

CS 262

Lecture 14: Chaining of Local Alignments, Protein Profile HMMs, and Classification

February 22, 2007

Overview

First we will get a brief overview of two popular alignment tools for protein sequencing. In 1994, a tool named CLUSTALW came out, and it was a huge improvement over previous tools. It's one of the better-known tools of bioinformatics. We won't talk about it, though, because since then, other programs have continued improving on it. MUSCLE improved its speed and runs much faster than CLUSTALW (and is also more accurate). ProbCons improved the accuracy of the returned sequences and is one of the most accurate tools today. These are interesting because they let us combine different ideas we have previously described to solve one task.

The next major topic we will cover is aligning genomic DNA from different species. For example, we might want to align the human and mouse genomes, or maybe even many different species at once. Each is about three billion letters long, and they are very scrambled with respect to each other, so the problem of aligning them is fundamentally different from what we have seen before (relatively short things). We will build a system out of several components to help us align them. To help us in this task, we will cover a family of algorithms that implements sparse dynamic programming, which will be the heart of this lecture.

Progressive Alignment (recap of previous lecture)

When we are aligning multiple sequences and we know the structure of the tree, we can use a heuristic that works relatively well in practice. We can start at the bottom and align neighboring leaves with some algorithm we already know (for example, Needleman-Wunsch). Each time we align two sequences, we create a profile to represent the alignment. A profile represents the frequencies of every letter and gap in the alignment. We can then align these just as we would normal sequences except with a modified substitution score. For a review of these, see the previous lecture.

If we don't know the structure of the tree, we can first compute pairwise distances between each pair of sequences. With that, we can cluster them into trees with UPGMA or Neighbor Joining. Once we have a tree, we can align the sequences over that tree with progressive alignment.

Protein Aligners

MUSCLE

Input: A set of sequences with no tree structure defined. N is the number of sequences and L is the length of each sequence.

First it calculates a tree over which it will perform progressive alignment. This could simply be done by performing all $O(N^2)$ pairwise alignments, but since MUSCLE is designed to be very fast, it wants to avoid that. So it instead builds a distance function $D_{DRAFT}(x,y)$ that is defined in terms of the number of common k -mers (where k equals three). So for each sequence, we build a list of all its k -mers in $O(L \log L)$ time (by sorting the k -mers). Then for each pair, we walk down their lists and get the number of k -mers in both lists in linear time. This gives a crude estimate of their distance in

$N^2 L \log L$ time. This is much more efficient than doing pairwise Needleman-Wunsch, which would be $O(N^4)$.

With these distances it builds a draft tree T_{DRAFT} with UPGMA. This takes quadratic time.

It then does progressive alignment over T_{DRAFT} to create multiple alignment M_{DRAFT} . This takes time $O((N-1)L^2)$ and gives us a first shot at a multiple alignment of the sequences.

The next step is to improve this alignment. Based on M_{DRAFT} it builds an elaborate Kimura-based distance function $D(x,y)$ that is based on some evolutionary measure. This should be much more accurate than D_{DRAFT} .

It next builds a more elaborate tree T based on D , again with UPGMA.

It performs progressive alignment over T to build M , which is the almost-final alignment.

Lastly, it does many rounds of iterative refinement to improve this alignment. The general idea is to remove one sequence, realign it to all the others, and then put it back. This would lead to a better alignment. MUSCLE does this by cutting the tree at a random edge and then realigning the profiles of those two subsets. Each of these steps takes the time of one alignment, and we have already done a linear number of alignments, so we can do as many of these as we want without changing the running time. MUSCLE lets you choose how many rounds you want to do.

1. Fast measurement of all pairwise distances between sequences
 - $D_{DRAFT}(x, y)$ defined in terms of # common k-mers ($k \sim 3$) – $O(N^2 L \log L)$ time
2. Build tree T_{DRAFT} based on those distances, with UPGMA
3. Progressive alignment over T_{DRAFT} , resulting in multiple alignment M_{DRAFT}
 - Only perform alignment steps for the parts of the tree that have changed
4. Measure new Kimura-based distances $D(x, y)$ based on M_{DRAFT}
5. Build tree T based on D
6. Progressive alignment over T , to build M
7. Iterative refinement; for many rounds, do:
 - *Tree Partitioning*: Split M on one branch and realign the two resulting profiles
 - If new alignment M' has better sum-of-pairs score than previous one, accept

Interesting side note

When we align two profiles, we can align them optimally in quadratic time. But profiles are just compact representations of the underlying alignments. The original problem, aligning two multiple alignments, does pairwise alignment of two alignments using affine gaps. This is NP-hard. The compact representation with profiles can be done optimally in quadratic time because it loses a lot of information. The intuition for this is that at any point when we're aligning two columns, we are closing a number of gaps. To make the optimal decision here, we have to remember where each of the gaps opened. Unless we remember where the gaps opened, we don't know whether or not we should penalize for a gap.

ProbCons

First it computes all posterior matrices M_{xy} , where $M_{xy} = P(x_i \sim y_j)$ using pairwise HMMs. That is, it computes all the probabilities that one thing is aligned to another, which is the sum over all possible alignments of x and y of the probability of that alignment times the indicator function of whether x_i and y_j align in that alignment. This can be done in a very similar way to the Posterior algorithm (with forward and backward) we learned for normal HMMs.

It then re-estimates the posterior matrices M'_{xy} with probabilistic consistency. First, recall the consistency heuristic we discussed last lecture. To score the match of x_i to y_j , we could ask for a third sequence z_k to help align the two. This is done here probabilistically. It is a heuristic step, but it works well. So here we ask for a third sequence z_k what the probability is that x_i goes to z_k which goes to y_j . So we multiply these two probabilities over all locations in all sequences z . We average this over all possible sequences z (including x and y themselves). See the formula in the picture below. This is matrix multiplication and takes time $O(NL^3)$ (the cubic step comes from going over all i,j,k). Note that these matrices are sparse (almost all entries are zero) because a given position is very unlikely to align to most positions outside a small range. So if we discard all small probabilities (less than 1%), we are left with only a very small number (five or ten) of probabilities. This lets us do this step in time linear in the sequence length times the average number of entries per row, which is around ten. So this step is actually extremely fast. Doing it for all sequences takes time cubic in the number of sequences (for all x,y,z) but linear in the sequence length.

Then for every pair x,y it computes the maximum expected alignment accuracy. That is, it defines A_{xy} to be the alignment that maximizes $E(x, y) = \sum_{i,j \text{ aligned} \in A} M'_{xy}(i, j)$. This sum is the expected number of correctly-aligned pairs in the two sequences, or the expected accuracy.

It then builds a tree T with hierarchical clustering (starting with the things with the largest such sum) using as its similarity measure $E(x,y)$. This step is quick.

From there it does progressive alignment to maximize E .

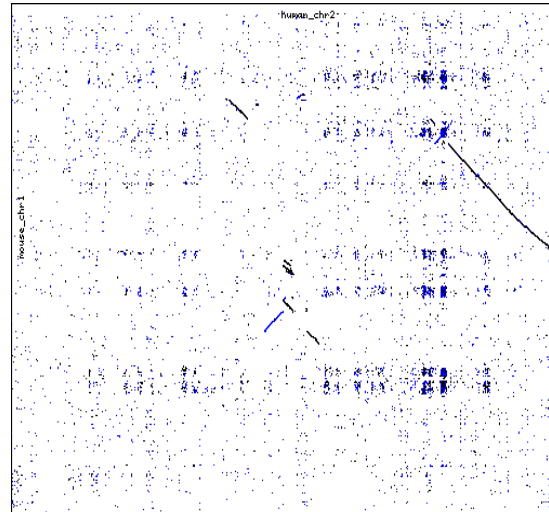
Lastly, it does the standard iterative refinement procedure.

1. Computation of all posterior matrices M_{xy} : $M_{xy}(i, j) = \text{Prob}(x_i \sim y_j)$, using a HMM
2. Re-estimation of posterior matrices M'_{xy} with **probabilistic consistency**
 - $M'_{xy}(i, j) = 1/N \sum_{\text{sequence } z} \sum_k M_{xz}(i, k) \times M_{yz}(j, k); \quad M'_{xy} = \text{Avg}_z(M_{xz}M_{zy})$
3. Compute for every pair x, y , the maximum expected accuracy alignment
 - A_{xy} : alignment that maximizes $\sum_{\text{aligned } (i, j) \text{ in } A} M'_{xy}(i, j)$
 - Define $E(x, y) = \sum_{\text{aligned } (i, j) \text{ in } A_{xy}} M'_{xy}(i, j)$
4. Build tree T with hierarchical clustering using similarity measure $E(x, y)$
5. Progressive alignment on T to maximize $E(\dots)$
6. Iterative refinement; for many rounds, do:
 - **Randomized Partitioning**: Split sequences in M in two subsets by flipping a coin for each sequence and realign the two resulting profiles

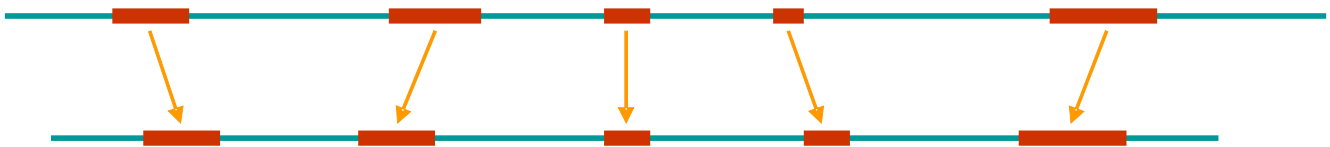
Both of these tools are available online, as is their source code, which is public domain.

Rapid Global alignments on long sequences

This is an alignment of a human and a mouse chromosome. Each dot is a region of local similarity. Long lines are long regions that are similar, but they actually contain many gaps in them. The goal is to find the long lines but to ignore the surrounding noise. Our motivation is that genomic sequences are very long and scrambled, so we cannot simply do our normal algorithms, as they will be too slow.

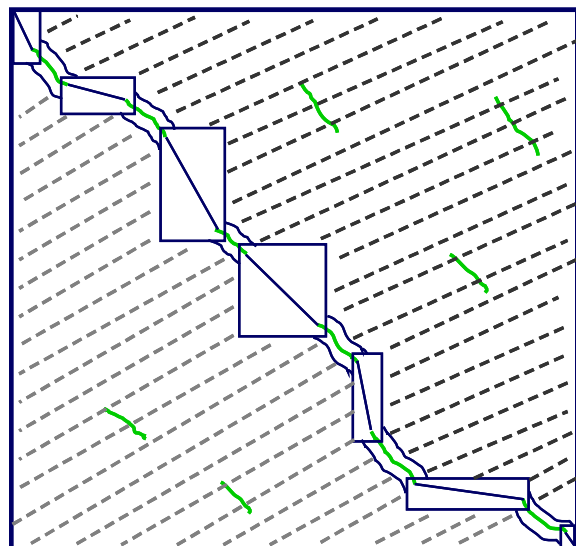


The main idea of the algorithm is that most areas have nothing to do with each other. But interesting areas, ones with genes, often have strongly conserved islands of similarity. For instance, genes that have to do with DNA replication will be conserved almost exactly across different species, so those areas will be very similar. So when aligning two large genomic sequences, there will be highly diverse regions interrupted by shorter conserved regions.



Our strategy takes advantage of this pattern. We first quickly identify the islands of strong similarity. We do this using fast local alignment tools such as BLAST. We then chain an optimal ordering of those together to form a longer alignment in $N \log N$ time that gives us a rough idea of the global alignment. We can then refine this alignment, since aligning the small islands cuts the original large problem into multiple shorter problems.

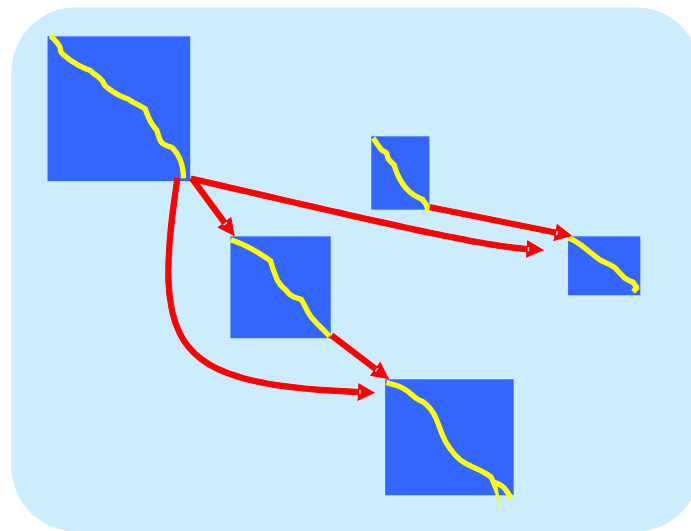
This is an example of saving cells in dynamic programming. We first find local alignments with a tool like BLAST and then chain them to pick the best ordered list of the locations, which we will see how to do later in this lecture. We then run a restricted dynamic programming algorithm over the local aligners, which can be very restricted, and over the areas in between, which is unrestrained. This gets rid of a large part of the matrix and lets us do dynamic programming on an area hopefully linear in the length of the sequences.



Chaining Local Alignments

Task: We have a set of local alignments. There might be a lot of them (perhaps tens of thousands), for instance from repeats. We want to find a chain of them that has the highest weight. Two items can be in a chain if the bottom-right corner of one is higher (above and to the left) of the top-left corner of the other.

We could implement this with a quadratic-time algorithm. It would build a DAG (directed acyclic graph) where the nodes are the local alignments and with directed edges that connect alignments that can be chained. Then for every local alignment we look at everything it can be chained from and choose the one that maximizes the score. That is, for each node, the optimal path ending in it is the max over all nodes that have an edge to it of the weight of that node (the score of the chain up to that point) plus the weight of the edge connecting them (perhaps based on the score of the current alignment and the distance between the two). This works, but it is quadratic, since in the worst case a local alignment can have a linear number of incoming arrows.



Sparse Dynamic Programming: Longest Increasing Subsequence (LIS)

Motivation and idea: LCS

Let us go back to a moment to the LCS problem (find the longest common subsequence of two sequences). We can do a quadratic-time solution to this with DP with a match score of 1 and no mismatch/gap penalties. The LCS is then all the matched elements. In the DP matrix, a common subsequence is just a chain of matches. So solving LCS is really just solving a chaining problem.

Now imagine a case where there are very few matches. Can we solve this in time proportional to the number of common matches rather than in time proportional to the product of the two sequence lengths? We can. The idea is that each match can be matched only to one of the previous matches. But it's not obvious how to do this without considering all (linear number of) other matches.

Sparse DP and LIS.

Problem: Given an ordered sequence, find the longest increasing subsequence.

Quadratic algorithm: Create a list L . At the i^{th} position in L , store the last letter of the longest increasing subsequence that contains exactly i letters. So we store the longest increasing subsequence seen so far. Then at each place in the sequence, we check the entire list to see if we can increase the length of some longest increasing subsequence by one element.

$N \log N$ algorithm (explanation below):

Let input be $w: w_1, \dots, w_n$

INITIALIZATION:

```
L: last LIS elt. array  L[0]  = -inf
                        L[1]  =  $w_1$ 
                        L[2...n] = +inf
```

```
B: array holding LIS elts; B[0] = 0
```

```
P: array of backpointers
```

```
// L[j]: smallest  $j^{\text{th}}$  element  $w_i$  of  $j$ -long LIS seen so far
```

ALGORITHM

```
for i = 2 to n {
    Find j such that  $L[j - 1] < w[i] \leq L[j]$ 
    L[j]  $\leftarrow w[i]$ 
    B[j]  $\leftarrow i$ 
    P[i]  $\leftarrow B[j - 1]$ 
}
```

We have three arrays. L is the most conceptually interesting. $L[j]$ contains the smallest element we have seen so far that ends an increasing subsequence of j elements. So when we are done, the last element in L will be the last element in the longest increasing subsequence. B and P store pointers that let us reconstruct the entire sequence (since $L[1], L[2], \dots$ is not the longest increasing subsequence).

We run the algorithm by iterating over a sequence w . For each element in it, we find where to insert it in L . This can be done via binary search since L is sorted. The key idea is that $w[j] > L[j-1]$ means that $w[i]$ can be part of a j -long increasing subsequence ($L[j-1]$ is defined to be the smallest element that ends an increasing subsequence of $j-1$ elements, and $w[j]$ is larger than it, so it is the last element of a j -long increasing subsequence). And since $w[i] \leq L[j]$, it is “better” than it. That is, before, $L[j]$ was the smallest element that ends an increasing subsequence of j elements. But $w[i]$ is smaller than it and larger than the element that precedes it in the sequence, and so it is the new smallest element that ends an increasing subsequence of j elements. Anything that could follow the old $L[j]$ in an increasing subsequence could also follow $w[i]$. So we write $w[i]$ into $L[j]$ and store some traceback pointers.

To trace back when we are iterating, find the largest L , which will be the last element in the subsequence. Then look at $B[\text{index of that element}]$, which is the index of that element in the original string. Then look at $P[\text{that index}]$ for the previous element's index in the string, and then $P[\text{that index}]$ for the previous element. We continue until we get to $P[1]=0$.

The running time is $N \log N$ since the search takes time $\log N$ and there are a linear number of those.

We then covered an example on the string 256417. Follow the algorithm and the image below to see how it works. We walk down the string and add/update values as appropriate.

	W	1	2	3	4	5	6
		2	5	6	4	1	7
L	0	2	5	6	7		
B	0	5	4	3	6		
P			1	2	1	0	3

Traceback:

	W	1	2	3	4	5	6
		2	5	6	4	1	7
L	0	2	5	6	7		
B	0	5	4	3	6		
P			1	2	1	0	3

Next lecture

Next lecture we will apply this algorithm to chaining local alignments and not just increasing subsequences.