDVW
PEALNYGRY-SSESDVW
```

FIGURE 3.11

Induced pairwise alignment (projection).

Dynamic Programming

Having decided on a measure, or score, for determining the quality of a multiple alignment, we would like to compute the alignments of maximum score, given a set of sequences. These will be our *optimal alignments*.

It is possible to use a dynamic programming approach here, as we did in the two-sequence case. Suppose, for simplicity, that we have k sequences, all of the same length n

We use a k-dimensional array a of length $n + 1$ in each dimension to hold the optimal scores for multiple alignments of prefixes of the sequences. Thus, $a[i_1, \ldots, i_k]$ holds the score of the optimal alignment involving $s_1[1..i_1], \ldots, s_k[1..i_k]$.

After initializing with $a[0, \ldots, 0] \leftarrow 0$, we must fill in this entire array. Just to store it requires $O(n^k)$ space. This is also a lower bound for the computation time, because we have to compute the value of each entry. The actual time complexity is higher for a number of reasons. First, each entry depends on $2^k - 1$ previously computed entries, one for each possible composition of the current column of the alignment. In this composition, each sequence can participate with either a character or a space. Because we have k sequences, we have 2^k compositions. Removing the forbidden composition of all spaces, we obtain the final count of $2^k - 1$. This incorporates a factor of 2^k to the already exponential time complexity.

Then there is the question of accessing the data in the array. Very few programming languages, and certainly none of the most popular ones, will let users define an array with the number of dimensions k set at run-time. The alternative is to implement our own access routines. In any case, with or without language support, we can expect to spend $O(k)$ steps per access.

Another issue is the computation of column scores. The SP method requires $O(k^2)$ steps per column, as there are $k(k-1)/2$ pairwise scores to add up. Simpler schemes — for instance, just count the number of nonspace symbols — require at least $O(k)$ steps, given that we have to look at all arguments.

Finally, there is the actual computing of the value of $a[i_1, \ldots, i_k]$, which involves a maximum operation. Using boldface letters to indicate k-tuples, the command we must perform can be written as

$$a[\mathbf{i}] \leftarrow \max_{\mathbf{b} \neq \mathbf{0}} \{a[\mathbf{i} - \mathbf{b}] + \textit{SP-score}(\textit{Column}(\mathbf{s}, \mathbf{i}, \mathbf{b}))\},$$

where $\mathbf{b}$ ranges over all nonzero binary vectors of k elements, and

$$\textit{Column}(\mathbf{s}, \mathbf{i}, \mathbf{b}) = (c_j)_{1 \leq j \leq k}$$

with

$$c_j = \begin{cases} s_j[i_j] & \text{if } b_j = 1 \\ - & \text{if } b_j = 0. \end{cases}$$

The total running time estimate for this first plan for implementation is therefore $O(k^2 2^k n^k)$ if we use SP, or $O(k 2^k n^k)$ if column scores can be computed in $O(k)$. If k is fixed, k nested **for** loops can be used to fill in the array. Optimal alignments can be recovered from this array by a backtracking procedure analogous to the one used in the pairwise case (Section 3.2.1). Straightforward extensions yield a similar method for the case where the sequences do not necessarily have the same length. In any case, the complexity of this algorithm is exponential in the number of input sequences, and the existence of a polynomial algorithm seems unlikely: It has been shown that the multiple alignment problem with the SP measure is NP-complete (see the bibliographic notes).

Saving Time

The exponential complexity of the pure dynamic programming approach makes it unappealing for general use. The main problem is the size of the array. For three sequences we already have a cube of $O(n^3)$ cells, and as the number k of sequences grows we have larger and larger "volumes" to fill in. Clearly, if we could somehow reduce the amount of cells to compute, this would have a direct impact on processing time.

This section describes a heuristic that does exactly that. We will show how to incorporate it into the dynamic programming algorithm to speed up its computation. It is a heuristic because in the worst case all cells will have to be computed; in practice, however, we can expect a good speedup. The heuristic is based on the relationship between a multiple alignment and its projections on two-sequence arrays, and, in particular, it uses Equation (3.10) relating SP scores to pairwise scores. Thus, the method we are about to see works only for the SP measure.

The outline of the method is as follows. We have k sequences of length n_i, for $1 \leq i \leq k$, and we want to compute the optimal alignments according to the SP measure. We will still use dynamic programming, but now we do not want to treat all cells. We just want the cells "relevant" to optimal alignments, in some sense. But exactly which cells will we deem relevant, and why?

The answer is to look at the pairwise projections of the cell. In a preprocessing step, we create conditions that will allow us to perform a test of relevance for arbitrary cells. To take advantage of this test and reduce the number of cells we need to look at, we have to modify the fill-in order as well.

Dunque il problema è esponenziale e non si può ragionevolmente affrontarlo, se non per valori di k molto piccoli.

L'algoritmo euristico introdotto sopra è piuttosto complicato e non è riportato in questa dispensa; ma è bene sapere che esiste, e ovviamente si può trovare nel testo di Setubal e Meidanis. La sua complessità è di ordine $O(\max(kn^2, k^2a))$, ove a è la lunghezza dell' allineamento che sarà generato (quindi a sarà compreso tra n e kn). Non vi è modo di attestare la bontà del risultato, cioè di quanto questo è lontano dall'ottimo.