Discussion

6.4 ***Tell how you would compute the optimal alignment from the score matrix.***

The simplest way is to keep a matrix of alignments which keeps the optimal alignment for each pair of prefixes. This matrix would have the same dimensions as the opt_score matrix. Each cell would contain a pair of strings representing an alignment. Each string would consist of the characters A, C, G, T, or -. The character “-” indicates a gap. The optimal alignment for each entry in this matrix (corresponding to each pair of prefixes) can be computed by seeing which of the 3 possible choices in the corresponding cell of opt_score was the minimum. Say we’re computing the optimal alignment in row i, column j (so for the prefix of length i of one string with the prefix of length j of the other string).

If `align_i_with_j` was minimum, then we start with the optimal alignment stored in row `i-1` and column `j-1`, add the character `Dseq1[i]` to the first alignment string, and add the character `Dseq2[j]` to the second alignment string.

If `align_seq1_i_with_gap` was minimum, then we start with the optimal alignment stored in row `i-1` and column `j`, add the character `Dseq1[i]` to the first alignment string, and add the gap character “–” to the second alignment string.

If `align_seq2_j_with_gap` was minimum, then we start with the optimal alignment stored in row i and column `j-1`, add the character “–” to the first alignment string, and add the character `Dseq2[j]` to the second alignment string.

The last entry in this matrix will then have the optimal alignment.

Unfortunately, this matrix contains a great many characters. So many, in fact, that it is usually considered too many to keep, as it would use up too much of the computer’s memory. So instead of the algorithm described above, all that is usually stored in addition to the `opt_score` matrix is which of the possible 3 choices for opt_score was minimum for each cell/entry of opt_score. We need to record one of these three “choices” even for the cells filled during initialization where there were no choices, i.e. row 0 and column 0. Every entry in row 0 should indicate that `align_seq2_j_with_gap` was minimum. This is because the prefixes corresponding to the entries in row 0 contain no characters of DNA sequence 1, so all DNA sequence 2 characters are aligned with a gap. Similarly, every entry in column 0 should indicate that `align_seq1_i_with_gap` was minimum. The first entry (row 0, column 0) will not be used, so it doesn’t matter what is stored here.

Then we build the alignment backwards from the end. Knowing which choice was selected here tells us what the end of the alignment looks like. Recall that the three choices for the end of the alignment are that the final character of each DNA sequence may be aligned with each other, or that the final character of the first DNA sequence may be aligned with a final gap, or the final character of the second DNA sequence may be aligned with a final gap. So we can write down the end of the alignment. Then we can look at the remaining problem, working our way through this matrix until we get to the beginning.

Specifically, we begin in the last cell, so we set `i = m` and `j = n`, and we begin with two empty strings representing the optimal alignment.

If `align_i_with_j` was minimum, then we insert the character `Dseq1[i]` in the front of the first alignment string, and insert the character `Dseq2[j]` in the front of the second alignment string. Then we reduce both `i` and `j` by 1 (`i = i-1 and j = j-1`).

If `align_seq1_i_with_gap` was minimum, then we insert the character `Dseq1[i]` in the front of the first alignment string, and insert the character "–" in the front of the second alignment string. Then we reduce `i` by 1 and leave `j` unchanged (`i = i-1`).

If `align_seq2_j_with_gap` was minimum, then we insert the character "–" in the front of the first alignment string, and insert the character `Dseq2[j]` in the front of the second alignment string. Then we leave `i` unchanged and reduce `j` by 1 (`j = j-1`).

When we get to `i=0` and `j=0` we are done.

6.6 ***Describe your testing strategy for code from Sections 6.1, 6.2, and 6.3. You should describe the types of inputs you used.***

To test the code appropriately, you should try all the types of valid and invalid inputs which a user might enter. Note that we are talking about testing various types of inputs which a user enters into the running code, not what the code should do (which was called error checking).

For all sections, you should test the following types of invalid input to ensure the program handles them appropriately:

- A DNA sequence with characters other than A, C, G, T.
- A DNA sequence with numeric "characters".
- A DNA sequence with more than 20 bases (if this is invalid for your program).

In addition, test sequences of differing lengths for Hamming distance and weighted distance.

For all sections, test the following types of valid input to ensure the program handles them appropriately:

- 20-character long sequences
- 1 character sequences
- identical sequences
- sequences which differ at every position
- sequences which differ only in the first position
- sequences which differ only in the last position

In addition, for the weighted distance you should check:

- sequences which have a different Hamming distance than weighted distance
- using a matrix with 1's for all non-diagonal entries to test that the weighted distance then returns the same answer as Hamming distance.

In addition, for the minimum distance you should check:

- sequences of equal length for which the minimum distance is the weighted distance (i.e., no gaps are required).
- sequences of equal length which require a gap for the minimum distance (i.e., sequences with different minimum distance than weighted distance)
- sequences where the first sequence is shorter than the second.
- sequences where the first sequence is longer than the second.
- sequences of unequal length where the only gaps required for the minimum distance are at the end of the shorter sequence.
- sequences of unequal length where the only gaps required for the minimum distance are at the beginning of the shorter sequence.
- sequences of unequal length where the only gaps required for the minimum distance are in the middle of the shorter sequence.
- sequences of unequal length where gaps are required at the beginning of the longer sequence for the minimum distance.
- sequences of unequal length where gaps are required at the end of the longer sequence for the minimum distance.
- sequences of unequal length where gaps are required in the middle of the longer sequence for the minimum distance.
- sequences of unequal length where gaps are required in both sequences for the minimum distance.

Any of these tests could uncover a problem in the code which might go undetected by the other tests.

In what ways did your testing strategy differ for the different distance methods? Why?

In the Hamming distance method, all pairs of characters which are not the same contribute equally to the total distance, but this is not true of the weighted distance. For this reason, we needed to test more cases for the weighted distance method than for the Hamming distance method. In particular, we could test a pair of DNA sequences which have the same non-zero weighted distance as Hamming distance, and another pair of DNA sequences where the weighted distance and Hamming distance differ. For truly robust testing, we could use pairs of DNA sequences which use each of the 16 entries in the substitution matrix.

Many more types of input need to be tested for the minimum distance method than for the other distance methods. This is because the minimum distance method is more flexible and much more complex.

How did the results of the different methods vary? Why?

The Hamming distance will never be greater than the weighted distance. This is because they both consider pairs of characters between 2 sequences in their "natural" alignment (i.e., the first character of each sequence are "paired", the second characters are "paired", etc.). In this pairing, differing characters have a difference of 1 in the Hamming distance, and a difference of at least 1 in the weighted distance.

If two DNA sequences can be compared with the weighted distance (that is, if the two DNA sequences have the same length), then the minimum distance will never be greater than the weighted distance. This is because they both use the same substitution matrix, but the minimum distance has the flexibility to add gaps to get a smaller distance. The minimum distance will never be greater than the weighted distance, because the weighted distance is one of the valid alignments considered by the minimum distance method.

It should be noted that the Hamming distance need not be less than the minimum distance. For example, take the DNA sequence "TATATATATA" ("TA" repeated 5 times) and for our first string insert a "C" in the beginning, and for our second string instead add a "C" at the end, so our two DNA sequences are: CTATATATATA TATATATATAC The Hamming distance between these two DNA sequences is 11 (all pairs differ), the weighted distance is 31 (the distance between the initial bases "C" and "T" is 1, the other 10 distances are 3), and the minimum distance is 10 (achieved by adding a gap to the end of the first sequence and a gap at the beginning of the second sequence), so in this case the minimum distance is smaller than the Hamming distance.

The minimum distance is more flexible than the other two methods since the two DNA sequences it considers do not need to be the same length.