CS 466 – Introduction to Bioinformatics
Lectures 10-12

Mohammed El-Kebir

October 26, 2018

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

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 Carillo-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 whether a cell (vertex) $(i_1, \ldots, i_k)$ is guaranteed not to 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 Carillo-Lipman algorithm implements such a pruning step. For ease of exposition, we consider the case with three 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

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

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

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

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

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

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.

3 Tree and Star Alignments

In general, heuristics come with no hard, theoretical guarantees on their worst-case performance. For instance, the greedy progressive alignment algorithm that we will see in class has no such guarantees. In other words, we do not know how far off the cost of the returned solution is from the optimal cost. In this section, we will describe a constant-factor approximation algorithm that comes with theoretical guarantees on its performance. That is, the cost of a returned solution is at most a constant factor more than the optimal cost.

Let $\mathbf{v}_1, \ldots, \mathbf{v}_k \in \Sigma^*$ be our input strings. Recall that $D(\mathbf{v}_i, \mathbf{v}_j)$ is the optimal (weighted) edit distance between $\mathbf{v}_i$, and $\mathbf{v}_j$. We start with the following key definition.

Definition 1. *Let T be a tree with k nodes, where each node is labeled with a distinct string from $\{\mathbf{v}_1, \ldots, \mathbf{v}_k\}$. Then, a multiple alignment A of $\mathbf{v}_1, \ldots, \mathbf{v}_k$ is called* consistent *with T if the induced pairwise alignment of $\mathbf{v}_i$, and $\mathbf{v}_j$ has cost $D(\mathbf{v}_i, \mathbf{v}_j)$ for each edge $(\mathbf{v}_i, \mathbf{v}_j)$ of T.*

The following theorem states that it is easy to compute an alignment that is consistent with a given tree T.

Theorem 1 (Gusfield [1])**.** *Let T be a tree whose k nodes are each labeled by a distinct string from $\{\mathbf{v}_1, \ldots, \mathbf{v}_k\}$. We can compute an alignment $A(T)$ of $\mathbf{v}_1, \ldots, \mathbf{v}_k$ that is consistent with T in $O(k^2n^2)$ time.*

Proof. We will prove this theorem by constructing $A(T)$ one string at a time. We will show that the theorem holds, by proving inductively that adding each string $\mathbf{v}_i$ while maintaining consistency takes $O(in^2)$ time. The base case $i = 2$ is trivial, amounting to a pairwise alignment of $\mathbf{v}_1$ and $\mathbf{v}_2$ that by definition is consistent with a tree T that connects the two vertices by a single edge. Computing the optimal pairwise alignment takes $O(in^2) = O(2n^2) = O(n^2)$ time.

As for the step $i > 2$, by the induction hypothesis we are given a tree T' that is consistent with strings $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_{i-1}$. Let $\mathbf{v}_i$ be a string adjacent in T' to one of $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_{i-1}$. Let $\mathbf{v}_j$ be the vertex that is adjacent to $\mathbf{v}_i$ in T'. Let $\bar{\mathbf{v}}_j$ denote the gapped sequence corresponding to $\mathbf{v}_j$ in the multiple alignment $A(T')$. We align $\bar{\mathbf{v}}_j$ and $\mathbf{v}_i$ with the added rule that $\delta(-, -) = 0$. That is, two opposing gaps have a cost of 0.

Let $\bar{\mathbf{v}}'_j$ and $\bar{\mathbf{v}}_i$ be the two resulting gapped sequences. If the optimal alignment does not insert any new gaps into $\bar{\mathbf{v}}_j$ then we add $\bar{\mathbf{v}}_i$ to $A(T')$. The result is a multiple alignment with one more string, where the induced cost of $\bar{\mathbf{v}}'_j$ and $\bar{\mathbf{v}}_i$ equals $D(\mathbf{v}_j, \mathbf{v}_i)$ and where the induced costs from the previous alignment remain unchanged. However, if the optimal alignment inserted a new gap into $\bar{\mathbf{v}}_j$ between characters l and $l+1$, we insert a gap between columns l and $l+1$ in each sequence of the original multiple alignment $A(T')$. Observe that the induced costs of the original alignment remain unchanged, whereas the induced alignment of $\bar{\mathbf{v}}'_j$ and $\bar{\mathbf{v}}_i$ has cost $D(\mathbf{v}_j, \mathbf{v}_i)$. Thus, the new alignment is consistent with the tree T extended by the edge $(\mathbf{v}_j, \mathbf{v}_i)$. As for the running time, observe that the given alignment $A(T')$ composed of $(i-1)$ sequences has a length of at most $(i-1)n$ (recall that length of pairwise alignment of two sequences of length m and n is at most $m + n$). Thus, worst case, $\bar{\mathbf{v}}_j$ has length $(i-1)n = O(in)$ while $\mathbf{v}_i$ has length n. Thus, it takes $O(in^2)$ time to compute $\bar{\mathbf{v}}'_j$ and $\bar{\mathbf{v}}_i$.

The total running time is $\sum_{i=1}^{k-1} O(in^2) = O(k^2n^2)$. Hence, we can compute an alignment $A(T)$ of $\mathbf{v}_1, \ldots, \mathbf{v}_k$ that is consistent with T in $O(k^2n^2)$ time. □

3.1 Star Alignment

We say that a cost function $\delta : (\Sigma \cup \{-\}) \times (\Sigma \cup \{-\}) \to \mathbb{R}$ satisfies the *triangle inequality* if

$$\delta(x, z) \leq \delta(x, y) + \delta(y, z). \tag{5}$$

Recall that $D(\mathbf{v}_i, \mathbf{v}_j)$ is the optimal (weighted) edit distance between $\mathbf{v}_i$ and $\mathbf{v}_j$. We have the following definition.

Definition 2. *Given strings* $\mathbf{v}_1, \ldots, \mathbf{v}_k \in \Sigma^*$*, the* center string $\mathbf{v}_c$ *(where* $c \in [k]$*) is the input string that minimizes* $\sum_{i=1}^{k} D(\mathbf{v}_c, \mathbf{v}_i)$*. The* center star *is a star tree of* k *nodes with the center node labeled by* $\mathbf{v}_c$ *and each of the remaining* $k-1$ *nodes labeled by a distinct string from* $\{\mathbf{v}_1, \ldots, \mathbf{v}_k\} \setminus \{\mathbf{v}_c\}$*.*

We use the previous theorem to obtain an alignment A_c of $\mathbf{v}_1, \ldots, \mathbf{v}_k$ consistent with the center star. Let $d(\mathbf{v}_i, \mathbf{v}_j)$ denote the pairwise alignment cost of $\mathbf{v}_i$ and $\mathbf{v}_j$ induced by A_c. Clearly, $d(\mathbf{v}_i, \mathbf{v}_j) \geq D(\mathbf{v}_i, \mathbf{v}_j)$. We introduce the shorthand $d(A_c) = \sum_{i<j} d(\mathbf{v}_i, \mathbf{v}_j)$.

Lemma 1. *Let* δ *be a cost function that satisfies the triangle inequality. Then, for any input string* $\mathbf{v}_i$ *and* $\mathbf{v}_j$*, it holds that* $d(\mathbf{v}_i, \mathbf{v}_j) \leq d(\mathbf{v}_i, \mathbf{v}_c) + d(\mathbf{v}_c, \mathbf{v}_j) = D(\mathbf{v}_i, \mathbf{v}_c) + D(\mathbf{v}_c, \mathbf{v}_j)$*.*

Proof. Consider any column of A_c. Let x, y and z be the characters in this columns of strings i, c and j, respectively. By the triangle inequality, we have that $\delta(x, z) \leq \delta(x, y) + \delta(y, z)$. Thus, $d(\mathbf{v}_i, \mathbf{v}_j) \leq d(\mathbf{v}_i, \mathbf{v}_c) + d(\mathbf{v}_c, \mathbf{v}_j)$. By Theorem 1 it follows that $d(\mathbf{v}_i, \mathbf{v}_c) + d(\mathbf{v}_c, \mathbf{v}_j) = D(\mathbf{v}_i, \mathbf{v}_c) + D(\mathbf{v}_c, \mathbf{v}_j)$. □

Let A^* be the optimal alignment of strings $\mathbf{v}_1, \ldots, \mathbf{v}_k$ with cost $d(A^*)$.

Theorem 2. $d(A_c)/d(A^*) \leq 2(k-1)/k < 2$*.*

Proof. We start by defining

$$f(A_c) = \sum_{\substack{(i,j) \in [k]^2 : \\ i \neq j}} d(\mathbf{v}_i, \mathbf{v}_j) \text{ and } f(A^*) = \sum_{\substack{(i,j) \in [k]^2 : \\ i \neq j}} d^*(\mathbf{v}_i, \mathbf{v}_j). \tag{6}$$

Clearly, $2d(A^*) = f(A^*)$ and $2d(A_c) = f(A_c)$. Recalling that $D(\mathbf{v}_i, \mathbf{v}_j)$ is the optimal (weighted) edit distance between $\mathbf{v}_i$ and $\mathbf{v}_j$, we have by the previous lemma that

$$f(A_c) = \sum_{\substack{(i,j) \in [k]^2 : \\ i \neq j}} d(\mathbf{v}_i, \mathbf{v}_j) \tag{7}$$

$$\leq \sum_{\substack{(i,j) \in [k]^2 : \\ i \neq j}} [D(\mathbf{v}_i, \mathbf{v}_c) + D(\mathbf{v}_c, \mathbf{v}_j)]. \tag{8}$$

Observe that for any fixed j, the terms $D(\mathbf{v}_c, \mathbf{v}_j)$ and $D(\mathbf{v}_j, \mathbf{v}_c)$ show up $2(k-1)$ times. Furthermore, observe that $D(\mathbf{v}_c, \mathbf{v}_j) = D(\mathbf{v}_j, \mathbf{v}_c)$. Thus, we have

$$f(A_c) \leq \sum_{\substack{(i,j) \in [k]^2 : \\ i \neq j}}^{k} [D(\mathbf{v}_i, \mathbf{v}_c) + D(\mathbf{v}_c, \mathbf{v}_j)] = (2k-1) \sum_{j=1}^{k} D(\mathbf{v}_c, \mathbf{v}_j). \tag{9}$$

From the other side, we have

$$f(A^*) = \sum_{\substack{(i,j)\in[k]^2 : \\ i\neq j}} d^*(\mathbf{v}_i, \mathbf{v}_j) \tag{10}$$

$$\geq \sum_{\substack{(i,j)\in[k]^2 : \\ i\neq j}} D(\mathbf{v}_i, \mathbf{v}_j) \tag{11}$$

$$= \sum_{i=1}^{k} \sum_{j=1}^{k} D(\mathbf{v}_i, \mathbf{v}_j) \tag{12}$$

Now, the crucial observation is that the sum $\sum_{i=1}^{k} \sum_{j=1}^{k} D(\mathbf{v}_i, \mathbf{v}_j)$ of the minimum costs of all ordered pairs of strings equals summing the cost of k different stars, each centered around one of the k input strings. We picked $\mathbf{v}_c$ such that it was the star with smallest cost. Thus, we have

$$f(A^*) \geq \sum_{i=1}^{k} \sum_{j=1}^{k} D(\mathbf{v}_i, \mathbf{v}_j) \tag{13}$$

$$\geq k \sum_{j=1}^{k} D(\mathbf{v}_c, \mathbf{v}_j) \tag{14}$$

Hence, we have

$$\frac{d(A_c)}{d(A^*)} = \frac{f(A_c)}{f(A^*)} \leq \frac{(2k-1)\sum_{j=1}^{k} D(\mathbf{v}_c, \mathbf{v}_j)}{k \sum_{j=1}^{k} D(\mathbf{v}_c, \mathbf{v}_j)} = \frac{2(k-1)}{k} < 2. \tag{15}$$

□

References

[1] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, New York, NY, USA, 1997.