Notes on a multiple alignment cost-bound of Carrillo and Lipman*

John Kececioglu†

Spring 1988

Abstract A concrete implementation of a branch-and-bound algorithm for computing a multiple sequence alignment minimizing the sum of the costs of all induced pairwise alignments is discussed. The algorithm incorporates a bound of Carrillo and Lipman on the cost of an optimal alignment and uses a definition of Altschul for counting gaps in multiple sequence alignments.

1 The bound

In [CL88], Carrillo and Lipman consider the problem of computing a multiple sequence alignment whose sum of costs of all induced pairwise alignments is minimal. For a multiple sequence alignment $A_{1,\cdots,K}$ of sequences $S_1, S_2, \ldots, S_K$, let $A_{i,j}$ be the induced pairwise alignment of S_i and S_j. Then in their formulation, a multiple sequence alignment A has cost $c(A) = \sum_{i<j} c(A_{i,j})$. (We write A for $A_{1,\cdots,K}$ when context resolves the ambiguity.)

Carrillo and Lipman realized that not all alignments A need be considered. Define $d(S_1, S_2, \ldots, S_K) \equiv \min_A c(A)$. Let $L \equiv \sum_{i<j} d(S_i, S_j)$ and U be lower and upper bounds on $d(S_1, \ldots, S_K)$, and $\mathcal{A}$ be a multiple sequence alignment of minimal cost. Then

$$\begin{aligned} U - L &\geq \sum_{i<j} [c(\mathcal{A}_{i,j}) - d(S_i, S_j)] \\ &\geq c(\mathcal{A}_{p,q}) - d(S_p, S_q) \end{aligned}$$

for all distinct p and q, since for all alignments A, $c(A_{i,j}) \geq d(S_i, S_j)$. Rearranging, we derive the Carrillo and Lipman bound

$$c(\mathcal{A}_{i,j}) \leq d(S_i, S_j) + U - L \tag{1}$$

*These unpublished notes were originally written for the paper, D.J. Lipman, S.F. Altschul and J.D. Kececioglu, "A tool for multiple sequence alignment," *Proceedings of the National Academy of Sciences USA* 86, 4412–4415, 1989, but were not incorporated into it. They were later incoporated into the paper, S.K. Gupta, J.D. Kececioglu and A.A. Schäffer, "Making the shortest-paths approach to sum-of-pairs multiple sequence alignment more space efficient in practice," Proceedings of the 6th *Symposium on Combinatorial Pattern Matching*, Springer-Verlag Lecture Notes in Computer Science 937, 128–143, 1995, at the urging of Alejandro Schäffer. A more complete bibliography has been added to the original notes.

†Department of Computer Science, The University of Arizona, Tucson, Arizona 85721. Electronic mail: `johnk@cs.arizona.edu`

Note that $\Omega(K^2)$ terms are thrown away; while the bound may achieve equality, for K equally dissimilar sequences, $c(\mathcal{A}_{i,j}) - d(S_i, S_j)$ will be overestimated by a factor $O(K^2)$.

In the classic dynamic programming algorithm, a directed acyclic graph is constructed in which source to sink paths of weight C correspond to alignments of cost C. For multiple sequence alignment, this graph is conveniently represented with vertices embedded at integer coordinates in K-space and sequences labelling the coordinate axes. Inequality (1) restricts consideration to those paths whose projection onto the plane for every sequence pair A and B has weight at most $d(A, B) + U - L$.

2 An algorithm

We design an algorithm incorporating Inequality (1) where pairwise alignments are scored as the sum of single-symbol insertion, deletion, and substitution costs; this algorithm is later extended for an affine gap cost function. Input consists of the sequences $S_1, S_2, \ldots, S_K$ of lengths $N_1, N_2, \ldots, N_K$, and the difference $\delta \equiv U - L$. (We will write N for $\max_i N_i$.)

The algorithm is naturally organized in two stages. In the first stage, $O(K^2)$ pairwise sequence comparisons are performed to determine, for each plane defined by sequence pair S_i and S_j, the set of points $\langle p, q\rangle$ on a path from $\langle 0, 0\rangle$ to $\langle N_i, N_j\rangle$ of weight at most $d(S_i, S_j) + \delta$. Let $F_{i,j}$ be the set of such points $\langle p, q\rangle$ on the face formed by S_i and S_j. At the conclusion of this stage, the lower bound $L = \sum_{i<j} d(S_i, S_j)$ and upper bound $U = L + \delta$ can be computed.

In the second stage, an alignment corresponding to a shortest source to sink path whose projection lies wholly within the $F_{i,j}$ is computed. This weighted directed acyclic graph $G = \langle V, E\rangle$ has vertex set

$$V = [0, N_1] \times [0, N_2] \times \cdots \times [0, N_K],$$

and edge set

$$E = \left\{ v \to w \parallel w - v \in \{0, 1\}^K - \{0\}^K \right\},$$

where we use the standard componentwise subtraction of vectors. For character a, define $a^1 \equiv a$ and $a^0 \equiv \epsilon$. Using this notation, edge $\langle v_1, \cdots, v_K\rangle \xrightarrow{e} \langle w_1, \cdots, w_K\rangle$ has weight $w(e) = \sum_{i<j} d\left((S_i[w_i])^{w_i - v_i}, (S_j[w_j])^{w_j - v_j}\right)$, where $A[k]$ denotes the k-th character of string A. Let $R_{i,j}$ be the region in K-space of points $\langle p_1, p_2, \ldots, p_K\rangle$ such that $\langle p_i, p_j\rangle \in F_{i,j}$, and define $R \equiv \bigcap_{i<j} R_{i,j}$. Then the paths of interest lie within $G \setminus R$, the subgraph of G induced on the vertex subset R.

This algorithm is formalized in Figure 1. Two fundamental data structures are used. Function $\textsc{heap}()$ creates an empty heap; $\textsc{insert}(x, k, h)$ inserts item x with key k into heap h; $\textsc{extract}(h)$ returns and removes the item of minimum key in heap h; $\textsc{decrease}(x, k, h)$ decreases to k the key of x in h. Function $\textsc{trie}()$ creates an empty trie; $\textsc{install}(x, s, t)$ installs item x with string s into trie t; $\textsc{lookup}(s, t)$ returns the item with string s in trie t, or Λ if none exists. In addition, two new data types are used. Type **point** is a K-tuple of integers; a point variable p has named fields $p.1, p.2, \cdots, p.K$, and is created by $p \leftarrow \textsc{point}(p_1, p_2, \cdots, p_K)$. Type **vertex** has two named fields: a distance D, and a point P; statement $v \leftarrow \textsc{vertex}(d, p)$ creates a vertex v with $v.D = d$ and $v.P = p$. The heap of vertices is ranked by distance, while the trie of vertices is constructed over the integer sequences forming points. Shortest paths are computed using the distance field; adjacent vertices are generated and accessed using the point field.

algorithm $\textsc{msa}$(**sequence** $S_1, S_2, \cdots, S_K$; **number** δ)

  trie T ; **heap** H ; **vertex** s, t, v, w ; **point** p, q

  [Stage 1.]

  $L \leftarrow 0$

  for $I \leftarrow 1$ **to** $K-1$ **do**

    for $J \leftarrow I+1$ **to** K **do**

      Let $a_1 a_2 \cdots a_n = S_I$ and $b_1 b_2 \cdots b_m = S_J$.

      Compute $C_f[i,j] \equiv d(a_1 \cdots a_i, b_1 \cdots b_j)$ and $C_r[i,j] \equiv d(a_{i+1} \cdots a_n, b_{j+1} \cdots b_m)$.

      for $i \leftarrow 1$ **to** n **do**

        $F_{I,J}[i] \leftarrow \{j \parallel C_f[i,j] + C_r[i,j] \leq C_f[n,m] + \delta\}$

      od

      $L \leftarrow L + C_f[n,m]$

    od

  od

  $U \leftarrow L + \delta$

  [Stage 2.]

  $s \leftarrow \textsc{vertex}(0, \textsc{point}(0, 0, \cdots, 0))$; $t \leftarrow \textsc{vertex}(\infty, \textsc{point}(N_1, N_2, \cdots, N_K))$

  $T \leftarrow \textsc{trie}()$; $\textsc{install}(s, s.P, T)$; $\textsc{install}(t, t.P, T)$

  $H \leftarrow \textsc{heap}()$; $\textsc{insert}(s, s.D, H)$

  while $\neg$ $\textsc{empty}(H)$ **do**

    $v \leftarrow \textsc{extract}(H)$; $p \leftarrow v.P$

    if $v = t$ **then**

      output $\textsc{traceback}(t)$; **return**

    fi

    if $v.D + \sum_{i<j} C_r[p.i, p.j] \leq U$ **then**

      for all $q \leftarrow \textsc{point}(q_1, q_2, \cdots, q_K)$ **s.t.** $q - p \in \{0,1\}^K - \{0\}^K$ **and** $q_j \in F_{i,j}[q_i]$ **do**

        $w \leftarrow \textsc{lookup}(q, T)$

        if $w = \Lambda$ **then**

          $w \leftarrow \textsc{vertex}(\infty, q)$; $\textsc{insert}(w, w.D, H)$; $\textsc{install}(w, w.P, T)$

        fi

        if $v.D + w(p \xrightarrow{e} q) < w.D$ **then**

          $w.D \leftarrow v.D + w(e)$; $\textsc{decrease}(w, w.D, H)$

        fi

      od

    fi

  od

  output There does not exist a multiple sequence alignment with cost at most U.

end

Figure 1 The Algorithm.

Function TRACEBACK(v) returns an alignment by tracing backward through the graph from vertex v. (By examining the distance and point fields of preceding vertices, a shortest path can be reconstructed.)

One optimization incorporated in the algorithm of Figure 1 is that edges leaving some vertices may be ignored. When a vertex v is removed from the heap, $v.D$ is the length of a shortest path from s to v. (It is assumed that all insertion, deletion, and substitution costs—and hence all edge weights—are positive.) A lower bound on the length of a path from v to t is $\sum_{i<j} C_r[v.P.i, v.P.j]$. If this quantity plus $v.D$ exceeds U, then the length of any path from s to t through v exceeds U, and edges leaving v do not need to be examined.

This basic algorithm may be extended for an affine gap penalty function. We count gaps in multiple sequence alignments using the definition of Altschul [A88]. Altschul evaluates the length of a path through our graph as the sum of a weight function defined over *pairs* of edges. Thus the weight of an edge leaving a vertex depends on how that vertex was reached, and it is no longer correct to extend shortest paths as in Figure 1. Conceptually, we construct a new graph in which the length of a path is once more the sum of the weights of its edges, and paths of length C are in one-to-one correspondence with alignments of cost C under Altschul's measure. Formally, given the old graph $G = \langle V, E\rangle$, we form the new graph $G' = \langle V', E'\rangle$ where $V' = E$ and $E' = \{\langle u, v\rangle \to \langle v, w\rangle \parallel u \to v \to w \text{ is a path in } G\}$.

We modify the algorithm as follows. A new type **edge** is used, with three named fields: V, the tail vertex of the edge in the old graph G; W, the head of the edge in G; and D, the length of a shortest path from the source to W ending with this edge. Now the type **vertex** contains the two fields E, a list of entering edges from the old graph G, and P, its point coordinate, as before. The trie still maintains vertices for access by coordinate, but the heap now contains edges ranked by D distances. When an edge e is extracted from the heap, vertices x adjacent to $e.W$ are generated within the region R as before, but now the list $x.E$ is searched for the edge $f = \langle e.W, x\rangle = \langle f.V, f.W\rangle$. If found, the distance $f.D$ may be decreased to $e.D + w(\langle e.V, e.W\rangle \to \langle f.V, f.W\rangle)$. A vertex's edge list E can be realized as a search tree with edge tails as keys. While an incident edge list may be $O(2^K)$ in size worst-case, the graph G' is constructed "on the fly" as before, and only edges examined in G are inserted into the edge lists.

As a final algorithmic issue, consider the working memory needed to compute the $F_{I,J}[i]$ sets for two sequences S_I and S_J of length m and n. These sets are defined in terms of the matrices $C_f[i,j]$ and $C_r[i,j]$, which consume $O(mn)$ space, and may be the dominant storage requirement for inputs consisting of a few long sequences. By applying the divide-and-conquer technique of Hirschberg [H75], one can do better at a slight increase in time. The cost matrix C_f can be computed for successively increasing rows in $O(n)$ space, but the corresponding row of C_r is simultaneously needed to determine the column elements forming the set $F_{I,J}[i]$, and this technique applies to C_r for successively *decreasing* rows. By computing row $m/2$ of C_f in a forward pass, row $m/2$ of C_r in a reverse pass, summing to determine $F_{I,J}[i]$, and recursing on the rows in the first and second halves, one can reduce the working space without too much recomputation. Let $T(m,n)$ be the time required by this recursive process, w be the maximum width of the $F_{I,J}$ region on any row, and j be the minimum column element in the set $F_{I,J}[m/2]$. Then

$$T(m,n) \quad \leq \quad mn + T(\frac{m}{2}, w+j) + T(\frac{m}{2}, n-j)$$

$$
\begin{aligned}
&\leq mn \sum_{k=0}^{\log m} 2^{-k} + mw \sum_{k=0}^{\log m} \frac{2^k - 1}{2^k} \\
&= O(mn + wm \log m)
\end{aligned}
$$

The working storage is now $O(n + w \log m)$, since $O(n)$ space is needed to compute the cost matrices a row at a time, and the $\log m$ recursive calls require width w slices of the cost matrices as arguments. While this is $O(mn \log m)$ time and $O(n \log m)$ space worst-case, for $w = O(n/\log m)$, we have achieved linear space without exceeding quadratic time.

3 An implementation

The modified algorithm for an affine gap cost function has been coded in C; we now describe the concrete implementation of the data structures.

Making the realistic assumption that edge weights are integers, we can implement a discrete heap with buckets for the possible path lengths, the number of buckets being bounded by U. Then INSERT() and DECREASE() are constant time operations, and all EXTRACT() operations take $O(U + |R|)$ total time, which is in turn $O(KN + |R|)$. As a minor concern, the statement VERTEX$(\infty, \ldots)$ can be implemented as VERTEX$(U + 1, \ldots)$, since U is a bound on vertex distances.

Since the trie is constructed over integer sequences, the sons of a node are conveniently indexed through an array of pointers. Consider accessing the vertex in the trie associated with point $\langle p_1, p_2, \cdots, p_K \rangle$ in R. After having traversed the trie on prefix $p_1 p_2 \cdots p_{j-1}$, the next coordinate p_j must be in the set $\bigcap_{i<j} F_{i,j}[p_i]$. Let m and n be the minimum and maximum integers in this set. Then pointers to child nodes, and an encoding of the intersection, can be represented in a vector of length $O(n - m)$. These vectors tend to be short, and once allocated, never need change in length as new vertices are installed. Generation of adjacent vertices is often short-circuited by "falling off" the trie. To minimize space consumption, the P vertex field is not represented as a K-tuple of integers; $v.P$ points to the trie leaf associated with vertex v, and coordinates are recovered from trie node labels by following father links to the root.

The approach described computes an optimal alignment using Dijkstra's shortest path algorithm, but there are alternatives. Any algorithm which visits the vertices in a topological order can correctly compute vertex distances; perhaps the most natural order is lexicographic over vertex coordinates. Vertices can be enumerated lexicographically within region R very efficiently, but there are two disadvantages. First, every vertex in the region will be generated, while the shortest path algorithm visits only vertices whose distance is at most that of the sink from the source. (Even though this is achieved at the price of heap operations, which can in general be costly, the discrete bucketed heap has very low overhead.) Second, the optimization of ignoring out-edges of vertices whose lower bounded sink return cost exceeds the upper bound fits very nicely into a shortest path approach; if every path into a vertex within the region is pruned, the vertex is simply never visited. (This actually occurs quite frequently, since the Carillo and Lipman bound is rather loose, as explained earlier.) Attempting to compute return costs as the vertices are generated lexicographically via nested loops running over face intersections will result in a looser cut-off bound, since the exact distance to a vertex cannot be known until it is generated. Remedying this by enumerating vertices topologically by sweeping a breadth-first "front" through the graph will in effect be mimicking a shortest paths approach without the benefit of visiting only vertices whose distance is within that of the sink.

The final implementation issue concerns the treatment of holes in the region R. Recall $F_{I,J}[i] \equiv \{j \| C_f[i,j] + C_r[i,j] \leq d(S_I, S_J) + \delta\}$. These sets are implemented as a vector of j values, which can require $O(N)$ space for each set. Alternately, one could approximate the set in $O(1)$ space by storing the minimum and maximum integer elements, on the assumption that the sets are dense, and ignore holes. This will significantly reduce the overhead in a lexicographic enumeration, but in our experiments the number of vertices skipped by capturing holes in the K-dimensional region outweighs the increased enumeration overhead. (And the fewer vertices visited in a shortest path enumeration as discussed earlier outweigh the low overhead of a lexicographic enumeration with holes.)

References

[A88] Altschul, S. "Gap costs for multiple sequence alignment." *Journal of Theoretical Biology* 138, 297–309, 1989.

[A89] Altschul, Stephen F. "Leaf pairs and tree dissections." *SIAM Journal on Discrete Mathematics* 2:3, 1989.

[ACL89] Altschul, Stephen F., Raymond J. Carroll and David J. Lipman. "Weights for data related by a tree." *Journal of Molecular Biology* 208, 1989.

[AL89] Altschul, Stephen F. and David J. Lipman. "Trees, stars, and multiple biological sequence alignment." *SIAM Journal on Applied Mathematics* 49:1, 197–209, 1989.

[CL88] Carrillo, Humberto and David Lipman. "The multiple sequence alignment problem in biology." *SIAM Journal on Applied Mathematics* 48, 1073–1082, 1988.

[LAK89] Lipman, David J., Stephen F. Altschul and John D. Kececioglu. "A tool for multiple sequence alignment." *Proceedings of the National Academy of Sciences USA* 86, 4412–4415, 1989.

[Spou89] Spouge, J.L. "Speeding up dynamic programming algorithms for finding optimal lattice paths." *SIAM Journal on Applied Mathematics* 49, 1552–1556, 1989.

[H75] Hirschberg, D.S. "A linear space algorithm for computing maximal common subsequences." *Communications of the ACM* 18:6, 341–343, 1975.