Computational Molecular Biology

Last updated: September 12, 2024

Hello! These lecture notes were originally created during the 2018 iteration of CS 181 (Computational Molecular Biology) taught at Brown University by Sorin Istrail. The original notes were provided by Shivam Nadimpalli, and future updates have been made by course TAs including Elliot Youth, Daniel Ben-Isvy, Sam Maffa, Iris Huang, and Hannah Beakley. Please email any comments or typos to cs1810tas@lists.brown.edu.

Contents

1 Sequence Alignment

(Sept. 10)

In this chapter of the course, we will take a look at those algorithms that revolutionized the field of computational biology, and laid down robust foundations for its future.

We will first look at the problem of global alignment, after which we will consider the statistical foundations of this topic, and then finally we'll study local and gap alignment. The algorithmic technique of dynamic programming will play a key role throughout. Let's get started.

1.1 Global Alignment

Consider two DNA sequences $X =$ `ATTACG` and $Y =$ `ATATCG`. What's the optimal way to *align* these two sequences?

We haven't formally defined what any of this means, but going off our intuition, we would guess that the fewer mismatches an alignment has, the better it is. It's easy to see that the best alignment for these two sequences, then, is:

```
ATTA-CG
|||||||
A-TATCG
```

The problem of *global alignment* of two sequences is that of finding an alignment for every residue in each sequence.

Some foreshadowing: we will soon be looking at *local* alignment, in which we look for the best alignment of any subsequences of the given sequences. The motivation behind this, as well as how it differs from global alignment, will be made clear then.

1.1.1 Formal setup

However, what if a `(T, -)` alignment is really bad for us for some reason? Or if we want to minimize the number of `(A, C)` alignments? We can do so by penalizing such undesirable alignments, and this can be done by choosing an appropriate scoring scheme.

Given two strings over a finite alphabet Σ, a *scoring scheme* is a function $\delta : \Sigma \times \Sigma \to \mathbb{R}$. In the case of DNA sequences, $\Sigma = \{\texttt{A}, \texttt{T}, \texttt{G}, \texttt{C}\}$, and in the case of protein sequences Σ would be the set of 20 amino acids.

It's usually convenient to represent a scoring scheme in the form of a table. For example:

	A	T	C	G
A	1	0	0	0
T	0	1	0	0
C	0	0	1	0
G	0	0	0	1

Figure 1. The Unitary or Diagonal Scoring Scheme.

We also introduce a related concept: the *gap penalty* is a function $\delta : \Sigma \to \mathbb{R}$ which tells you the cost for aligning a character with a gap (-). Gaps are also called *indels* by biologists, which stands for "insertion/deletion mutations".

Abuse of Notation: We will use δ to refer to the scoring scheme and the gap penalty.

Finally, suppose we have an alignment $x_1 \dots x_n$ and $y_1 \dots y_n$ where $x_i, y_i \in \Sigma \cup \{\texttt{-}\}$, then the *score* of the alignment is given by $\Sigma_{i=1}^{m} \delta(x_i, y_i)$.

For example, under the unitary scoring scheme (Figure 1) with 0 gap penalty, the alignment of X and Y given in §1.1:

```
A   T   T   A   -   C   G
|   |   |   |   |   |   |
A   -   T   A   T   C   G
-----------------------------
1 + 0 + 1 + 1 + 0 + 1 + 1  =  5
```

has a score of 5.

If we change the gap penalty to -1, then the score of the same alignment changes to 3. From this, it's clear that you can get different scores for the same alignment under different scoring schemes, and that the optimal alignment of two sequences depends on the scoring scheme being used.

Finally, we can formally state the problem of global alignment:

> The optimal global alignment of two sequences under a chosen scoring scheme and gap penalty is the alignment of the two sequences with maximum score.

Now, how would one find this optimal global alignment? One way is to enumerate all possible alignments of the given sequences, and pick the one with the best score. But we'll see in the homework that this process is computationally inefficient. The principle of *dynamic programming*, however, comes to our rescue, as we will see in §1.1.3.

1.1.2 The edit graph

Let Σ be a finite alphabet, and let $X = x_1 \dots x_m$ and $Y = y_1 \dots y_n$ be strings over Σ. To X and Y, we associate a directed graph termed the *edit graph* of X and Y.

The vertices of the graph are the vertices of the rectangular grid of size $(m+1) \times (n+1)$. By $v(i, j)$ we will mean the vertex in the grid corresponding to the $(i+1)^{\text{th}}$ row and the $(j+1)^{\text{th}}$ column.

Between the vertices, we have three kinds of edges:

1. A *gap-in-Y* edge: for $1 \le i \le m, 1 \le j \le n$, we have an edge $v(i-1, j) \to v(i, j)$. This edge corresponds to $(x_i, \texttt{-})$ in the alignment.
2. A *gap-in-X* edge: for $1 \le i \le m, 1 \le j \le n$, we have an edge $v(i, j-1) \to v(i, j)$. This edge corresponds to $(\texttt{-}, y_i)$ in the alignment.
3. An *alignment* edge: for $1 \le i \le m, 1 \le j \le n$, we have an edge $v(i-1, j-1) \to v(i, j)$. This edge corresponds to (x_i, y_j) in the alignment.

What exactly does this graph tell us? This will become clear through the following example:

Example 1.1.1. Suppose we want to construct an edit graph for two sequences `AC` and `AGC`. We set up a grid of dimensions 3×4 as shown in the picture below:

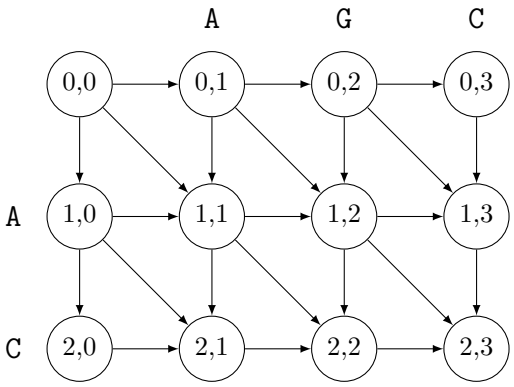

Figure 2. Edit Graph for `AC` and `AGC`.

Let's try looking at a path in this graph. Say we have the following sequence of vertices:

$$v(0,0) \to v(1,1) \to v(1,2) \to v(2,3)$$

Note that it can be interpreted as the following alignment:

```
A-C
AGC
```

Similarly, suppose we're given an alignment:

```
AC--
A-GC
```

We can construct a directed path in the graph:

$$v(0,0) \to v(1,1) \to v(2,1) \to v(2,2) \to v(2,3)$$

We come to the crucial insight which will allow us to develop an algorithm for global alignment:

> Given two sequences X and Y, there is a 1-to-1 correspondence between directed paths from $v(0,0)$ to $v(m,n)$ in their edit graph and their alignments.

We will explore this idea further in the next section.

1.1.3 The Needleman-Wunsch algorithm

(Sept. 12)

Recall that any path through the edit graph represents a sequence of matches, mismatches, and indels that uniquely represents one possible alignment of the two sequences. It's worth noting that in grid corresponding to the edit graph, we index from 0.

The fundamental concept of Needleman and Wunsch was that to calculate the optimal alignment score, one would only need to calculate and enumerate all the ways that an aligned pair can be added to a shorter segment to produce an alignment of an additional position in the sequences. If one always chooses the highest possible score for the extension, the highest score accumulated in the end is the global optimal score. And the path taken during the summation of the optimal score is the optimal alignment.

1.1.4 The main algorithm

Suppose we have two sequences X, Y over Σ of lengths m and n respectively. By X_i we will represent the i^{th} character of X.

We initialize a table of size S of dimensions $(m+1) \times (n+1)$. By $S_{i,j}$ we will represent the element in the $(i,j)^{\text{th}}$ cell of our table S.

function GLOBAL ALIGNMENT($X \in \Sigma^m, Y \in \Sigma^n$)
  $S_{0,0} \leftarrow 0$
  for $i \in \{1, 2, \ldots, m\}$ **do**
    $S_{i,0} \leftarrow S_{i-1,0} + \delta(X_i, \text{-})$
  end for
  for $j \in \{1, 2, \ldots, n\}$ **do**
    $S_{0,j} \leftarrow S_{0,j-1} + \delta(\text{-}, Y_j)$
    for $i \in \{1, 2, \ldots, m\}$ **do**

$$S_{i,j} \leftarrow \max \begin{cases} S_{i-1,j-1} + \delta(X_i, Y_j) \\ S_{i-1,j} + \delta(X_i, \text{-}) \\ S_{i,j-1} + \delta(\text{-}, Y_j) \end{cases}$$

    end for
  end for
  return $S_{m,n}$
end function

1.1.5 How to obtain the optimal alignment from this table?

Obviously for many applications, we don't just want to know what the score of the optimal alignment of two sequences is, but also what the alignment that achieves this score is.

There are several strategies for obtaining the optimal alignment:

- One solution is to fill out the table in the algorithm above, and then starting at the bottom right cell of the matrix, and recalculate the max that lead to the cell's value. This time, rather than just taking the max, record the cell from which the max originated (resolving ties arbitrarily: optimal alignments aren't unique in general).

  In other words, we determine whether the term of the recurrence originating from an X-gap, a Y-gap, or a match/mismatch was optimal, and backtrace to the corresponding position.

- An alternative (and computationally more efficient) solution is to place backpointers as we're computing a cell entry, indicating which cell it came from. And then we can follow these backpointers from $S_{m,n}$ to $S_{0,0}$ to obtain the optimal alignment.

1.1.6 Complexity of Needleman-Wunsch

We first look at the time complexity of the above algorithm. Indeed, we have:

- Line 2 requires 1 operation.
- Line 4 requires 3 operations, and is repeated m times.
- Line 7 requires 3 operations, and is repeated n times.
- Line 9 requires 9 operations, and is repeated mn times.
- Line 12 is a single operation.

Thus, Needleman-Wunsch runs in $\mathcal{O}(1 + 3m + 3n + 9mn + 1) = \mathcal{O}(mn)$ time. If $m \sim n$, then the runtime is $\mathcal{O}(n^2)$, and so this is a *quadratic* algorithm.

Note that it also requires $\mathcal{O}(mn)$ space, which is the size of our table.[1]

1.1.7 BLOSUM Matrices

As stated in the introduction in §1.2, we want to account for the probability of finding two different amino acids together during sequence alignment using empirical data. In order to do so, we use the technique of *substitution matrices* to assign match and mismatch scores more meaningfully.

The most popular type of substitution matrices are the BLOSUM (BLOcks SUbstitution Matrices) matrices introduced by Henikoff and Henikoff (1992), so named because they are created from data in the BLOCKS database. Instead of a formal construction, we'll look at a concrete example.

Let us align strings over the alphabet $\Sigma = \{\text{A, B, C }\}$. A *block* is just a fixed region in a given set of aligned sequences. For example, suppose that from some species whose proteins encoded over Σ, we obtain six sequences of the same protein that is 4 residues long from 6 different organisms. This gives us a block as follows:

Organism 1	B	A	B	A
Organism 2	A	A	A	C
Organism 3	A	A	C	C
Organism 4	A	A	B	A
Organism 5	A	A	C	C
Organism 6	A	A	B	C

Figure 3. Sample block.

We want to figure out what evolutionary information the block above encodes, so as to improve on our scoring scheme for sequence alignment.

Now, in our block above, we have a total of $6 \times 4 = 24$ residues. Of these 24 residues, we have 14 As, 4 Bs, and 6 Cs.

[1] However, it is possible to reduce the space usage to $\mathcal{O}(m + n)$ using Hirschberg's divide-and-conquer algorithm.

As we have 6 letters per column, there's $\binom{6}{2} = 15$ ways of picking a pair of letters in each column. As there are 4 columns, we have $15 \times 4 = 60$ possible ways of picking a pair of letters at the same position in the strings (i.e. in the same column) from the above table.

The table below lists the frequencies of these possible alignment pairs that we observe in the table above:

Aligned Pair	**Observed Frequency**
A to A	26/60
A to B	8/60
A to C	10/60
B to B	3/60
B to C	6/60
C to C	7/60

Here's an example of how the entries in the table above were computed: note that the only "B to B" alignments in the table above are in the third index in our sequences, and we have three such instances. One is between Organism 1 and Organism 4, the second is between Organism 1 and Organism 6, and the third one is between Organism 4 and Organism 6.

Now that we've gotten some observed data, let's see how it lines up with our expected data. As we have A occurring in our sequences with probability 14/24, B with probability 4/24, and C with probability 6/24 (see 2 paragraphs before table above), we should have an "A to A" alignment with probability $14/24 \times 14/24$.

Similarly, we must have an "A to B" alignment with probability $2 \times 14/24 \times 4/24$ and so on. We can thus obtain expected frequencies.

We'll use the log-likelihood ratio f of each possible aligned pair in order to compare these two frequencies. This value is defined as:

$$f = 2 \times \log_2 \left(\frac{\text{Observed frequency}}{\text{Expected frequency}} \right)$$

Let's put everything together:

Aligned Pair	**Observed frequency**	**Expected Frequency**	f
A to A	26/60	196/576	0.70
A to B	8/60	112/576	-1.09
A to C	10/60	168/576	-1.61
B to B	3/60	16/576	1.70
B to C	6/60	48/576	0.53
C to C	7/60	36/576	1.80

Figure 4. BLOSUM matrix scores.

How does this help us in our alignment algorithm in any way? Indeed, we can use a scoring scheme taking the value of f in the table above, and this is a statistically-justified scoring scheme.

For example, if the observed frequency of a substitution between A and B is less than the expected frequency, then $\delta(\text{A}, \text{B}) < 0$ as seen above. Namely, we penalize for an A-B alignment, and we can do so because from our observed data, this alignment doesn't really happen that frequently.