7.1 Affine Gapped Alignment in Action

Here a brief summary of the affine gapped alignment alignment is given, and an example initialization, population, and backtrace are given. Please consult the handout available on the course website for a more detailed explanation of the process.

7.1.1 Recurrence Relationship

Recall the recurrence relationship for affine gap alignment.

\[
V_{i,j} = \max \left( E_{i,j}, F_{i,j}, G_{i,j} \right)
\]

\[
G_{i,j} = \begin{cases} x_i = y_j : & V_{i-1,j-1} + \alpha \\ x_i \neq y_j : & V_{i-1,j-1} - \beta \end{cases}
\]

\[
E_{i,j} = \max \left( E_{i,j-1} - \tau, V_{i,j-1} - \gamma - \tau \right)
\]

\[
F_{i,j} = \max \left( F_{i-1,j} - \tau, V_{i-1,j} - \gamma - \tau \right)
\]

A description of each term of this recurrence is given in Figure 7.1.

When aligning strings of lengths \(n\) and \(m\), rather than writing out four separate matrices, it’s often easier to visualize as a single \((n + 1) \times (m + 1)\) matrix with 4 entries per cell. In this document, the position and color of an entry determines whether it corresponds to \(V, G, E,\) or \(F\).

In the diagram to the left, each of the arrows corresponds to one of the dependencies in the recurrence. The arrow emanating from \(G\) represents a match or mismatch, depending on whether the characters at a given position match. The arrows from \(E\) and \(F\) correspond to opening and extending gaps. Finally the arrows from \(V\) simply take the maximum over the other three matrices within each cell.

Figure 7.1: Explication of affine gapped alignment recurrence.
7.1.2 Base Cases and Initialization

The above recurrence covers the general case of affine gapped alignment, but we need to be careful about how we initialize the dynamic programming matrices. Specifically, it is not possible to extend a gap that does not exist, so we need to perform some special initialization. The base cases look like this:

\[
\begin{align*}
V_{0,0} & \leftarrow 0 \\
G_{i,0}, G_{0,j} & \leftarrow -\infty \\
F_{0,j}, E_{i,0} & \leftarrow -\infty
\end{align*}
\]

Now, suppose we wish to align the following pair of strings:

\[
x = \text{AAG} \quad y = \text{ATATTG}
\]

Applying these rules, we obtain the following dynamic programming matrices after initialization. In these matrices, if a value is not written, it is either not yet defined or \(-\infty\). We initialize the first row and column, each with values representing extending a single gap from the 0th character to the end of each string, both without using a single character from the other string.

7.1.3 Completing the alignment

After initialization, we apply the ordinary recurrence relation to obtain the result in each cell. Below, the dynamic programming matrices are given, along with the completed backtrace for the given strings. Consult the handout for intermediate steps and an explanation of this process in much greater detail.
As we have discussed in previous lectures, identifying the maximum similarity score is equivalent to finding the highest-weight or longest path in the edit graph. In general, finding the longest simple path through a graph is NP-hard, but in a directed acyclic graph, the problem is equivalently difficult to finding the shortest simple path.

To see this, consider first that a DAG contains no cycles: once a path reaches a node, there is no way to return to this node. For this reason, we have no issues with infinite weight paths and negative cycles. Then, we can simply negate the weight of each edge, and the shortest path through the negated graph is the longest path in the unnegated graph. See Figure 7.2 for an illustration of this concept.

Dijkstra’s Algorithm is guaranteed to find the shortest path through any graph that does not contain any negative-weight edges. This is not directly applicable when we have negative similarity scores, but we can still use this algorithm to solve global alignment problems with the following trick.
Suppose we wish to identify the optimal global alignment for any strings \(x, y\) with scoring matrix \(\delta\). We may some \(\delta'\) with some constant \(c\) such that:

\[
\delta'(a, b) = \begin{cases} 
    a, b \in \Sigma & : \delta(a, b) + 2c \\
    a \text{ or } b \text{ is a gap} & : \delta(a, b) + c
\end{cases}
\]

We may select \(c\) such that each value in the range of \(\delta\) is negative, and then the negated edit graph contains only positive edges, allowing us to use Dijkstra’s Algorithm to solve for the longest path.

To see that the alignments will be the same, consider that, for strings of lengths \(m\) and \(n\), supposing WLOG that \(m \leq n\), there are exactly \(a\) aligned characters and \(b\) gaps, such that \(a \in \{0, 1, 2, \ldots, m\}\), and \(b = n - m + 2(m - a)\). Any alignment produced using \(\delta'\) will have the score of the alignment produced using \(\delta\), plus an additional \(2ca\) for aligned characters and \(cb\) for gap characters.

Substituting these values in, we see that \(2ca + c(n - m + 2(m - a)) = 2ca + cn - cm + 2cn - 2ca = c(n + m)\). This value does not depend on \(a\) and \(b\) (only on the lengths of \(x\) and \(y\), thus we may conclude that the score of any alignment produced using \(\delta\) differs from that produced by \(\delta'\) differs only by a constant amount.

In local alignment, we can’t use this trick to ensure a graph with nonnegative edge weights. In order to find the shortest path in a DAG, we can use the Bellman-Ford algorithm, which runs in \(O(nm(n + m))\) time on a graph of diameter in \(O(n + m)\). Note that this time complexity is inferior both to that of Dijkstra’s algorithm (which runs in \(O(nm \log(nm))\) time), and our previous dynamic programming solution.

Fortunately, we can leverage the fact that we have a DAG to obtain a more efficient algorithm. In a DAG, we can use topological sorting, where we sort the vertices of the graph according to the partial ordering \(a < b \iff b\text{ is reachable from } a\). The existence of this partial ordering is a property of the DAG: if the edit graph contained cycles, such an ordering would not exist.

In the context of alignment, this is very easily interpretable: any node \(N_{i,j} < N_{k,l}\) implies that \((i, j) \neq (k, l), k \geq i, \text{ and } l \geq j\).

To topologically sort a graph, we require \(O(V + E)\) time, and to identify the shortest path in a topologically sorted graph, we also require \(O(V + E)\) time. In a two-dimensional edit graph, there are \(O(nm)\) vertices, and each vertex is associated with no more than 3 edges, thus there are \(O(nm)\) edges as well. We may therefore conclude that identifying the shortest path in a DAG via topological sorting is possible in \(O(nm)\) time: exactly the same time complexity as by our dynamic programming algorithm!

With the above derivations out of the way, let us now take a moment to appreciate the significance of the result. We have algorithms for computing the global, local, general gapped, and affine gapped alignments of any pair of sequences, and we should at least have a good idea of how to generalize these algorithms to operate over more than two input sequences. However, by examining the problem from a graph theoretic perspective, we can also convert the problem into a DAG, and then apply an existing algorithm (shortest path via Topological Sort) to solve the problem.

Furthermore, since the edit graphs for global, local, and affine gapped alignment all contain \(O(nm)\) edges and vertices, we can obtain the same time complexity using this technique as with our dynamic programming solutions.

If we approach the problem in this manner, our solutions are theoretically simpler and our implementations (generally) require less code. On the theoretical side, we only need to prove that our algorithm produces the
correct edit graph and that the longest path through the edit graph is equivalent to the maximum similarity alignment. As for implementation, we need only write a program that translates the recurrence relationships into edit graphs: the work of initializing and filling the dynamic programming matrix is subsumed by the longest path algorithm. Backtracing too becomes simpler: we need only translate the longest path into an alignment.