Disclaimer: These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.

25.1 Sparse Dynamic Programming

Sparse dynamic programming is an useful tool for computing alignments of long sequences. The basic idea is to first compute the best local alignments between the sequences and then to use dynamic programming on the local alignments instead of on the individual elements of the sequences. Galil, Myers, and Eppstein have done work on using this strategy for alignment.

25.2 Problem Statement

Given two sequences, \( A \) and \( B \), let \( f = A[f.b.x \ldots f.e.x] \times B[f.b.y \ldots f.e.y] \) denote a fragment, or a local alignment between \( A \) and \( B \). Associated with each fragment \( f \) is a score, \( f.score \), corresponding to the score of the local alignment of the fragment. For two fragments, \( f \) and \( g \), let \( f < g \) if and only if \( f.e.x \leq g.b.x \) and \( f.e.y \leq g.b.y \). Given \( F \) fragments, we wish to find a chain of fragments \( f_1, f_2, \ldots, f_L \), such that \( f_1 < f_2 < \ldots < f_L \) and the following score is maximized:

\[
score_{\text{chain}} = \sum_i f_i.score - \mu \sum_{i>1} [(f_i.b.x - f_{i-1}.e.x) + (f_i.b.y - f_{i-1}.e.y)] - \mu(f_1.b.x + f_1.b.y) - \mu((M - f_L.e.x) + (N - f_L.e.y))
\]

where \( N = |A|, N = |B| \), and \( \mu \) is a penalty factor for the distances in between the adjacent fragments in the chain.

25.3 \( O(F^2) \) Algorithm

Pearson and Lipman came up with an \( O(F^2) \) algorithm for this problem in 1984. It is a dynamic programming algorithm that for each fragment \( f \), computes the value \( \text{Best}(f) \), which is equal to the score of the best chain up to and including \( f \). The dynamic programming recurrence is

\[
\text{Best}(f) \leftarrow \max \left\{ \text{score} - \mu(f.b.x + f.b.y) \max_{g \leq f} \text{Best}(g) + \text{score} - \mu(f.b.x - g.e.x) + (f.b.y - g.e.y) \right\}
\]

and after \( \text{Best}(f) \) has been calculated for all \( f \), the best chain is found by

\[
\text{Best} \leftarrow \max_{f} (\text{Best}(f) - \mu((M - f.e.x) + (N - f.e.y)))
\]
At the time that $Best(f)$ is calculated, the values of $Best(g), \forall g < f$ must have already been calculated. To this end, it is sufficient to consider the fragments in increasing value of $f.b.y$.

25.4 $O(F \log F)$ Algorithm

We can improve the complexity of the Pearson and Lipman algorithm by not explicitly considering every fragment $g, g < f$ when computing $Best(f)$. We will use a data structure that for each $x$, gives the best fragment to chain from if we connect to a fragment starting at $x$. At given scan line ($y$ value) in the recursion, we can plot, versus $x$, the possible scores obtained by chaining to one of the already processed fragments. At scan line $y$, all fragments $f$ with $f.e.y \leq y$ contribute a line to this plot starting at $(f.e.x, Best(f) - \mu(y - f.e.y))$ and continuing to the right with slope $-\mu$. Note that advancing the scan line changes the heights of these lines, but does not change the their relative order. When we consider a fragment $g$, the best fragment to chain to is the fragment that has the highest line in the plot at $x = g.b.x$. Therefore, all we need to do is keep track of the max envelope of this plot. Determining the best fragment to chain to for a given $x$ amounts to doing a binary search over this envelope. Using an appropriate data structure, we can perform finding, inserting, and deleting with the envelope in $\log(F)$ time. Because each fragment $f$ is considered for insertion into the envelope once, and $Best(f)$ is computed once, the total time for the algorithm is now $O(F \log F)$.

As the scan line advances, the algorithm now must deal with two events. The first event is the computation of $Best(f)$ when the start point of a fragment $f$ is hit at $(f.b.x, f.b.y)$. The second event is the (possible) insertion of $f$ into the max envelope when the end point $(f.e.x, f.e.y)$ is reached. We will maintain the envelope as a list $E = [f_1, f_2, \ldots, f_k]$ of the fragments that make up the envelope, with $f_1.e.x < f_2.e.x < \ldots < f_k.e.x$. Let $Find(L, x) = \min j : L[j].e.x \geq x$ (returning 0 if there is no such $j$). Also, let $f \gg g$ denote that the fragment $f$ dominates the fragment $g$ in the max envelope. A fragment $f$ dominates $g$ if
Figure 25.1: A plot of the scores that can be obtained by chaining a newly considered fragment $f$ to a fragment that has already been processed. The dotted line indicates the max envelope. The point at which the vertical line hits the max envelope determines the best fragment to which $f$ should be chained.

$$\text{Best}(f) - \mu[(g.b.x - f.b.x) - f.b.y] > \text{Best}(g) + \mu g.b.y$$

The algorithm is roughly as follows:

$$E \leftarrow \emptyset$$

**foreach** event in order of $y$ **do**

  if event = begin $f$

    $j \leftarrow \text{Find}(E, f.b.x)$

    if $j = 0$

      $\text{Best}(f) \leftarrow f.score - \mu(f.b.x + f.b.y)$

    else

      $g \leftarrow \text{E}[j]$

      $\text{Best}(f) \leftarrow \text{Best}(g) + f.score - \mu((f.b.x - g.e.x) + (f.b.y - g.e.y))$

  else if event = end $f$

    $j \leftarrow \text{Find}(E, f.e.x)$

    if $j = 0$ or $f \gg E[j]$ **then** Insert(E, j, f)

    $j \leftarrow j + 1$

  **while** $j < \text{len}(E)$ **and** $E[j] \gg E[j + 1]$

  $\text{Delete}(E, j + 1)$