Computational Biology
Lecture 3: More problems, Global Alignment

Saad Mneimneh

DNA sequencing

Recall that in biological experiments only relatively short segments of the DNA can be investigated. To investigate the structure of a long DNA molecule, the DNA is broken into smaller fragments that can be studied more easily by biological experiments. For instance, physical mapping aims at ordering these fragments based on overlap information obtained from hybridization experiments with a set of probes.

Shortest Superstring

In DNA sequencing, we have we deal with a more accurate overlap data for the fragments. For instance, in such an experiment, we manage to sequence each of the individual fragments, i.e. obtain the sequence of nucleotides for each fragment (there might be some experimental errors). Now we have map them back on the original DNA.

This leads to the following nice theoretical abstraction, the Shortest Superstring (a one that does not necessarily capture reality): Given a set of strings s_1, s_2, ..., s_k, find the shortest string s such that each s_i appears as a substring of s.

Here's an example. Consider the following strings (all binary strings of length 3): 000, 001, 010, 011, 100, 101, 110, 111. A trivial superstring would be 000 001 010 011 100 101 110 111. However a shortest superstring would be 0 0 0 1 1 1 0 1 0 0.

Intuitively speaking, the shortest superstring uses as much overlap as possible to construct the original DNA (because it is the shortest string that contains all fragments). Is that a realistic thing to do? No, what if the DNA contains repeats? Then the shortest superstring will not capture that. Furthermore, the model does not account for the possible experimental errors.

The shortest superstring problem is NP-hard.

Sequencing by Hybridization

Although the sequencing problem can be solved in fast and efficient techniques these days, no such techniques existed 10 years ago. One method for DNA sequencing known as Sequencing by Hybridization was suggested by four biologists independently in 1988. They suggested to build a DNA "ship" (called microarray) that would contain many small fragments (probes) acting as the memory of the chip. Each fragment will give some information about the DNA molecule at hand (by hybridization), and they will collectively solve the DNA sequencing problem. Of course no one believed that this would work. Now microarrays constitute an industry.

Theoretically speaking, how can we use such a microarray to sequence the DNA. We can make the memory of the ship constitute of all possible probes of length l. We ship will determine whether each of the probes hybridizes with the DNA molecule. Note that every substring of length l in the original DNA will hybridize with some probe in the microarray. Therefore, we collect information about all substrings of length l in the original DNA. This information can be used to figure out the DNA sequence. We will call this problem, sequencing by hybridization, SBH.

Note that SBH is theoretically not much different than the shortest superstring problem described above. In fact it is a special case where all strings have the same length l. Although the shortest superstring problem is NP-hard, SBH can be solved efficiently as we will see later on.

Finding CG island

Given a DNA sequence we would like to find out which stretches on the DNA are genes, i.e. code for proteins. We face the gene prediction problem. One approach to solving the Gene Prediction problem is by predicting what is known as CG islands. CG is the most infrequently occurring dinucleotide in many genomes because CG tends to mutate to TG by a process called methylation. However, CG tends to appear frequently around genes in areas called CG island. Therefore, given the DNA, determining which areas are CG islands and which areas are not helps us to identify genes. Theoretically, this problem is analogous to the following dishonest casino problem: At the casino, a biased coin is used from time to time. The players cannot notice since the switch between the two coins happens with a small probability p. Given a sequence of coin tosses that you collect over a period of time, can you find out when the biased coin was used and when the fair coin was used? Where is the analogy here? Well, Given the DNA sequence, can you tell which parts are CG islands and which parts are not? We will use a Hidden Markov Model to abstract this probabilistic setting and answer questions such as the one above.

Similarity search

After sequencing, the biologist would have no idea about a newly sequenced gene. In a similarity search we are interested in comparing the newly sequenced gene with others whose functionality is known. This will help us determine the functionality of the newly sequence genes. Whether it is a genes, a protein, or just a DNA fragment, we can look at the new sequence a string over some alphabet (A, G, C, T for instance in case of DNA). Using a distance metric we can then determine to what extent the strings are alike. This is often called an edit distance. The edit distance is a measure of how many operations are needed to transform one string into another. The operations can be insertion, deletion or substitution of a symbol. Hence the edit distance is a natural measure of the similarity between the two strings, because these are the three kinds of mutations that occur in DNA and protein sequences. We will see when we discuss alignment algorithms that we can work with a more elaborate weighted edit distance, where each operation is assigned a weight, or a score.

As for an example for now, if we decide to limit the mutations to just insertions and deletions and no substitutions then the edit distance problem is equivalent to finding the longest common subsequence. Given two strings $u == u_1...u_m$ and $v = v_1...v_n$ a common subsequence of u and v of length k is a set of indices $i_1 < i_2 < ... < i_k$ and $j_1 < j_2 < ... < j_k$ where $u_{i_x} = v_{j_x}$ for $x = 1...k$. Let $LCS(u, v)$ denote the length of the longest common subsequence, then the minimum number of insertions and deletions is clearly equal to $m - LCS(u, v) + n - LCS(u, v) = m + n - 2LCS(u, v)$. This problem is a basic case of a general problem known as Sequence Alignment.

Global Alignment

In order to detect similarities/differences between two sequences we align the two sequences on above the other. For instance the following two sequences look very much alike: GACGGATTAG and GATCGGAATAG. The similarity becomes more obvious when we align the sequences as follows:

```
GA-CGGATTAG
GATCGGAATAG
```

A gap was inserted in the first sequence to properly align the two sequences. Sequence alignment is the insertion of gaps (or spaces) in arbitrary locations in the sequences so that they end up with the same size, provided that no gap in one sequence should be aligned to a gap in the other.

We are interested in finding the best alignments between two sequences. But how do we formalize the notion of best alignment? We will define a scoring scheme for the alignment. The best alignment is the alignment with the highest score. Since our goal is to determine how similar two sequences are, our scoring scheme should reflect the concept of similarity. A basic scoring scheme would be the following: two identical characters receive a score of +1, two different characters receive a score of -1, and a character and a gap receive a score of -2. Then the score of our alignment will be the sum of the individual scores in each column of the alignment. For example, the score of the alignment above is $+1(9) - 1(1) - 2(1) = 6$. Here we choose to penalize a gap more than a mismatch because insertions and deletions are less likely to occur in practice than substitutions.

In general, our scoring could be $+m$ for a match, $-s$ for a mismatch, and $-d$ for a gap. The score of the alignment will therefore be:

$$+m(\#matches) - s(\#mismatches) - d(\#gaps)$$

The scoring scheme could be generalized even further. For instance, we could have a scoring matrix S where $S(a, b)$ is the score for matching character a with character b. The gap (or the space) symbol could be made part of that scoring matrix; however, it is usually dealt with separately. As we will see later on, the score for a gap could depend on the length of the gap. For instance two gap characters in a sequence could be scored differently than twice the score of a single gap. The reason for such an approach is that gaps tend to occur in bunches, i.e. once we have a gap, we are likely to encounter another one again, so we might want to penalize the start of the gap sequence more. This results in an non-linear gap penalty function that cannot be captured by the scoring matrix S.

Now that we have an idea on how to score our alignment we go back to the question of how to compute the optimal alignment. One possibility is by using a greedy algorithm which looks at all possible alignments and determines the optimal one. However, the number of alignment grows exponentially with the sequence length resulting in a slow algorithm.

We will make use of Dynamic programming to solve the problem. In dynamic programming we first find a solution for the sub-problem and then store the results obtained and use them in the computation of larger sub-instances of the problem. The dynamic programming algorithm is going to align prefixes of the two sequences x and y. By looking at the optimal alignment of prefixes of x and y, we will be able to obtain the optimal alignment for larger prefixes. At the end, we have the alignment between x and y.

In developing our dynamic programming algorithm, we will rely on two main observations:

First, using our scoring scheme of $(+m, -s, -d)$, the score of an alignment (whether optimal or not) is additive. In other terms, if the alignment can be split in two parts, and each part is scored separately, then the sum of the two scores is equal to the score of the alignment. For instance, the alignment above can be split as follows:

```
GA- | CGGATTAG
GAT | CGGAATAG
```

The score of the alignment can be obtained as the sum of the scores of the two parts, i.e. (1+1-2) + (1+1+1+1-1+1+1+1) = 0 + 6 = 6. This is true for any split of the alignment. The additive property implies that if the alignment is optimal so are the alignments corresponding to the two parts (because otherwise we can obtain a better alignment).

Second, the optimal alignment between $x_1...x_i$ and $y_1...y_j$ can end with one of three possibilities.

- x_i is aligned to y_j
- x_i is aligned with a gap
- y_j is aligned with a gap

Using the additivity property mentioned above, the score of an optimal alignment ending in the first case will be $OPT(x_1...x_{i-1}, y_1...y_{j-1}) + s(x_i, y_j)$ where $OPT(x_1...x_{i-1})$ is the score of the optimal alignment between $x_1...x_{i-1}$ and $y_1...y_{j-1}$, and $s(x_i, y_j) = +m$ if $x_i = x_j$ and $s(x_i, y_j) = -s$ otherwise.

Similarly, the score of an optimal alignment ending in the second case will be $OPT(x_1...x_{i-1}, y_1...y_j) - d$. Finally, the score of an optimal alignment ending in the third case will be $OPT(x_1...x_i, y_1...y_{j-1}) - d$.

Let $A(i, j)$ be the score of the optimal alignment between $x_1...x_i$ and $y_1...y_j$. Then according to the discussion above, if we know $A(i-1, j-1)$, $A(i-1, j)$, and $A(i, j-1)$, then we can determine $A(i, j)$. How? Well, one the the three possibilities above has to be true for the optimal alignment between $x_1...x_i$ and $y_1...y_j$; therefore, $A(i, j)$ should be computed as

$$A(i, j) = \max[A(i-1, j-1) + s(x_i, y_j), A(i-1, j) - d, A(i, j-1) - d]$$

If we view A as a matrix, then an entry $A(i, j)$ depends on three other entries as illustrated in Figure 1.

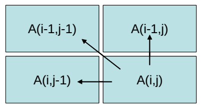


Figure 1: Computing $A(i, j)$

We can therefore carry the computation row by row from left to right (or column by column downward) until we obtain the value for $A(m, n)$ which will be the score of the optimal alignment between $x_1...x_i$ and $y_1...y_j$. But how do we start the process? We need some initialization. For instance $A(1, 1)$ will require knowledge of $A(0, 0)$, $A(0, 1)$ and $A(1, 0)$. Well, $A(0, 0)$ corresponds to aligning the two empty strings of x and y. Therefore it must have a score of zero. $A(i, 0)$ for $i = 1...m$ corresponds to aligning the prefix $x_1...x_i$ with the empty string of y, i.e. with gaps. Therefore, $A(i, 0)$ should be set to $-d \times i$ for $i = 1...m$. Similarly, $A(0, j)$ should be set to $-d \times j$ for $j = 1...n$.

Figure 2 shows an example of aligning the sequence $x = AAAC$ and $y = AGC$:

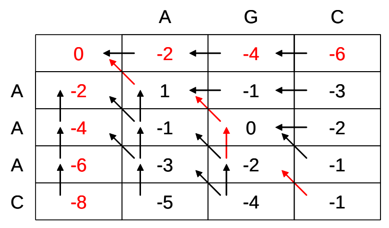
The question is now how to obtain the optimal alignment itself, not just its score. We will simply keep pointers in the dynamic programming table to keep a trace on how we obtained each entry; therefore, for each $A(i, j)$ we have a pointer(s) to the entry(ies) that resulted in the value for $A(i, j)$. At the end, we trace back on a path from $A(m, n)$ to $A(0, 0)$. On a path, a diagonal pointer at $A(i, j)$ signifies the alignment of x_i with y_j, a upward pointer at $A(i, j)$ signifies the alignment of x_i with a gap, and a left pointer at $A(i, j)$ signifies the alignment of y_j with a gap.

Figure 3 shows how pointers are kept during the computation.

Figure 4 illustrates the final algorithm, known as the Needleman-Wunsch algorithm. The algorithm has $O(mn)$ time and space complexity since it involves only the computation of the matrix A and each entry requires a constant time to compute (checking 3 other entries and setting at most three pointers).

		A	G	C
	0	-2	-4	-6
A	-2	1	-1	-3
A	-4	-1	0	-2
A	-6	-3	-2	-1
C	-8	-5	-4	-1

Figure 2: Example



AAAC
AG-C

Figure 3: Keeping pointers

```
1.    Initialization
          $A(0, 0)$    $= 0$
          $A(i, 0)$    $= -i.d$ for $i = 1 \ldots m$
          $A(0, j)$    $= -j.d$ for $j = 1 \ldots n$

2.    Main Iteration (Aligning prefixes)
          for each  $i = 1 \ldots m$
              for each  $j = 1 \ldots n$
                  $A(i, j) = \max \begin{cases} A(i-1, j-1) + s(x_i, y_j) & \text{[case 1]} \\ A(i-1, j) - d & \text{[case 2]} \\ A(i, j-1) - d & \text{[case 3]} \end{cases}$
                  $Ptr(i, j) = \begin{cases} \text{Diag} & \text{[case 1]} \\ \text{Up} & \text{[case 2]} \\ \text{Left} & \text{[case 3]} \end{cases}$

3.    Termination
          $A(m, n)$ is the optimal score, and
          from $Ptr(m, n)$ can trace back optimal alignment.
```

Figure 4: The Needleman-Wunsch Algorithm

References

Pevzner P., Computational Molecular Biology, Chapter 1.
Setubal J., Meidanis, J., Introduction to Molecular Biology, Chapter 3.