CS 2427 - Algorithms in Molecular Biology

Lecture #6b: 27 January 2006

Lecturer: Michael Brudno

Scribe Notes by: Graham Taylor

1 Sequence comparison

Consider the following problem. Someone gives you two arbitrary sequences (genomic or protein). They differ not just in evolutionary mutations, but also could have insertions/deletions. Biologists are often interested in comparing these sequences. Defining a method to measure similarity between sequences is a first step towards more complex tasks such as sequence alignment (discussed below). Before we describe some methods of comparison, it is essential to distinguish between what is a subsequence and what is a substring.

Definition A string v is a substring of a string u if $u = u'vu''$ for some prefix u' and suffix u''.

Definition A subsequence is any string that be obtained by deleting zero or more symbols from a given string.

The difference between the two is that a substring is contiguous while a subsequence need not be. The latter allows us to account for insertions and deletions in the string.

Example Consider the string `ACTATTAC`.

- `ACT`, `CTAT`, and `ATTAC` are substrings (and subsequences)
- `ATA`, `CAAC`, and `ATTT` are subsequences but not substrings

If we are given two strings of different length, say `ACTAAGT` & `ACACGT`, how might they be compared?

Two approaches to consider are:

1. Compare substrings. We have methods of finding the longest common substring (typically done by using suffix trees) but this method will overlook insertions and deletions.
2. Compare subsequences. But how do we find the longest common subsequence?

1.1 Finding the longest common subsequence (LCS)

One method of comparing two strings is by first identifying their longest common subsequence. Fist we will see how to solve the LCS problem, and then this approach will be extended to the more general sequence alignment problem. To find a LCS, we employ a technique called "dynamic programming". This is a method for reducing the runtime of algorithms exhibiting the properties of overlapping subproblems and optimal substructure. What this means is that we can break a larger problem into subproblems, find the optimal solution to these subproblems, and then assemble these solutions into a solution for the overall problem.

Dynamic programming stores its work in a matrix, $M(i,j)$ which is defined by a recursion and a set of initial values. The value of $M(i,j)$ is the length of the longest common subsequence that uses the first i letters of sequence 1 and first j letters of sequence 2. $M(i,j)$ can be computed iteratively, as follows

$$M(i,j) = \max\left(M(i-1,j), M(i,j-1), M(i-1,j-1) + (A_i == B_j)\right).$$

The last term means we increase the value of the longest common subsequence by 1 if the sequences have the same letter at position i and j, respectively. If we wish to reassemble the actual LCS, and not just track length, we must also store pointers to the matrix element from which each matrix element was computed.

Example Find the LCS of the strings `ACACGT` and `ACTAAGT`.

We define $M(i,k) = 0 \forall k < 1 and M(k,j) = 0 \forall k < 1$. We then begin applying the recursion.

$M(1,1) = \max\left(M(0,1), M(1,0), \underline{M(0,0)+(1)}\right) = 1 \ (A(1) = B(1))$

$M(1,2) = \max\left(M(0,2), \underline{M(1,1)}, M(0,1)+(0)\right) = 1 \ (A(1) \neq B(2))$

$M(2,1) = \max\left(\underline{M(1,1)}, M(2,0), M(1,0)+(0)\right) = 1 \ (A(2) \neq B(1))$

$M(2,2) = \max\left(M(1,2), M(2,1), \underline{M(1,1)+(1)}\right) = 2 \ (A(2) = B(2))$

and so on. Once we have the values of cell $M(i,j)$'s "up", "right", and "up-right" neighbours, we can fill in its value. We continue recursively until the matrix is filled:

0	A	C	T	A	A	G	T
A	↖1	1	1	1	1	1	1
C	1	↖2	←2	2	2	2	2
A	1	2	2	↖3	3	3	3
C	1	2	2	3	↖3	3	3
G	1	2	2	3	3	↖4	4
T	1	2	3	3	3	4	↖5

Note that the elements corresponding to the max above have been underlined. These can be stored as pointers to step back through the matrix and reconstruct the LCS. The length of the LCS is given by the bottom-left element, $M(m,n)$. In this example, the LCS has length 5. It can be reconstructed by following the pointers backward, from $M(m,n)$. These are indicated by arrows in the figure above. From this reconstruction, we see that the LCS is `ACAGT`.

Note that the diagonal movements to cells i,j where $A(i) = B(j)$ correspond to elements in the LCS. Diagonal movements to cells i,j where $A(i) \neq B(j)$ are not included in the LCS. Horizontal or vertical movements correspond to skipping over elements (we will refer to these later as gaps).

These gaps are necessary for us to process strings of different length. Note that in some cases, there may be multiple options for the pointers (when two or three neighbouring elements have the same value). In this case, an alternative LCS may be identified, but it will have the same length.

So far, we have ignored the value of mismatches, insertions and deletions. How can we account for these?

1.2 Sequence alignment

Note that there are three ways of arriving at a value for $M(i, j)$ (through the three terms in the maximum, above). We may take the final term, $M(i-1, j-1) + (A_i == B_j)$, which corresponds to $N(A_i, B_j)$ matches. The other two correspond to "gaps", when an element in sequence A is not matched to an element in sequence B (or vice-versa). If the term $M(i-1, j)$ is maximum, this corresponds to a gap in sequence A and if the term $M(i, j-1)$ is maximum, this corresponds to a gap in sequence B.

An alignment of two sequences A and B is an arrangement by position where both can be padded by gap symbols, and the length of both, counting the gap symbols is the same.

Example Given the sequence AGCACACA and ACACACTA, two possible alignments (out of many) are:

```
AGCACAC-A          or          AG-CACACA
A-CACACTA                      ACACACT-A
```

The sequence alignment specifies a way of transforming sequence A into B and vice-versa through insertions, deletions and replacements. The optimal global alignment is one which minimizes the total cost of transforming A into B, where each operation has some cost associated with it. While there are many ways to define the cost assignment, one simple approach is to consider the Jukes-Cantor model (discussed in Lecture 3).

1.3 Scoring matrices

Our task is to assign a score to

- matches,
- mismatches,
- and gaps.

Let us consider first the case of scoring matches and mismatches for genomic sequences. A scoring matrix for an alphabet Σ with elements $S(i, j)$ can be built such that a high score is assigned if $i = j$ (corresponding to a match) and low score if $i \neq j$. The construction of S is statistically motivated. Given a gap-free alignment, we have two potential hypotheses for the alignment: Homologous (alignment because the sequences are alike because of shared ancestry) and Random.

The score is defined as the log-odds of the alignment (data) given the Homologous hypothesis and the alignment (data) given the Random hypothesis. Assuming independent sites and the

Jukes-Cantor model for the former hypothesis, and independent sites and equal frequencies for all base pairs in the latter hypothesis, the log-odds has the form

$$\log \frac{P(data|H)}{P(data|R)} = a \log \frac{1-p}{1/4} + d \log \frac{p}{3/4},$$

where a is the number of agreements (matches), d is the number of disagreements (mismatches), and $p = \frac{3}{4}(1 - e^{-8\alpha t})$ is the probability, from the Jukes-Cantor model, that an observed nucleotide has changed after some time t. The parameter αt is the Jukes-Cantor distance. (For the derivation and more details, see Speed's lecture notes on the course webpage).

Since $p < \frac{3}{4}$, $\log(\frac{p}{3/4})$, and $\log(\frac{1-p}{1/4}) > 0$, we can rewrite the score as

$S = a \times \sigma + d \times (-\mu)$, where σ is the score for a match and $-\mu$ is the score (penalty) for a mismatch. Since sites are treated independently, we can look at each pair of elements in the alignment one at a time, and consider its score, which can be expressed in matrix form, $S(i, j)$, and dependent on the parameters αt (Fig. 1). The off-diagonal elements have identical scores since the Jukes-Cantor model assumes that every nucleotide can mutate into another with equal probability (a similar argument is made for the diagonals). If we had used a more complex (and realistic) model, then each element could be different.

	A	C	G	T
A	4	-2	-2	-2
C	-2	4	-2	-2
G	-2	-2	4	-2
T	-2	-2	-2	4

Figure 1: Similarity matrix example

The elements of the log-odds score matrix are usually positive on the diagonal and negative off the diagonal. This is not always the case. We note that in the Jukes-Cantor model, the parameter αt is a distance, whereas the score $S(i, j)$ is a similarity. $S(i, j)$ is also known as a similarity matrix. While we have presented one way of generating a similarity matrix, many other matrices, based on other principles, do exist.

For simplicity, we will assign a constant penalty of $G = -5$ to a gap. So we can see that longer gaps will be penalized more (by a constant amount per gapped element), but we do not distinguish, in terms of penalty, between opening a new gap and extending an existing gap. We may want to break down the penalty such that opening a gap is penalized more than simply extending an existing gap. This will be explored in a future lecture.

To implement this simple scoring assignment, we can update our equation for $M(i, j)$ as follows:

$$M(i, j) = \max\left(M(i-1, j) + G, M(i, j-1) + G, M(i-1, j-1) + S(A_i, B_j)\right).$$

While dynamic programming is used for many problems in computer science, Needleman and Wunsch in 1970 were the first to apply this technique to sequence alignment and thus the above recurrence is known as the Needleman-Wunsch algorithm. The first term in the max expression above corresponds to aligning sequence A to a gap while the second term corresponds aligning sequence B to a gap. The constant gap penalty, G, is given to each gap in the alignment. The

third term corresponds to a match (or mismatch) which is scored according to the similarity matrix, S. Note the difference between the LCS dynamic programming algorithm when we added 1 only when we had a match. Now we can choose to mismatch, applying the mismatch score for two elements when it is more cost-effective than "gapping".

Example Consider the alignment :
```
AG-CACACA
ACACACT-A
```

The first, fourth, fifth, sixth and ninth pairs in the alignment correspond to choosing the third term in the max expression above (a match).
The second, and seventh pairs in the alignment correspond to choosing the third term (a mismatch).
The third pair corresponds to choosing the second term (a gap in sequence A).
The eighth pair corresponds to choosing the first term (a gap in sequence B).

1.4 Computational concerns

Consider the pseudo-code (Matlab notation!), below, for filling the matrix M. Assume M is initialized to zeros and S is in memory.

```
for i=1:m
  for j=1:n
    %Returns both the maximum value and the index (1,2,3) of the maximum element
    [M(i,j),ind] = max ( M(i-1,j)+G, M(i,j-1)+G, M(i-1,j-1) + S(A_i,B_j) )
    if ind == 1
      Pointer(i,j) = (i-1,j)
    elseif ind == 2
      Pointer(i,j) = (i,j-1)
    else
      Pointer(i,j) = (i-1,j-1)
    end
  end
end
```

Here we see that the matrix is filled in row-by-row. It could also be computed column-by-column. However, there is an alternative which is advantageous in the case of aligning very long sequences. If we fill the matrix along the diagonal, note that the computation of an element in the diagonal does not rely on any elements in that same diagonal – only elements from previous diagonals (Fig 2). This allows for parallelization of the algorithm, since diagonal elements can be computed simultaneously (assuming that the previous diagonals have already been computed). This is not possible in a row-by-row filling, since elements in the row rely on other elements in the same row.

The order is shown in Figure 3. Diagonal 1 is filled, then diagonal 2, and so on . . . Each element of the same number can be passed to a different cluster to be computed.

The time and space complexity of computing M is $O(mn)$ where m and n are the lengths of sequence 1 and 2, respectively. If we do not care about what the LCS actually looks like, but just its length, then our complexity is reduced to $O(\min(m, n))$.

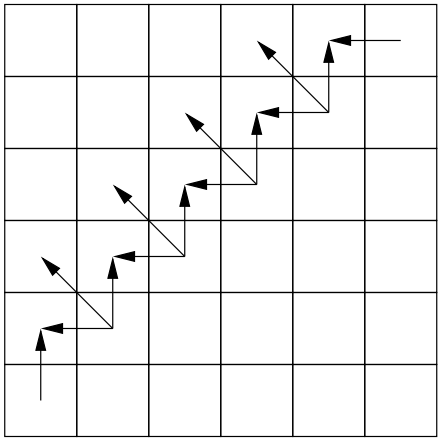

Figure 2: Diagonals only rely on previously computed diagonals

1	2	3	4	5	6
2	3	4	5	6	7
3	4	5	6	7	8
4	5	6	7	8	9
5	6	7	8	9	10
6	7	8	9	10	11

Figure 3: Order of diagonal processing

References

These lecture notes contain some material from the course readings,

- Huson's notes on HMMs & CpGs
- Speed: Lecture Notes on Sequence Alignment