CS 262 Lecture 2 Scribe Notes
Sequence Alignment

Lecture By: Prof. Serafim Batzoglou
Scribe: Nico Chaves

Jan 7, 2016

1-Sentence Summary: In this lecture, we learned a basic algorithmic technique for comparing biological sequences.

1 Evolution & Motivation For Sequence Alignment

One way to measure similarity between species is the number of substitutions per site.

Phylogenetic tree: Notice that some of the branches on the tree (see the image shown in lecture) are longer than others. For example, mice have a much longer branch than humans.

There are 2 important reasons for this:
1. **Mice have a shorter lifespan**, so they've undergone many more generations.
2. **Population size**: many generations ago, humans had a much smaller population than mice. When the bulk of human evolution occurred in Sub-Saharan Africa, human population was small. As we'll see when we study population genetics, the larger the population size, the more mutations get introduced per generation.

Evolutionary Rates:

- Evolutionary rates are different in different regions of the genome.
- Evolutionary pressure is a major reason for this.
- There is evolutionary pressure to maintain important functions.
- If a mutation destroys some very important part of the genome, then the mutated individual will have a smaller number of expected viable progeny.
- Therefore, mutations that occur in functional regions of the genome tend to be purified away.
- This is why important regions of the genome are more likely to be conserved. Furthermore, one can identify potential genes by finding sequences of DNA that are conserved across species.

This explains why the mouse genome was sequenced for $300 million in 2001. Researchers could justify this cost because the mouse genome could help them identify human genes. In particular, researchers sequenced the mouse genome so they could align it to the human genome. Then, they identified genes by finding sequences of DNA that aligned best between the human genome and the mouse genome.

2 Sequence Alignment

Sequence Alignment: Given 2 sequences, we place gaps between letters of each sequence such that we maximize some notion of sequence similarity. Intuitively, matches between letters indicate more similarity between the sequences, while gaps and mismatches indicate less similarity.

But what notion of similarity should we use? Given 2 sequences, there are many different possible alignments, resulting in different numbers of matches, mismatches, and gaps. In lecture, we saw 3 different alignments of the 2 sequences AGGCTAGTT and AGCGAAGTTT:

```
AGGCTAGTT,
AGCGAAGTTT

AGGCTAGTT-          6 matches, 3 mismatches, 1 gap
AGCGAAGTTT

AGGCTA-GTT-         7 matches, 1 mismatch, 3 gaps
AG-CGAAGTTT


AGGC-TA-GTT-        7 matches, 0 mismatches, 5 gaps
AG-CG-AAGTTT
```

Notice that the 2nd alignment is better than the 1st alignment in terms of matches and mismatches, but it's worse in terms of gaps. The 3rd alignment is the best in terms of mismatches, but the worst in terms of gaps. Based on our preferences for matches, mismatches, and gaps, we might decide that different alignments are more similar than others.

Now we'll formalize our notion of similarity so that we can systematically decide which alignments are more similar than others. We'll count the # of matches, mismatches, and gaps for a given alignment. Then, we define the scoring function:

$$F = m \times (\#\text{ of matches}) - s \times (\#\text{ of mismatches}) - d \times (\#\text{ of gaps})$$

where:
m = score for a match
-s = score for a mismatch
-d = score for a gap
Note: we'll talk later in the course about how to actually set the parameters m, s, and d.

How to represent different alignments:

- We represent alignments using a matrix.
- An alignment in the matrix is a path from the top left corner of the matrix to the bottom right corner made up of moves down, right, and diagonally down to the right.

There are $>> 2^N$ possible alignments. We need an efficient way to compute the best alignment. Notice that alignment is additive. In particular:

- Suppose we take an alignment of sequences and score the complete alignment.

- Now suppose we split the alignment into a left side and a right side, and that we score each side of the alignment separately.
- These 2 scores will add up exactly to the score of the entire alignment. This is why we can apply **dynamic programming**.

Consider the sequences $x_1 \dots x_M$ and $y_1 \dots y_N$. Furthermore, consider an alignment of the substrings $x_1 \dots x_i$ and $y_1 \dots y_j$.

- There are MN such alignments of subsequences.
- Define a matrix (called "the dynamic programming table") F(i,j):

$$\text{F(i,j)} = \text{optimal score of aligning } x_1 \dots x_i \text{ with } y_1 \dots y_j$$

- Then the optimal score of aligning the full sequences x and y is given by F(M,N)
- Imagine a particular x_i and y_j. There are 3 possible cases of alignment:
  Case 1: x_i actually should align to y_j
  Case 2: x_i aligns to a gap and y_j aligns somewhere before x_i
  Case 3: y_j aligns to a gap and x_i aligns somewhere before y_j

We know the score for each of the 3 cases in terms of the matrix F for smaller indexes (i.e., in terms of matrix cells up and/or to the left of the current cell). Therefore, we can compute F(i,j) as follows.

Case 1: x_i aligns to y_j. The position i,j incurs either a mismatch penalty or gets a match score. We add this value to $F(i-1, j-1)$, which is the score of aligning $x_1 ... x_{i-1}$ to $y_1 ... y_{j-1}$:

$$F(i,j) = F(i-1, j-1) + \begin{cases} m & \text{if } x_i = y_j \\ -s & \text{otherwise} \end{cases}$$

Case 2: We incur a gap cost of -d. We also add the score of aligning $x_1 ... x_{i-1}$ to $y_1 ... y_j$. Then:

$$F(i,j) = F(i-1, j) - d$$

Case 3: Again, we incur a gap cost of -d. We also add the score of aligning $x_1 ... x_i$ to $y_1 ... y_{j-1}$. Then:

$$F(i,j) = F(i, j-1) - d$$

How do we know which of these 3 cases is correct? Answer: take the max.

We inductively assume that we know F(i,j-1), F(i-1,j), and F(i-1,j-1), all of which are optimal. Therefore, we select:

$$F(i,j) = max \begin{cases} F(i-1, j-1) + s(x_i, y_j) \\ F(i-1, j) - d \\ F(i, j-1) - d \end{cases}$$

where:

$$s(x_i, y_j) = \begin{cases} m & \text{if } x_i = y_j \\ -s & \text{otherwise} \end{cases}$$

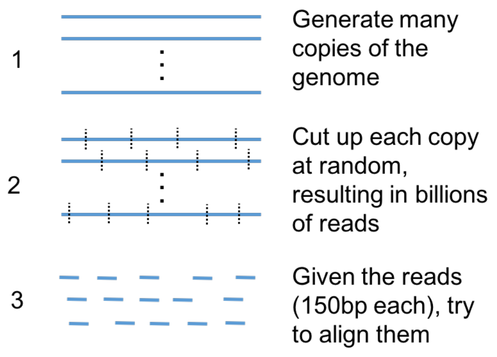
Base cases:
$F(0,0) = 0$ (aligning 2 empty strings has a score of 0)
$F(0,j) = -d \times j$ for j=1...N
$F(i,0) = -d \times i$ for i=1...M

- We can fill in the F matrix row-by-row, column-by-column, or diagonally (if we want to parallelize).
- We can reconstruct the optimal alignment by starting from the bottom-right of the DP table F and following the backpointers.
- The **time and space requirements for this algorithm are both O(NM)**. We need to store a matrix with NM elements, and the algorithm must do a constant amount of computations for each of the NM elements in the matrix.

3 Variant #1: Don't penalize gaps at the beginning & the end

A variation on the algorithm: we might decide that gaps at the beginning and end of either sequence don't matter and that they shouldn't be penalized. Why would we want to do this?

- When we sequence genomes, we take many copies of the original genome and cut them up at random.
- This results in many fragments of the genome.
- These short subsequences of about 150 base pairs each are called "reads" or "sequence reads".
- The following diagram illustrates the above procedure, and it's similar to a diagram that Professor Batzoglou drew on the whiteboard. (Please note that the image shown is not to scale. For example, the reads should actually be MUCH shorter than the genomes.)


1
Generate many copies of the genome
2
Cut up each copy at random, resulting in billions of reads
3
Given the reads (150bp each), try to align them

- We end up with billions of these reads. To make sense of the reads, we want to find reads that have a high degree of overlap.
- This is called the overlap detection problem, which is a variant of the sequence alignment problem.
- We can solve overlap detection using the following 2 changes to the sequence alignment algorithm:

1) For all i,j set F(i,0)=0 and F(0,j)=0

2) $F_{OPT} = \max \begin{cases} \max_i F(i,N) \\ \max_j F(M,j) \end{cases}$

The 2nd change means that we don't necessarily start backtracking from the bottom right corner of the F matrix. Instead, we find the largest score in either the last row or the last column, and we backtrack from this location. Furthermore, we may not backtrack all the way to the top left corner of the F matrix. Instead, we may terminate somewhere else along the 1st row or 1st column of F.

4 Variant #2: Local Alignment

Local alignment problem: If we're given 2 sequences, we might not want a global alignment. Instead, we may just want the optimal alignment between any 2 substrings of the complete sequences.

Why is the local alignment problem relevant?
Answer: In actual biological sequences, we do not usually look for similarity across the entire genome. The genomes of different species are "garbled around". So when you compare different genomes, you actually want to look for local alignments between parts of chromosomes.

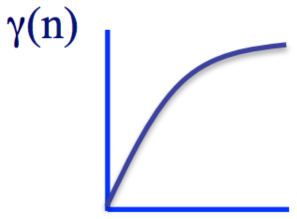
The idea behind the local alignment problem is ignoring badly aligning regions. First, we add an extra case to our dynamic programming iteration:

$$F(i,j) = max \begin{cases} 0 \\ F(i-1,j-1) + s(x_i, y_j) \\ F(i-1,j) - d \\ F(i,j-1) - d \end{cases}$$

To terminate, we search for the maximum value in the entire matrix: $F_{OPT} = \max_{i,j} F(i,j)$. To output the optimal local alignment, we backtrack from the cell containing F_{OPT} until we reach a cell containing a value of 0. The **time and space requirements for this algorithm are both still O(NM).**

5 Scoring the Gaps More Accurately

- Next, we covered how to score the gaps in a more sophisticated way.
- In biological sequences, gaps come in bunches. Therefore, if you encounter a gap of length L, it's not a very good approach to score this gap as $-L \times d$.
- Instead, you'd want each additional gap penalty to take a smaller penalty than the previous one.
- This means you'd want some kind of concave gap penalty.
- The lecture slides show the following plot of a concave gap penalty function $\gamma(n)$ (where n is the size of the gap):



- Concave gap penalty functions satisfy the property:

$$\gamma(n+1) - \gamma(n) \leq \gamma(n) - \gamma(n-1) \; \forall n$$

- In words, this property means that in a sequence of consecutive gaps, each additional gap position takes a smaller penalty than the previous gap.
- **Space** required for convex gap dynamic programming: O(NM) (same as before)
- **Running time** for convex gap dynamic programming: $O(N^2M)$ (assuming N>M)
- The cubic runtime is caused by the fact that we need to consider all possible gap lengths at each step of the algorithm.

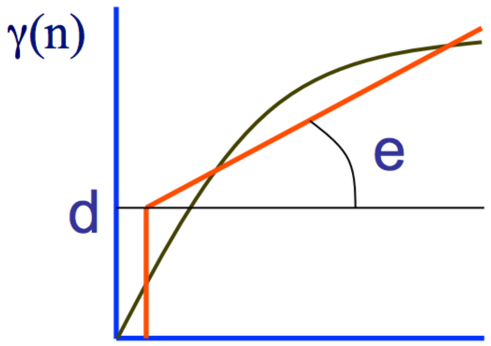
- In other words, we need to do a linear amount of work at each position of the F matrix.

In order to avoid this cubic runtime, the compromise is to use an **affine gap** penalty as follows:

- The 1st gap incurs a penalty of d (i.e., the score of the 1st gap is -d).
- Every subsequent gap incurs a penalty of e (where e < d).
- Mathematically, an affine gap penalty function takes the form:

$$\gamma(n) = \underset{\substack{|\\ \text{gap}\\ \text{open}}}{d} + (n-1)\times \underset{\substack{|\\ \text{gap}\\ \text{extend}}}{e}$$

- The lecture slides show the following plot of an affine gap penalty function (shown in red) superimposed over the concave gap penalty we saw before:



Note: most alignments today use affine gap scoring functions.

Implementing sequence alignments with affine gaps:

- When we optimize F(i,j), we must consider the 2 gap possibilities (inserting a gap to either the x or y sequence).
- However, we don't know if this gap is the 1st gap ("opening a gap") or if it's extending a previous gap.
- These 2 different cases have different scores (either -d or -e), but we're not keeping track of enough information to tell which case we're in.
- We solve this issue by defining multiple dynamic programming matrices (F, G, H, and V) to keep track of the different possibilities.

Informally, these matrices can be described as follows:

- F(i,j) is the optimal score of aligning $x_1 ... x_i$ to $y_1 ... y_j$ ***if*** $\mathbf{x_i}$ **actually aligns to** $\mathbf{y_j}$.
- G(i,j) is the optimal score of aligning $x_1 ... x_i$ to $y_1 ... y_j$ ***if*** $\mathbf{x_i}$ **aligns to a gap after** $\mathbf{y_j}$ (so y_j aligns to some position before x_i).
- H(i,j) is the optimal score of aligning $x_1 ... x_i$ to $y_1 ... y_j$ ***if*** $\mathbf{y_j}$ **aligns to a gap after** $\mathbf{x_i}$ (so x_i aligns to some position before y_j).

- V(i,j) represents the best score of aligning $x_1 ... x_i$ to $y_1 ... y_j$. In other words, V(i,j) is the max of F(i,j), G(i,j), and H(i,j).

Mathematically, these matrices can be expressed as follows:

$$V(i,j) = \max\{F(i,j), G(i,j), H(i,j)\}$$

$$F(i,j) = V(i-1,j-1) + s(x_i, y_j)$$

where $s(x_i, y_j)$ is the same as it was previously.

$$G(i,j) = \max \begin{cases} V(i-1,j) - d \\ G(i-1,j) - e \end{cases}$$

Recall that G(i,j) represents the scenario when x_i aligns to a gap after y_j. The 1st case inside the max operator represents opening a new gap, while the 2nd case represents extending a previously opened gap. Note that if $V(i-1,j) = G(i-1,j)$, then the 2nd case will always be chosen since $e < d$. In other words, the above equation is consistent with the fact that *if* x_{i-1} aligned to a gap *and* x_i aligns to a gap, then the gap score for x_i should be -e. H(i,j) is expressed similarly:

$$H(i,j) = \max \begin{cases} V(i,j-1) - d \\ H(i,j-1) - e \end{cases}$$

Initialization when using an affine gap penalty:
$V(i,0) = -d - (i-1) \times e$
$V(0,j) = -d - (j-1) \times e$

- **Space** required when using affine gap penalty: O(NM)
- **Running time** when using affine gap penalty: O(NM)

Now suppose we want to use a piecewise linear gap penalty. This penalty would be similar to the affine penalty function, except that it would have extra line segments. In particular, a gap that is preceded by a large number of consecutive gaps would incur a penalty smaller than e. We could use a similar approach as above, but we'd need 2 extra matrices per extra line segment. Asymptotically, the space and running time requirements would still both be O(NM), because we only need to add a constant number of matrices.

6 Bounded Dynamic Programming

- Sometimes, we may have sequences that are extremely similar to one another.
- For example, we may be preparing reads of the same genome.
- In this case, it's wasteful to compute the entire dynamic programming matrix.
- Instead, we may expect that the optimal alignment lies close to the diagonal.
- In this case, we can fix some "radius" k around the diagonal in the DP matrix, and ignore everything around it.
- **Time & space requirements** are both: $O(N \times k(N)) << O(N^2)$
- Thus, bounded dynamic programming is much more efficient at aligning sequences

7 Linear Space Alignment

- So far, we've described algorithms that require quadratic time and space (except for bounded DP, which requires domain-specific conditions).
- When sequences are large, quadratic space requirements can become an issue.
- We want to find a global alignment algorithm that only requires a linear amount of space.
- Next lecture, we'll see a divide-and-conquer algorithm to solve this problem.