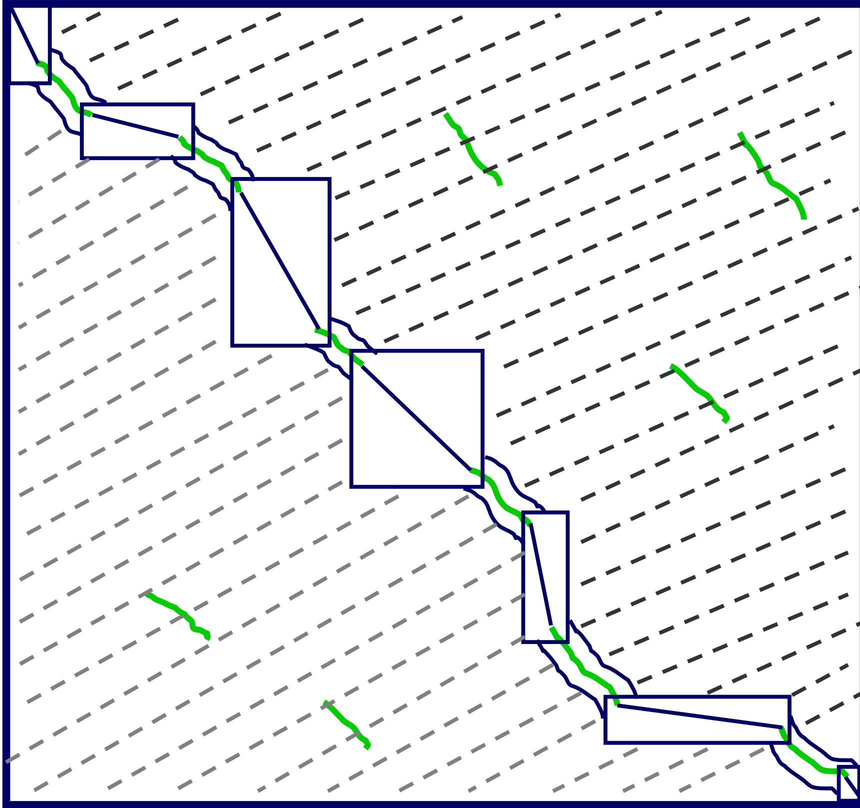
CS262 Computational Genomics
Prof. Serafim Batzoglou
Lecture 15: **Chaining of Local alignments, Protein Profile HMMs and Classification**
Feb. 28, 2006
Scribe: Ilya Shlyakhter

1. Chaining of Local alignments.

When doing global alignment of two complete genomes (e.g. mouse and human) to find long conserved regions, we can't use Needleman-Wunsch because 1) the quadratic running time is too slow for such long sequences, and 2) the results could be arbitrary because of all the duplication and shuffling in the genomes since the species diverged (e.g. of the many human and mouse duplicates of some original segment, which ones get paired in a global alignment?) For these reasons, global alignment in this situation is done by 1) using BLAST to find local alignments that are better than some threshold, and 2) choosing a chaining of the local alignments that maximizes some score (e.g. the number and size of the local alignments in the chain). This method is likely to find, for example, long multi-exon genes in human and mouse that evolved from a common gene. Evolutionary pressures conserve the contents of exons and regulatory elements, so BLAST will find their local alignments; evolutionary pressures also conserve the order of the exons (for proper splicing) and of regulatory elements (e.g. promoter before gene), so the corresponding local-alignment rectangles in the Needleman-Wunsch matrix will be chainable (running from upper-left corner of the matrix towards the lower-right corner).

This motivates the problem of chaining local alignments. This is illustrated by the following picture:



Here, green lines show local alignments found by blast. Blue rectangles show connections between successive local alignments in a chain. When constructing a chain, we have the freedom to 1) choose local alignments through which to thread the chain, and 2) route the global alignment between successive local alignments subject to remaining within the blue rectangles; 3) fine-tune the regions of the global alignment corresponding to the local alignments, subject to remaining within a small distance of the local alignments. Note that once a chaining has been chosen, the area of the dynamic programming matrix that must be explored in choosing a global alignment is a small fraction of the dynamic programming matrix, and is essentially linear rather than quadratic in genome length. In the following, we focus on the problem of choosing the local alignments through which to thread the chain.

First, we consider the simpler related problem of finding a longest strictly increasing subsequence in a list of numbers. We give an O(n log n) algorithm for this problem. It goes through the input characters w_i for $i=1,\ldots,n$, keeping an array L[j] where L[j] is the smallest possible tail of a j-long increasing subsequence of $w_1,\ldots,w_i$. I.e. if there are any j-long increasing subsequences of $w_1,\ldots w_i$, then L[j] holds the smallest tail element across all these subsequences; otherwise, L[j] is undefined. Note that 1) if L[j] is defined then L[j'] is defined for $j'<j$, and if L[j] is undefined then L[j'] is undefined for $j'>j$ because if a (j+1)-long increasing subsequence of $w_1,\ldots,w_i$ exists it contains a j-long increasing subsequence of $w_1,\ldots,w_i$; 2) we have $L[j] < L[j+1]$ because a (j+1)-long increasing subsequence of $w_1,\ldots,w_i$ with a tail L[j+1] smaller than L[j] would

contain a j-long increasing subsequence of w_1,…,w_i with a tail smaller than L[j], and L[j] is the smallest tail of any j-long increasing subsequence of w_1,…,w_i. L[j]=L[j+1] can't happen because then the j'th element of the (j+1)-long increasing subsequence of w_1,…,w_i ending in L[j+1] would have to be at least L[j] (or a j-long increasing subsequence of w_1,…,w_i with a tail smaller than L[j] would exist), and simultaneously be smaller than L[j+1] (or the (j+1)-long subsequence ending in L[j+1] wouldn't be strictly increasing).

Initially, we set L[1] to w_1: w_1 is the smallest tail of the only possible 1-long increasing subsequence of w_1,…,w_1. Now suppose L[j] is correctly computed for w_1,…,w_i and we move to w_{i+1}. For any j such that L[j] < w_{i+1}, there exists a (j+1)-long increasing subsequence of w_1,…,w_{i+1) ending in w_{i+1}: we take the j-long increasing subsequence of w_1,…,w_i of which L[j] is the tail and append w_{i+1}. Let's find the largest j for which L[j] < w_{i+1}; then we have L[j] < w_{i+1} <= L[j+1] (if L[j+1] is defined of course). As noted above, there exists a (j+1)-long increasing subsequence of w_1,…,w_{i+1} ending in w_{i+1}. This subsequence ow w_1,…,w_{i+1} has a tail element that is not larger than the tail element of any other (j+1)-long subsequence of w_1,…,w_{i+1}. Every other (j+1)-long subsequence either ends in w_{i+1} or doesn't. If it does, its tail is w_{i+1} itself. If it doesn't, then it is a (j+1)-long subsequence of w_1,…,w_i and L[j+1] is the smallest tail of all such subsequences and w_{i+1} <= L[j+1]. Thus, we're entitled to set L[j+1] to w_{i+1} and this will preserve the invariant that L[j+1] is the smallest tail of all (j+1)-long increasing subsequences of w_1,…,w_{i+1}. Note that the sorting of L[] in increasing order is also preserved by this assignment. Also, note that for constructing (j+2)-long sequences from (j+1)-long sequences, the new smaller-or-equal value w_{i+1} of L[j+1] works at least as well as or better than the previous value of L[j+1]. For any i'>i+1, if L[j+1] < w_{i'} then also w_{i+1} < w_{i'} and so any increasing subsequence we could have obtained previously by appending to L[j+1] can still be obtained by appending to w_{i+1}, but the converse does not hold. Thus, by replacing L[j+1] with the smaller-or-equal value w_{i+1} we have not hurt (and possibly have improved) our ability to construct (j+2)-long increasing subsequences from existing (j+1)-long ones.

Runtime analysis: because L[] is sorted in increasing order, finding the largest j such that L[j] < w_{i+1} can be done by binary search in O(log n) time. Since we go through n characters w_1,…,w_n, the total runtime of this procedure is O(n log n). Termination: At the end, we take the largest j for which L[j] is defined; then L[j] is the tail of a j-long increasing subsequence of w_1,…,w_n, and there is no j'-long increasing subsequence of w_1,…,w_n for j'>j. Now, how do we get the actual longest increasing subsequence rather than just its length and tail?

To be able to recover the actual longest increasing subsequence, we keep backtrack pointers. Let us keep, for each j, not just the value L[j] of the smallest tail of any j-long subsequence of w_1,…,w_i, but also the index B[j] of the character with which that smallest-tail j-long subsequence of w_1,…,w_i ends. (Index of the character means its position in the input list w_1,…,w_i). Also, let's keep for each input position i' for

which B[j]=i' and L[j]=w_{i'} for some j, the index P[i'] of the (j-1)'st character in the j-long subsequence whose tail is L[j].

Now assume that L[], B[] and P[] are set correctly for step i of the algorithm, and consider the move to step (i+1). Recall that when we set L[j+1] to w_{i+1}, it was because we could construct a (j+1)-long increasing subsequence of w_1,…,w_{i+1} by taking the j-long increasing subsequence of w_1,…,w_i with tail L[j] and appending w_{i+1}. We record this fact by setting B[j+1] to i+1 (since w_{i+1} is the new tail L[j+1] of our new (j+1)-long subsequence of w_1,…,w_{i+1})and P[i+1] to B[j] (since B[j+1]==i+1, L[j+1]==w_{i+1} and the j'th character L[j] of our new (j+1)-long subsequence of w_1,…,w_{j+1} is the tail of the j-long subsequence of w_1,…,w_i from which we derived our new subsequence and has index B[j] in the input sequence). At the end, we take the largest j for which L[j] is defined, use B[j] to get the index of the tail L[j] of the j-long increasing subsequence of w_1,…,w_n, use P[B[j]] to get the next-to-last character of this subsequence, then continue to use P[] to trace back to the beginning of the sequence. Initially, we set B[0] to 0 so that we can stop the backtracing when we hit a zero entry in P[].

Here is an example of running this algorithm (from 2005 scribed notes):

Given an ordered alphabet $a = a_1,a_2,\ldots,a_m$ such that $a_i < a_{i+1}$, we have a sequence $w = w_1,w_2,\ldots,w_m$. An increasing subsequence of w is one where we take a subset of elements from w, keep them in order with respect to their positions in w and the property $w_i < w_{i+1}$ holds. The longest increasing subsequence is the longest such sequence for w.

Original:

2 4 9 8 5 3 6 3 9

Increasing Subsequence:

2 4 8 9

Note: that the above is not the *longest* increasing subsequence.

Now, one might ask, "How do we find the longest increasing subsequence efficiently?" The answer is that we use one of the easiest yet most confusing algorithms. The code for input $w=w_1,w_2,\ldots,w_n$ follows.

Initialization:
L: 1-index array
L[1] = w_1
B: array of back pointers of length n
P: array that reconstructs the chain

```
for i = 2 to n
{
        Find j such that L[ j ] < w[ i ] <= L[ j + 1 ]
        L[ j + 1 ] = w[ i ]
```

```
        B[ j + 1 ] = i
        P[i]  = ( j > 0 ? B[ j ] : 0)
}
```

But what on earth is actually going on. The entry L[j] contains the last element, w_i, which is the smallest element in w that ends an increasing subsequence that is j-long. This forces L to be sorted. An example helps to demonstrate this effect.

w = 3 1 4 1 5 9 2

L: 3
B: 1 _ _ _ _ _ _
P: 0 _ _ _ _ _ _

L: 1
B: 2 _ _ _ _ _ _
P: 0 _ _ _ _ _ _

L: 1 4
B: 2 3 _ _ _ _ _
P: 0 0 2 _ _ _ _

L: 1 4 (the 1 is a different 1 than in the previous step)
B: 4 3 _ _ _ _ _
P: 0 0 2 0 _ _ _

L: 1 4 5
B: 4 3 5 _ _ _ _
P: 0 0 2 0 3 _ _

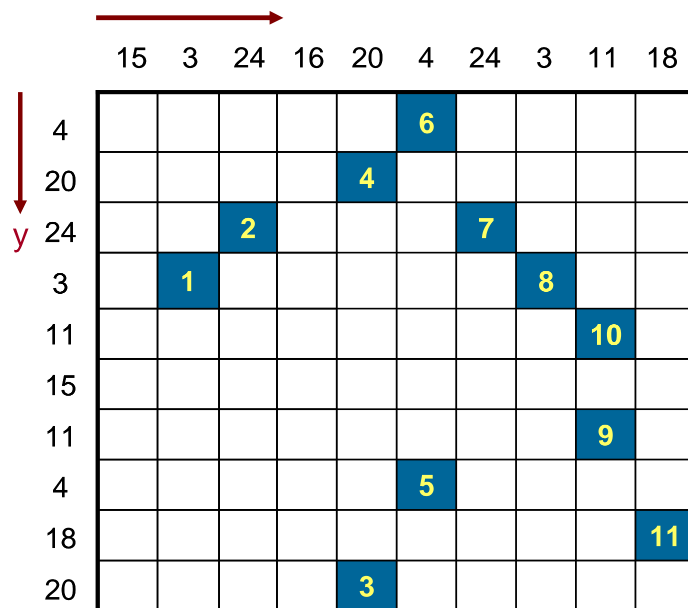
L: 1 4 5 9
B: 4 3 5 6 _ _ _
P: 0 0 2 0 3 5 _

L: 1 2 5 9
B: 4 3 7 6 _ _ _
P: 0 0 2 0 3 5 4

To reconstruct the sequence we do the following:
|L| is the size of L
Start at P[B[|L|]] we track back
P[6] = 5
P[5] = 3
P[3] = 2
P[2] = 0
So our LIS is w[0], w[2], w[3], w[5], w[6] = null, 1, 4, 5, 9

Converting Longest Ccommon Subsequence to Longest Increasing Subsequence

We want to be able to express LCS in terms of LIS.



We insert every matching point (i,j) into w by first sorting by columns then by reverse row order as shown above.

The sequence for this ordering is

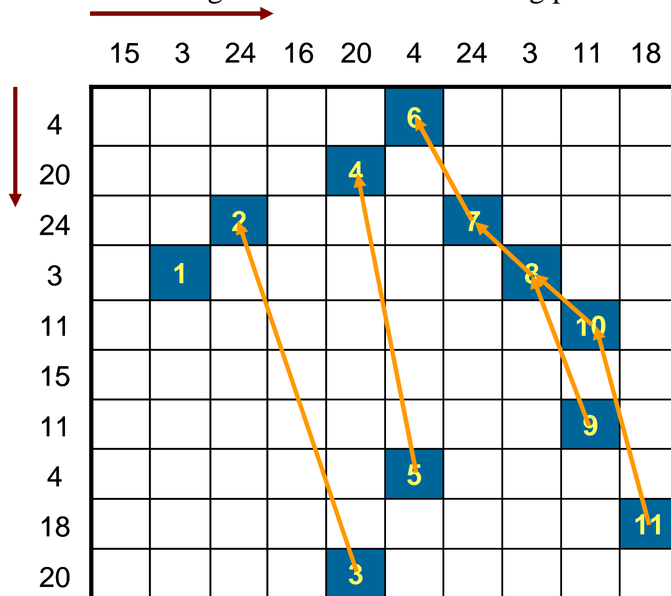
w = (4,2) (3,3) (10,5) (2,5) (8,6) (1,6) (3,7) (4,8) (7,9) (5,9) (9,10).

We can chain a to b if and only if the following is true:
a = (y,x), b = (y',x')
a is before b in w

- $(x < x')$ or $(x = x'$ and $y > y')$

This particular ordering models the fact that lower numerical values of y have fewer restrictions on what can chain to them.

Below is an image of the chained matching points:



This is the L array from the LIS algorithm:

L =

1. (4,2)
2. (3,3)
3. (3,3), (10,5)
4. (2,5), (10,5)
5. (2,5), (8,6)
6. (1,6), (8,6)
7. (1,6), (3,7)
8. (1,6), (3,7), (4,8)
9. (1,6), (3,7), (4,8), (7,9)
10. (1,6), (3,7), (4,8), (5,9)
11. (1,6), (3,7), (4,8), (5,9), (9,10)

LIS = (1,6), (3,7), (4,8), (5,9), (9,10)

Therefore we get
LCS = 4, 24, 3, 11, 18

Rectangle Chaining

Now we are ready to tackle our original problem, rectangle chaining. First, we establish some notation for the following algorithm:

- (h_j, l_j) – y-coordinates of rectangle j, h is the top y-coordinate and l is the bottom one
- w(j) – weight of rectangle j
- V(j) – optimal score of chain ending in j

- L – list of triples (l_j, V(j), j)

Note: L is sorted by l_j, top to bottom, and is implemented as a balanced binary tree.
Here is the algorithm doing the chaining:

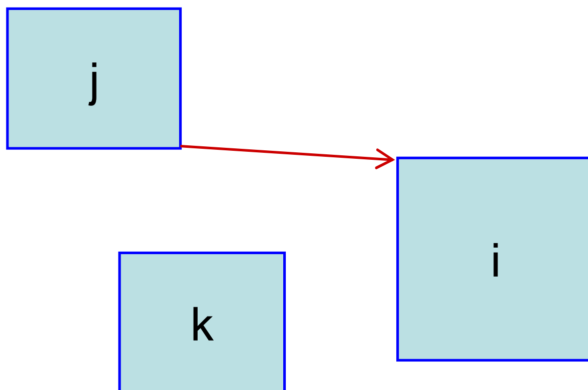
Go through rectangle x-coordinates, from lowest to highest (leftmost to rightmost in the picture):

1. When on the leftmost end of i:

    a. j: rectangle in L, with largest $lj < hi$
    b. $V(i) = w(i) + V(j)$

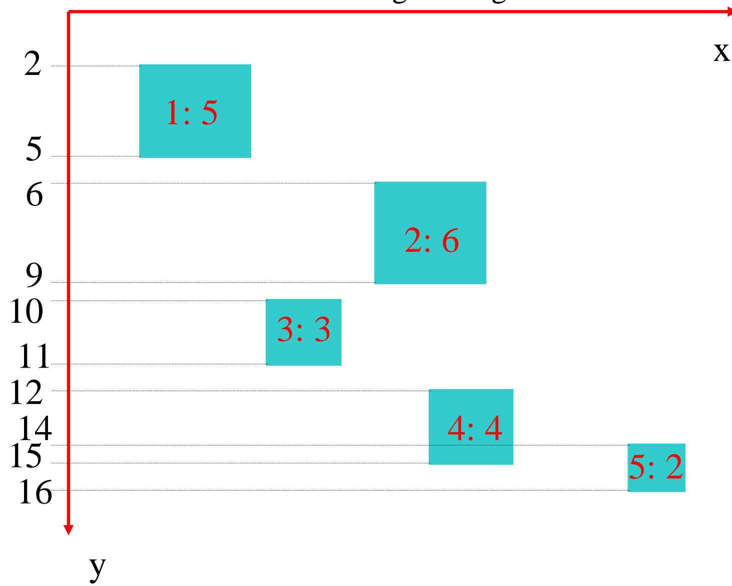
1. When on the rightmost end of i:

    a. k: rectangle in L, with smallest $lk \geq li$
    b. If $V(i) \geq V(k)$:
        i. **INSERT** (li, V(i), i) in L
        ii. **REMOVE** all (lj, V(j), j) with $V(j) \leq V(i)$ & $lj \geq li$



- j has largest coordinate that is chainable to i
- k is the most immediate more restricting rectangle

We will now chain the following rectangles:



c(x) => compute score of x
I(x) => insert x
R(x) => remove x

Box#: score -> back pointer

c(1) 1: 5 -> 0
I(1) L: (5, 5, 1)
c(3) 3: 8 -> 1
I(3) L: (5, 5, 1)
(11, 8, 3)
c(2) 2: 11 -> 1
c(4) 4: 12 -> 3
r(3) L: (5, 5, 1)
I(2) L: (5, 5, 1)
(9, 11, 2)
I(4) L: (5, 5, 1)
(9, 11, 2)
(15, 12, 4)
c(5) 5: 13 -> 2
I(5) L: (5, 5, 1)
(9, 11, 2)
(15, 12, 4)
(16, 13, 5)

And our chain is 1, 2, 5 with a score of 13.

Protein Classification

Number of known protein sequences and of solved 3D protein structures has been growing exponentially. However, the number of major distinct general shapes ("folds") that known proteins have has been converging to about 1000 (i.e. most newly discovered proteins fall into one of the already-known folds). Why is that? Perhaps because all proteins are related by evolution (new proteins can only come from long-established ones) so truly new shapes have little opportunity to arise. Or perhaps because there really only exist a few possible shapes that manage to meet the stringent requirements for a working protein, e.g. that it quickly old into native shape after being synthesized. Whatever the reason, this is good for biologists: when we find a new protein we can classify it into one of the 1000 known folds and already know a lot about it from the general properties of that fold.

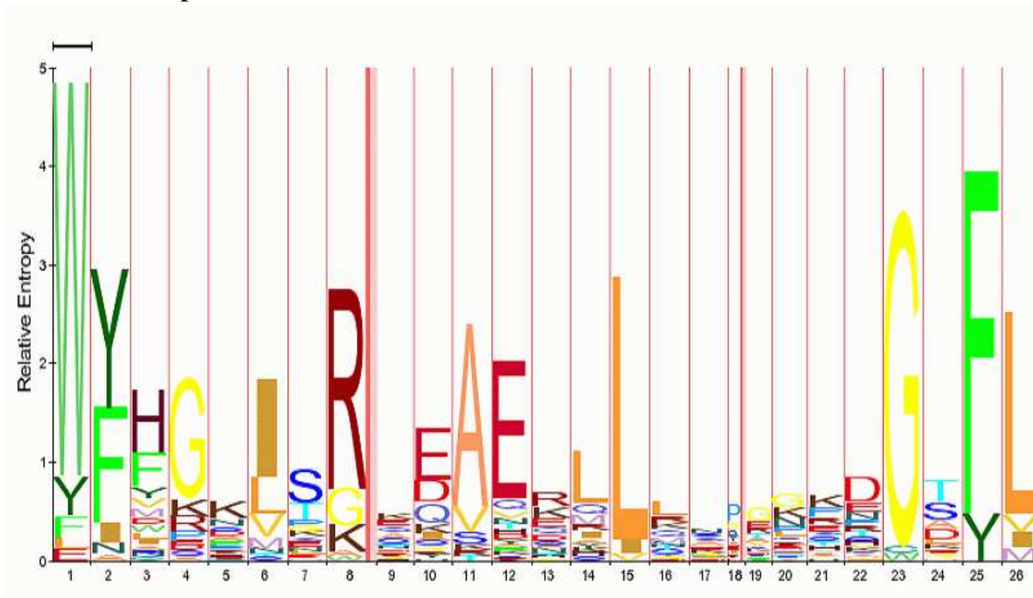
Protein folds are divided into protein superfamilies which are divided into protein families. Proteins in a superfamily have faint evolutionary similarities detectable at the sequence level. Proteins in a family have sequence similarity of 25% or higher.

Several protein classification systems exist. SCOP (scop.berkeley.edu) is entirely manual, has the highest classification quality, but covers fewest proteins. It is done mainly by one man (A. Murzin) with unique abilities. CATH www.biochem.ucl.ac.uk/bsm/cath) is semi-manual, covers more proteins and is less precise than SCOP, and FSSP (www.ebi.ac.uk/dali/fssp/fssp.html) is completely automatic, covers more proteins than CATH and is less precise than CATH. Ideally we want an automatic and precise way to classify a new protein into an existing family, superfamily (if it belongs to no known families) or fold (if it belongs to no known superfamilies). Current approaches include: BLAST/PsiBLAST, Profile HMMs and Supervised machine learning methods.

BLAST is simplest: you do one sequence-similarity search to find already-classified proteins with sequence similarity to the new protein. You assign to the new protein the classification from which you got the most BLAST hits to your protein.

PsiBLAST is a more accurate method of doing the same thing: you first build a sequence profile of proteins that are similar to the new proteins, then BLAST the database of already-classified proteins with this profile instead of with the single sequence of the new protein. To build the profile, you 1) find pairwise alignments of your new protein to proteins in the database, which have scores above a given threshold; 2) build a sequence profile summarizing the alignments you found 3) repeat, now using the profile of proteins similar to the new protein sequence instead of the new protein sequence itself (i.e. find pairwise alignments of the profile to proteins in the database, etc). This approach is much more precise than BLAST, especially when your new protein is an outlier within its family. Precision of automatic protein classification methods can be measured by leave-one-out experiments (take an already-classified protein and see how the automatic method would have classified it based on all other known proteins – would we rediscover the correct classification?)

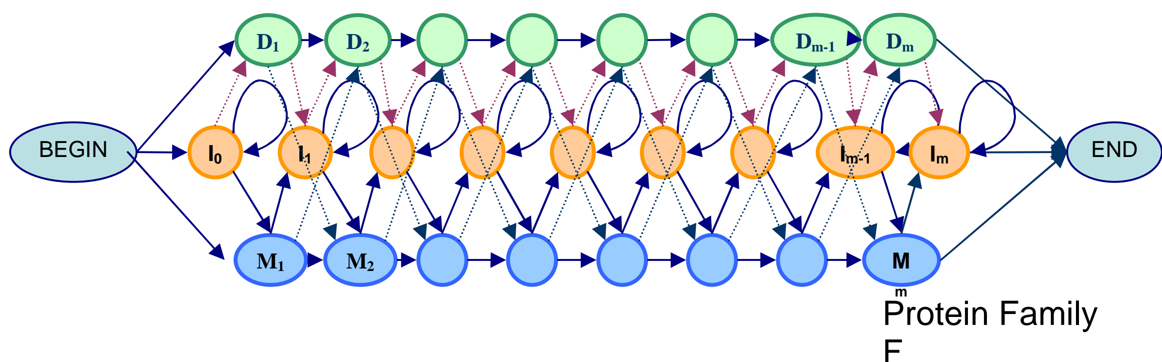
A PsiBLAST profile looks like this:



It gives the relative frequencies at each position of the various amino acids, as found in the protein sequences summarized by the profile.

A limitation of using profiles to characterize protein families, is that a profile can't capture that "most proteins in this family have a gap at position 10". This can be a common variation across a family (e.g. if this is a point on the outside of the protein where loops of various lengths are easily added without disturbing protein function). Conversely, profiles also can't capture that "at position 10 there must NOT be a gap because this position is critical to protein function"; profiles can capture frequencies of amino acids at each position, but not frequencies of gaps. We need a model of protein families that lets us model the locations of gaps you would see if you did a multiple-alignment of proteins in the family. Profile HMMs let us do just that.

A profile HMM looks like this:



The emission alphabet of this HMM is the set of amino acids. This HMM is a generative model; the probability that it generates a given protein depends on how "typical" the

protein is for the protein family represented by the HMM – or equivalently, how likely a given protein is to be part of this family.

So how is this HMM constructed? States in the bottom row correspond to profile positions with letters. If a profile position almost always contains a given amino acid, the corresponding M state almost always emits that amino acid; in general, the emission probability of an amino acid depends on the relative frequency of that amino acid in that position of the profile. The middle row states are insertion states. Their self-transition probabilitites model lengths of gaps at that profile location, and their emission probabilities can in principle model gap contents. The top row states are delection states; transitions to these states model the likelihood of deletions at a given profile location. The profile HMM is a concise summary of a multiple alignment of sequences in a protein family, and can be learned from a multiple alignment by converting the letter and gap frequencies at each position into emission and transition probabilities as suggested above.

Given a protein sequence and a profile HMM, you can "align" the sequence to the HMM by finding the most likely path through the HMM that would generate the sequence. This can be computed using a DP algorithm. For details, see Durbin Ch. 5.

Classification using profile HMMs can be done by computing a multiple alignment of sequences in each known protein family, summarizing this multiple alignment with a profile HMM, then for an unknown protein sequence computing the probability that this sequence is produced by each family's protein HMM and assigning the protein the family whose profile HMM has the highest likelihood of producing the protein.

Profile HMMs are generative models, and classification using generative models suffers from the problem that if there are not enough data points for some protein family, new proteins that are outliers in that protein family may not be correctly classified. An alternative approach is to use Support Vector Machines (SVMs) for classification. We fix n k-mers, define a distance function between two proteins as the number of k-mers shared by the two proteins, and treat proteins as points in n-dimensional space. We find a hyperplane separating points from two protein families in this space, and train an SVM to choose a small number of representative points (support vectors) from each protein family, and classify novel proteins based on their distance to these representative points. Experimental results show that this approach is much more accurate than using profile HMMs for classification, while having the same computational complexity.