CS 466 – Introduction to Bioinformatics
Lecture 7

Mohammed El-Kebir

September 16, 2020

Document history:

- 10/3/2018: Initial version
- 10/17/2018: Fixed incorrect running time of tree alignment
- 10/17/2018: Typos
- 10/25/2018: Typos
- 9/18/2019: Removed "Tree and Star Alignments" section
- 9/16/2020: 'Carrillo' $\longrightarrow$ 'Carrillo'

Contents

1 Problem Statement

Let Σ be the alphabet. We are given k strings $\mathbf{v}_1, \ldots, \mathbf{v}_k \in \Sigma^*$. A *multiple alignment* $A = [a_{p,i}]$ is defied as an $k \times \ell$ matrix where $\ell \in \{\max_{p \in [k]}\{|\mathbf{v}_p\}, \ldots, \sum_{p=1}^{k} |\mathbf{v}_p|\}$ such that (i) each entry $a_{p,i}$ is a character from the gap-extended alphabet $\Sigma \cup \{-\}$, (ii) removal of the gap characters from each row $\mathbf{a}_p$ yields input string $\mathbf{v}_p$ and (iii) there is no column $j \in [\ell]$ consisting of only gap characters in A, i.e. $a_{p,j} = -$ for all $p \in [k]$.

We consider the *Sum-of-Pairs (SP) score* $\mathrm{SP}(A)$, which uses a given pairwise scoring function $\delta : (\Sigma \cup \{-\}) \times (\Sigma \cup \{-\}) \to \mathbb{R}$ to score every column of an alignment A by considering all pairs of input sequences. Specifically, $\mathrm{SP}(A)$ is defined as

$$\mathrm{SP}(A) = \sum_{p=1}^{k} \sum_{q=p+1}^{k} \sum_{i=1}^{\ell} \delta(a_{p,i}, a_{q,i}). \tag{1}$$

We have the following two problems.

Problem 1. WEIGHTED SP-EDIT DISTANCE *Given strings* $\mathbf{v}_1, \ldots, \mathbf{v}_k \in \Sigma^*$ *and a scoring function* $\delta : (\Sigma \cup \{-\}) \times (\Sigma \cup \{-\}) \to \mathbb{R}$*, find a multiple alignment* A *such that* $\mathrm{SP}(A)$ *is minimum.*

Problem 2. SP-GLOBAL ALIGNMENT *Given strings* $\mathbf{v}_1, \ldots, \mathbf{v}_k \in \Sigma^*$ *and a scoring function* $\delta : (\Sigma \cup \{-\}) \times (\Sigma \cup \{-\}) \to \mathbb{R}$*, find a multiple alignment* A *such that* $\mathrm{SP}(A)$ *is maximum.*

Observe that the two problems differ only in the direction of their objective functions, minimization vs. maximization.

2 Carrillo-Lipman Algorithm

We consider the WEIGHTED SP-EDIT DISTANCE problem. Extending results from previous lectures, an optimal alignment for this problem is a shortest path from source $(0, \ldots, 0)$ to sink $(|\mathbf{v}_1|, \ldots, |\mathbf{v}_k|)$ in the edit graph. Thus, we could identify this shortest path using a shortest path algorithm. In particular, if the cost function δ assigns non-negative costs (which is the common case for this problem), we could use Dijkstra's algorithm.

Dijkstra's algorithm maintains a priority queue of unvisited vertices, where each vertex v has a priority $p(v)$ corresponding to the length of the shortest path computed thus far from the source vertex v_0 to v. Initially, the queue contains only the source vertex v_0 with priority $p(v_0) = 0$. The priority of the other vertices $v \neq v_0$ is set to $p(v) = \infty$. We pop a vertex v from the queue with lowest priority $p(v)$, and then update the priorities of its unvisited neighboring vertices w. Specifically, we set $p(w) := p(v) + \delta(v, w)$ and $\pi(w) = v$ if $p(w) > p(v) + \delta(v, w)$, and add w to the queue if it had not been there already. Next, we mark v as visited and repeat the procedure until the queue is empty. Each value $\pi(v)$ indicates the parent vertex of w on the shortest path from v_0 to v. (Note that the above description differs slightly from the original formulation of Dijkstra's algorithm, where the queue contains all vertices initially.)

This algorithm will identify the optimal alignment when run on the edit graph of an instance of WEIGHTED SP-EDIT DISTANCE with non-negative costs. While with dynamic programming we were filling out the table in a backward manner, i.e. for each cell $(i_1, \ldots, i_k)$ we considering its incoming edges, Dijkstra's algorithm fills out the table in a forward manner, updating the costs of the vertices incident to edges that are outgoing from $(i_1, \ldots, i_k)$ in a stepwise fashion. What if we could determine that a cell (vertex) $(i_1, \ldots, i_k)$ will *not* be part of the optimal alignment path? In that case, we would not add the neighbors of $(i_1, \ldots, i_k)$ to the queue, essentially pruning the search space.

The Carrillo-Lipman algorithm implements such a pruning step. For ease of exposition, we consider the case with $k = 3$ input strings $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$ that each have the same length n. Let

- $D(i, j, k)$ be the minimum SP-cost of aligning prefixes $\mathbf{v}_1[1..i], \mathbf{v}_2[1..j], \mathbf{v}_3[1..k]$,
- $d_{p,q}(i, j)$ be the cost of the *induced* pairwise alignment of $\mathbf{v}_p[1..i], \mathbf{v}_q[1..j]$ (where $1 \leq p < q \leq 3$) of the optimal multiple alignment of $\mathbf{v}_1[1..i], \mathbf{v}_2[1..j], \mathbf{v}_3[1..k]$,
- $D_{p,q}(i, j)$ be the minimum cost of aligning $\mathbf{v}_p[1..i], \mathbf{v}_q[1..j]$ (where $1 \leq p < q \leq 3$).

Clearly, $d_{p,q}(i,j) \geq D_{p,q}(i,j)$ as the induced pairwise alignment of an optimal multiple alignment are not necessarily optimal themselves. Moreover, by definition of the SP-score, we have $D(i,j,k) = d_{1,2}(i,j) + d_{1,3}(i,k) + d_{2,3}(j,k)$. Thus, we have

$$D(i,j,k) = d_{1,2}(i,j) + d_{1,3}(i,k) + d_{2,3}(j,k) \geq D_{1,2}(i,j) + D_{1,3}(i,k) + D_{2,3}(j,k). \tag{2}$$

Let's consider suffixes $\mathbf{v}_1[i..n], \mathbf{v}_2[j..n], \mathbf{v}_3[k..n]$. We define

- $D^+(i,j,k)$ be the minimum SP-cost of aligning suffixes $\mathbf{v}_1[i..n], \mathbf{v}_2[j..n], \mathbf{v}_3[k..n]$,
- $d^+_{p,q}(i,j)$ be the cost of the *induced* pairwise alignment of $\mathbf{v}_p[i..n], \mathbf{v}_q[j..n]$ (where $1 \leq p < q \leq 3$) of the optimal multiple alignment of $\mathbf{v}_1[i..n], \mathbf{v}_2[j..n], \mathbf{v}_3[k..n]$,
- $D^+_{p,q}(i,j)$ be the minimum cost of aligning $\mathbf{v}_p[i..n], \mathbf{v}_q[j..n]$ (where $1 \leq p < q \leq 3$).

Again, we have $d^+_{p,q}(i,j) \geq D^+_{p,q}(i,j)$, $D^+(i,j,k) = d^+_{1,2}(i,j) + d^+_{1,3}(i,k) + d^+_{2,3}(j,k)$ and thus

$$D^+(i,j,k) = d^+_{1,2}(i,j) + d^+_{1,3}(i,k) + d^+_{2,3}(j,k) \geq D^+_{1,2}(i,j) + D^+_{1,3}(i,k) + D^+_{2,3}(j,k). \tag{3}$$

The cost of the optimal alignment passing through (i,j,k) is $D(i,j,k) + D^+(i,j,k)$ (this should remind you of the Hirschberg algorithm!). Combining these two previous results, we get

$$D(i,j,k) + D^+(i,j,k) \geq D(i,j,k) + D^+_{1,2}(i,j) + D^+_{1,3}(i,k) + D^+_{2,3}(j,k). \tag{4}$$

Now, suppose we have alignment of $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$ with cost z. Note that we do not know anything about the quality of this alignment. If

$$D(i,j,k) + D^+_{1,2}(i,j) + D^+_{1,3}(i,k) + D^+_{2,3}(j,k) > z$$

then we know that

$$D(i,j,k) + D^+(i,j,k) > z.$$

In other words, if we force the alignment to go through (i,j,k) we will get a score that is worse than z. Hence, the optimal alignment will *not* pass through (i,j,k). The cool thing about this procedure is that $D^+_{1,2}(i,j) + D^+_{1,3}(i,k) + D^+_{2,3}(j,k)$ can be computed in $O(n^2)$ time. How do we go about finding an alignment with cost z? We have to resort to heuristics.