

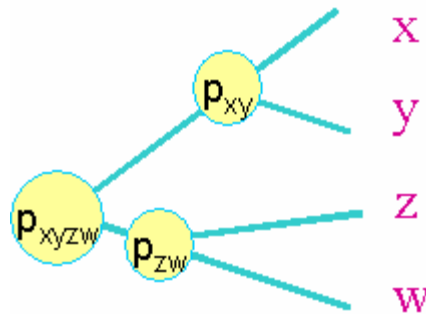
# CS 262 Computational Genomics

## Professor Serafim Batzoglou

### Lecture 14: Genomic Alignment

#### I. Multiple Sequence Alignments

##### Progressive Sequence Alignments (A Review)



Let us review the main idea behind **progressive sequence alignments**: When the phylogenetic tree is known, we begin by aligning the neighboring leaves of the tree with the Needleman-Wunsch global alignment algorithm. We still apply this procedure as we move up the tree, with the only difference being that we are now aligning **profiles** rather than sequences.

It will make  $N - 1$  alignments to produce a multiple alignment of  $N$  sequences. Recall that, in profiles, we only store the frequencies of each letter at each position of the sequence, so aligning two profiles of length  $N$  takes the same big-O time as aligning two sequences of length  $N$ :  $O(N^2)$ . And so it will take  $O((N - 1) * N^2) = O(N^3)$  time for us to produce our multiple alignment.

When the phylogenetic tree is unknown, then we have several options in how to reconstruct the tree. Different trees that we use will lead to dramatically different alignments in terms of quality.

In general, we lose information and accuracy whenever we align a pair of sequences into a profile. The more they have diverged, the more information and accuracy is lost. At one extreme, if the two sequences are almost identical (little evolutionary change), then we may obtain a perfect alignment. At another extreme, if two sequences are completely different from each other (due to billions of years of evolving apart, recall Jukes-Cantor model), then the alignment accuracy that we get is the same as that of aligning two pairs of random sequences (very small).

So, we want to align the most similar sequences first, as to minimize errors. The errors will propagate up the tree as we do progressive alignment. This encourages us to construct a good phylogenetic tree, one that will place similar sequences close together.

One way to do this is to perform all pair-wise alignments. Define  $D(x, y)$  as a measure of the evolutionary distance between  $x$  and  $y$ , based on pair-wise alignment. Then, construct our phylogenetic tree with a preferred tree-building procedure (UPGMA, Neighbor Joining, or other methods). Finally, perform progressive alignment based on the tree.

In practice, Neighbor Joining will result in more correct trees in terms of evolution, but it will first join sequences that are not optimally close to each other. There is experimental evidence that performing progressive alignment based on trees construct by UPGMA may lead to more accurate multiple sequence alignment. So, we may want to construct the tree with UPGMA, perform progressive alignment, and then construct a more correct phylogenetic tree with Neighbor Joining.

There are several ways of improving the multiple sequence alignment that we obtain. We will discuss a few of them below.

### Iterative Refinement

Under our current algorithm, as soon as we align two sequences, the choices that we made during alignment are fixed. Any mistakes that we make will propagate and accumulate up the tree. Therefore it is desirable to be able go back and correct them later.

An example:

We first align  $x$  and  $y$  into a fixed profile, and then align  $z$  and  $w$  into a fixed profile.

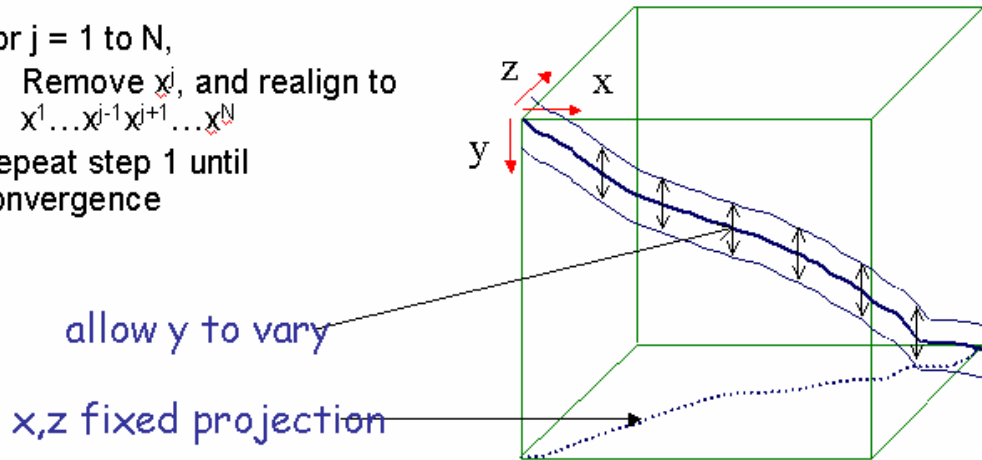
<b>x :</b>	<b>GAAGTT</b>		<b>Frozen!</b>
<b>y :</b>	<b>GAC-TT</b>		
<b>z :</b>	<b>GAACTG</b>		<b>Now clear correct y = G-ACTT</b>
<b>w :</b>	<b>GTACTG</b>		

The alignment given between  $x$  and  $y$  seemed quite reasonable when we were aligning them. However, we realize that the alignment is clearly not optimal after we have seen  $z$  and  $w$ . The nucleotides "AC" should have been shifted down one position. Sadly, there is no way we can fix this under the progressive alignment algorithm. The profiles are fixed as soon as they are produced.

We may remedy this problem to some extent, after multiple sequence alignment has been performed, via the **Barton-Stenberg algorithm**:

**Algorithm (Barton-Stenberg):**

1. For  $j = 1$  to  $N$ ,  
     Remove  $x^j$ , and realign to  
      $x^1 \dots x^{j-1} x^{j+1} \dots x^N$
2. Repeat step 1 until convergence



We may visualize the algorithm as making adjustments to a path in  $N$ -space. For each sequence  $x_j$ , we fix all other dimensions and allow variations in the dimension associated with  $x_j$ . We may do less work and restrict the amount of space within which  $x_j$  is allowed to change. In other words, we may change a sequence if and only if the resulting sequence is within some radius  $r$  (in  $N$ -space) of the original sequence. This will effectively give us a linear time algorithm.

An example of the Barton-Stenberg algorithm in action:

Example: align  $(x,y)$ ,  $(z,w)$ ,  $(x,y, z,w)$ :

```

x:   GAAGTTA
y:   GAC-TTA
z:   GAACTGA
w:   GTACTGA

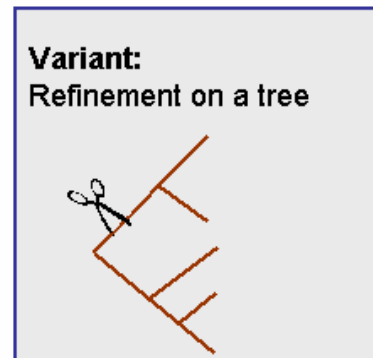
```

After realigning  $y$ : **+ 3 matches**

```

x:   GAAGTTA
y:   G-ACTTA
z:   GAACTGA
w:   GTACTGA

```



A variant of the Barton-Stenberg algorithm is, given a phylogenetic tree, instead of realigning one sequence at a time, we cut along a branch of the tree, project the alignments into two profiles across the cut, and iterate over all branches of the tree. Thus, the original Barton-Stenberg algorithm is the case where we only cut the leaves of the tree.

This does not solve the NP-hard problem of finding the optimal multiple sequence alignment, however. An example where the Barton-Stenberg algorithm breaks down:

**Example not handled well:**

x: GAAGTTA  
 Y<sub>1</sub>: GAC-TTA  
 Y<sub>2</sub>: GAC-TTA  
 Y<sub>3</sub>: GAC-TTA  
 z: GAACTGA  
 w: GTRACTGA

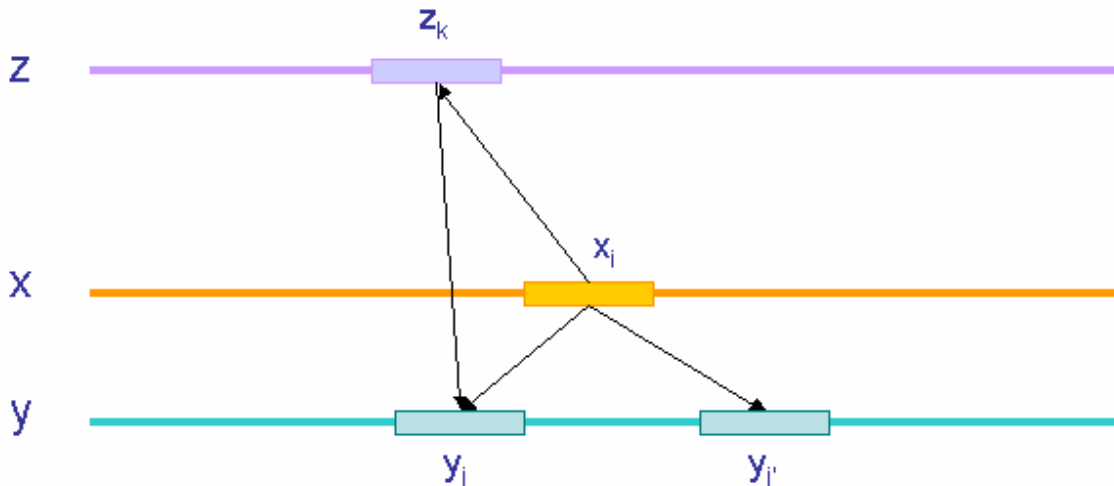
Realigning any single y<sub>i</sub> changes nothing

The nucleotides "AC" in each y<sub>i</sub> sequence need to be shifted one position to the right. Re-aligning any single y<sub>i</sub> changes nothing since the other two sequences, y<sub>j</sub> and y<sub>k</sub>, will attract the sequence in the wrong direction. If we're lucky, and all y sequences are all in the same sub-tree, then the variant we've discussed will work for this case, since all three y sequences will be re-aligned when they are cut from the tree. If we're unlucky and the y sequences are not in the same sub-tree, then we may not be able to solve this problem using iterative refinement.

Iterative alignment does not only apply to locate areas, as the examples shown would suggest. In principle, the hard part of multiple sequence alignment is aligning long genomic sequences, were much longer shifts could occur. Iterative alignment works for longer cases as well.

**Consistency**

Another scheme used by several programs to improve multiples sequence alignments is consistency. When aligning sequences x and y, the alignment procedure makes a lot of decisions where the choice is non-obvious. An example:



If the alignment scores between  $x$  and  $y_j$  and between  $x$  and  $y_j'$  are almost equal, it is not clear whether segment  $x_i$  (of any size) should be matched with  $y_j$  and  $y_j'$ . We have a probability of making a mistake whichever choice we make. The alignment algorithm will be making many decisions like this one, and thus is bound to make quite a few mistakes.

However, suppose we have another sequence  $z$ . If we align  $x$  and  $z$ , then the alignment algorithm would match the segment  $x_i$  to  $z_k$ . If we align  $y$  and  $z$ , then the alignment algorithm would match the segment  $z_k$  to  $y_j$ , and not  $y_j'$ . Obviously, the segment  $x_i$  should be aligned with  $y_j$  in this case.

So the main idea of consistency during progressive alignment is this: while aligning two sequences, we consider third sequences. We examine the pair-wise alignments between the current two sequences to those third sequences, and look for preferences suggested by the third sequence.

### Basic method for applying consistency

- **Compute all pairs of alignments  $xy$ ,  $xz$ ,  $yz$ , ...**
- **When aligning  $x$ ,  $y$  during progressive alignment,**
  - § For each  $(x_i, y_j)$ , let  $s(x_i, y_j) = \text{function\_of}(x_i, y_j, a_{xz}, a_{yz})$
  - § Align  $x$  and  $y$  with DP using the modified  $s(.,.)$  function

To be more precise, the value of  $s(x_i, y_j)$  should be increased if the  $z$  sequence provides evidence supporting this match ( $x_i$  and  $y_j$  both align to the same position in  $z$ ). The value should be decreased otherwise ( $x_i$  and  $y_j$  align to different positions in  $z$ ).

The first program to make use of consistency showed dramatic increase in accuracy over previous programs in the multiple sequence alignment of proteins.

### Some Resources

#### *Genome Resources*

Annotation and alignment genome browser at UCSC  
<http://genome.ucsc.edu/cgi-bin/hgGateway>

Specialized VISTA alignment browser at LBNL  
<http://pipeline.lbl.gov/cgi-bin/gateway2>

#### *Protein Multiple Aligners*

<http://www.ebi.ac.uk/clustalw/>  
CLUSTALW – most widely used

[http://phylogenomics.berkeley.edu/cgi-bin/muscle/input\\_muscle.py](http://phylogenomics.berkeley.edu/cgi-bin/muscle/input_muscle.py)

MUSCLE – most scalable

<http://probcons.stanford.edu/>

PROBCONS – most accurate

MUSCLE at a glance:

1. Fast measurement of all pairwise distances between sequences
  - $D_{\text{DRAFT}}(x, y)$  defined in terms of # common k-mers ( $k \sim 3$ ) –  $O(N^2 L \log L)$  time
2. Build tree  $T_{\text{DRAFT}}$  based on those distances, with UPGMA
3. Progressive alignment over  $T_{\text{DRAFT}}$ , resulting in multiple alignment  $M_{\text{DRAFT}}$ 
  - Only perform alignment steps for the parts of the tree that have changed
4. Measure new Kimura-based distances  $D(x, y)$  based on  $M_{\text{DRAFT}}$
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

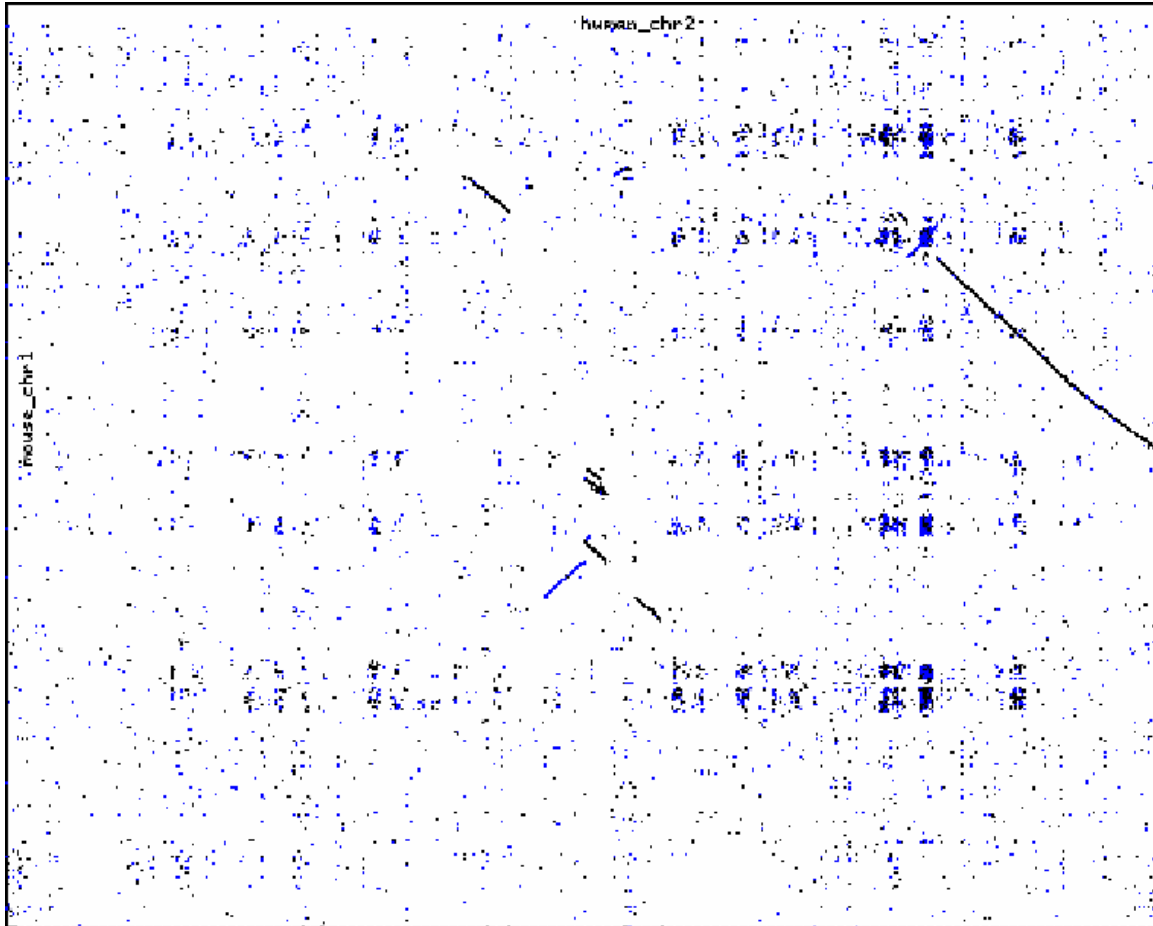
PROBCONS at a glance:

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*
  - $M'_{xy}(i, j) = 1/N \sum_{\text{sequence } z} \sum_k M_{xz}(i, k) \times M_{yz}(j, k)$ ;  $M'_{xy} = \text{Avg}_z(M_{xz}M_{zy})$
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

## II. Rapid Global Alignments

### Motivation

Here are all the sequence similarities between mouse chromosome 1 and human chromosome 2 that a program like BLAST would report:



Every dot that we see on the plot corresponds to a position where BLAST matched a k-mer between the two sequences, and extended the match to a significantly high-scoring alignment. Black dots correspond to forward alignments. Blue dots correspond to alignments between one chromosome and the reverse complement of another. Thus, black alignments go from top left to bottom right, while blue alignments go from bottom left to top right.

We know that there are regions within the two chromosomes that have evolved from the same region in the common ancestor species. However, the plot seems to suggest that the two chromosomes have been completely shuffled in relation to each other, causing a lot of noise to appear in the plot.

It's difficult to believe that all those dots on the plot are significant. The checkered pattern of the noise suggests that certain regions in one chromosome are repeatedly matched to regions in the other chromosome. Those are really repeats.

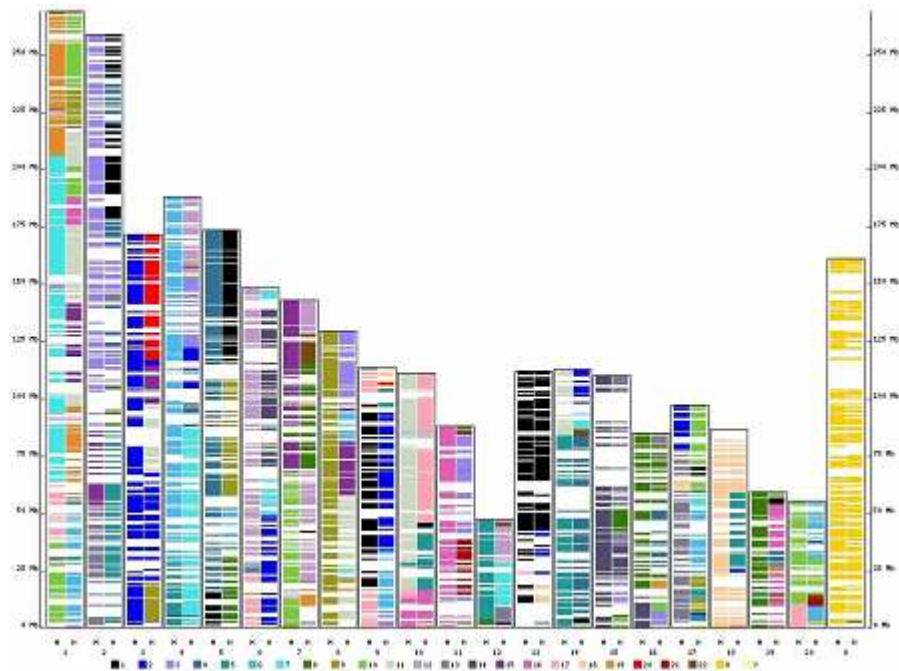
BLAST is relatively efficient, but the resulting pair-wise local alignments do not truly correspond to the original regions in the common ancestor species. Instead, much of what BLAST is detecting are the duplications that happened both before and after the two species diverged.

Nevertheless, there are some hints as to what the true common ancestor regions were. For example, there is a long black line at the right section of the plot.

Further motivation for rapid pair-wise global alignment:

- Genomic sequences are very long: § Human genome =  $3 \times 10^9$  –long  
 § Mouse genome =  $2.7 \times 10^9$  –long
- Aligning genomic regions is useful for revealing common gene structure § Useful to compare regions > 1,000,000-long However, with respect to this problem, the cloud of noise that we get from the previous plot is really not helpful at all.

Here, we see a color-coded comparison of the rat genome to the mouse genome on the left and the human genome on the right:



We see that common regions within the chromosomes come in huge blocks. The regions that we would like to align are millions of bases long or longer. So what algorithm should we apply?

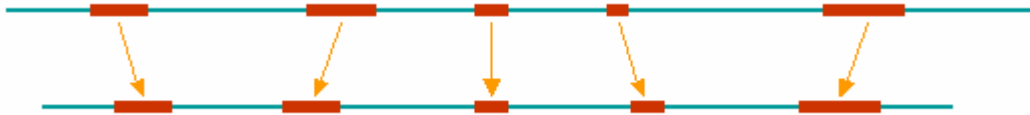
The Needleman-Wunsch global alignment algorithm takes quadratic time, which is not an acceptable running time in this case.

BLAST gives us all local alignments, in which repeats generates a lot of noise that makes the resulting plot confusing.

So what do we do?

### The Main Idea

Genomic regions of interest contain islands of similarity, such as exons in genes and regulatory regions controlling the expression of the genes.



So, in order to obtain a filtered version of the similarities between two genomes, one that will reveal the common ancestor regions, we should:

1. Find local alignments
2. Chain an optimal subset of them
3. Refine/complete the alignment

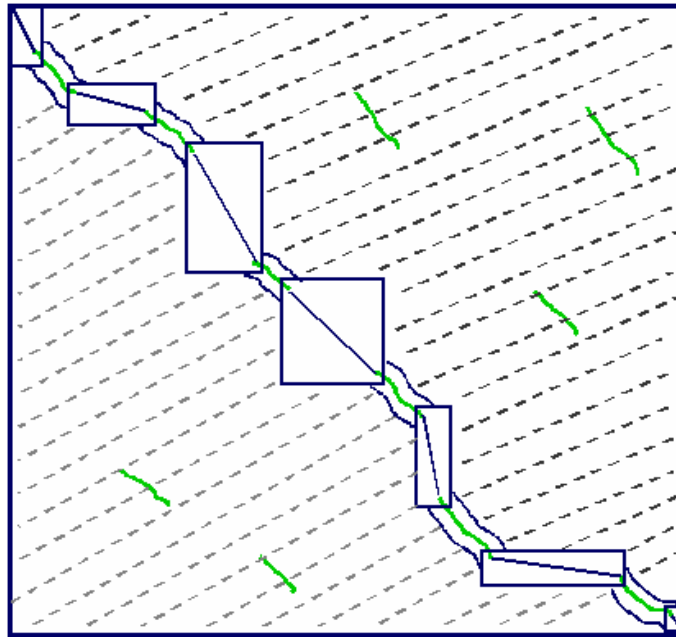
Systems that use this idea to various degrees include: MUMmer, GLASS, DIALIGN, CHAOS, AVID, LAGAN, TBA, and others.

To be more precise:

1. Find all local alignments, with any fast algorithm.
2. Chain the local alignments in  $O(N \log N)$  time, where  $N$  is the number of local alignments. The goal is to find a "heavy" chain of local alignments. In other words, the algorithm will find an ordered set of local alignments with the maximum score, where the score is defined as the sum of the scores of the local alignments we're chaining together.

We want a fast algorithm here, since we expect to get millions to trillions of local alignments from long genomes, most of them due to repeats.

3. Finally, we assume that our global alignment will be close to the chain constructed at step 2. So we restrict our dynamic programming procedure when inside links of the chain of local alignments, while allowing it to vary as much as possible while outside the chain:



So, we may run Needleman-Wunsch, while allowing a full Needleman-Wunsch box between each link in the chain, but allowing each link region to only vary only a little. Using appropriate parameters, the space that we take under consideration will be proportional to a large constant times the length of one of the sequences, and NOT the product of the two sequence lengths.

### **Sparse Dynamic Programming**

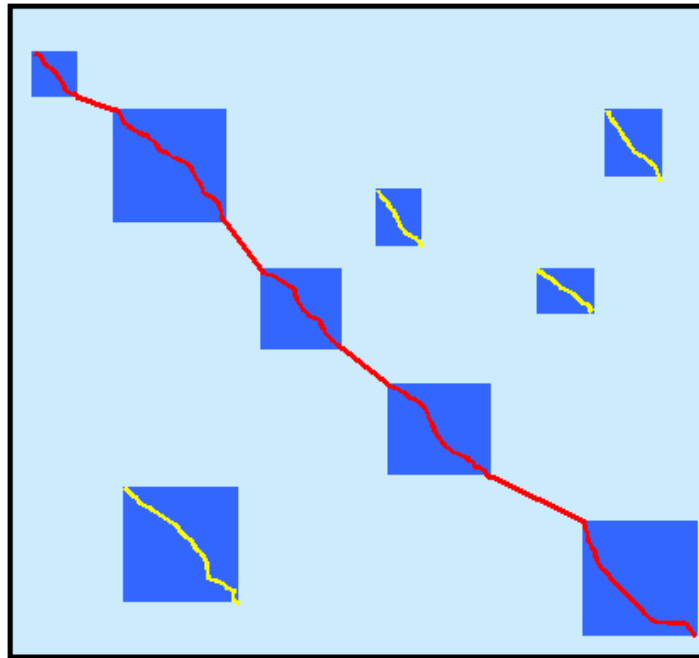
So how should be chain the local alignments? This will be tricky, so we begin with a precise definition of the problem, and then a few intermediate algorithms. The problem is this:

We have a set of local alignments. Each one of them is defined by two boundaries: the top left corner and the bottom right corner.

A convention that we have assumed in this course is that alignments begin in the top left corner and end in the bottom right corner. A consequence of this is that the x-coordinate in a two dimensional graph increases from left to right, as expected, but the y-coordinate in a two dimensional graph increases from top to bottom, opposite of most conventions.

Each local alignment has an associated weight, or score. A possible score would be how many bases, or letters, were matched in that alignment. Another would be a weighted sum of how many letters were matched, how many were gapped, how many were mismatched, etc.

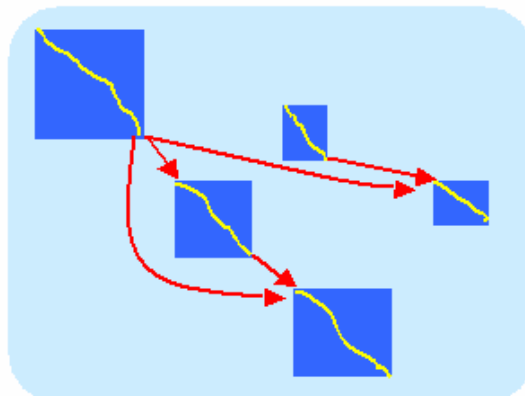
What we want is an ordered chain of local alignments, with the highest total score:



And, we want to do this in time  $O(N \log N)$ , where  $N$  is the number of boxes. But first, let us examine a quadratic time solution:

- Build Directed Acyclic Graph (DAG):
  - Nodes: local alignments  $[(x_a, x_b) \rightarrow (y_a, y_b)]$  & score
  - Directed edges: local alignments that can be chained
    - edge  $((x_a, x_b, y_a, y_b), (x_c, x_d, y_c, y_d))$
    - $x_a < x_b < x_c < x_d$
    - $y_a < y_b < y_c < y_d$

Each local alignment is a node  $v_i$  with alignment score  $s_i$



To clarify, we may construct an edge from the first node to the second one if the bottom right corner of first node is to the top left of the top left corner of second node. In other words, if the x-coordinate of the bottom right corner of the first node is *smaller* than the

corresponding x-coordinate of the top left corner of the second node, and the same is true for the y-coordinate.

Now that we have the DAG, we search for the heaviest path in this DAG, as follows:

**Initialization:**

Find each node  $v_a$  s.t. there is no edge  $(u, v_a)$

Set score of  $V(a)$  to be  $s_a$

**Iteration:**

For each  $v_i$ , optimal path ending in  $v_i$  has total score:  $V(i) = \max_j$  s.t. there is edge  $(v_j, v_i)$  (  $\text{weight}(v_j, v_i) + V(j)$  )

To clarify, the function  $\text{weight}(v_j, v_i)$  may simply return the score of node  $v_j$ , or something more elaborate, such as implementing a penalty for the gap distance between the two nodes.

**Termination:**

Optimal global chain:

$j = \text{argmax} ( V(j) );$  trace chain from  $v_j$

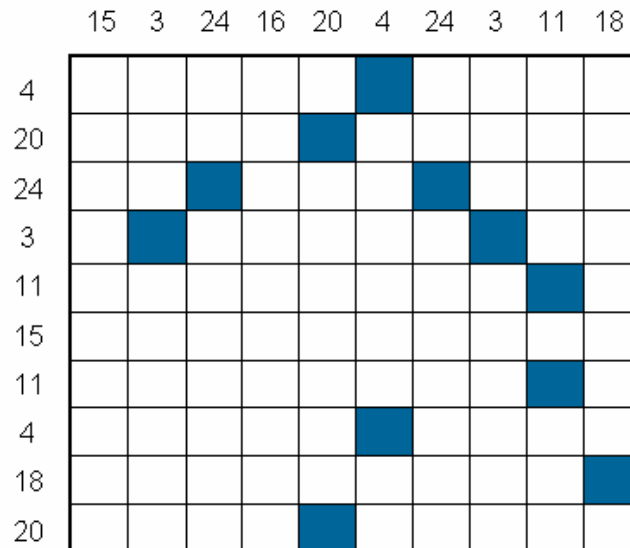
The worst case running time of this algorithm is quadratic in the number of local alignments, because we examine  $O(N)$  edges for each of the  $N$  nodes.

A subtle point is that, for this quadratic time algorithm, the  $\text{weight}(v_j, v_i)$  function may take quadratic time as well. However, for the faster algorithms that we will examine later, the weight function will have to be faster than quadratic.

The Sparse Dynamic Programming problem is really the **longest increasing subsequence problem**. For motivation, let us review the **longest common increasing subset problem**:

- Given two sequences
  - $x = x_1, \dots, x_m$
  - $y = y_1, \dots, y_n$
- Find the longest common subsequence
  - Quadratic time solution with DP
  - Apply global alignment with  $\text{match} = 1, \text{mismatch} = 0, \text{gap} = 0$
- How about when “hits”  $x_i = y_j$  are sparse?

The hits are more likely to be sparse if the alphabet is a large set, perhaps the set of all integers, since the hits are unlikely to happen by chance. An example:



- Imagine a situation where the number of hits is much smaller than  $O(nm)$  – maybe  $O(n)$  instead

Only the blue positions are matches. Intuitively, we are doing a lot of work (quadratic) compared to the number of blue elements / cells. It would be nice, in this case, to have a procedure that find the longest increasing subsequence with running time  $O(N \log N)$ , where  $N$  is the number of blue elements.

Of course, if a large fraction of the elements were blue, then  $O(N \log N)$  in the number of blue elements would be  $O(N^2 \log N)$  in the length of the sequences, which would be worse than quadratic. But, if the number of blue elements were a tiny fraction of the total number of elements, then  $O(N \log N)$  in the number of blue elements would be much better than quadratic.

This situation may remind you a lot of the chaining rectangles situation. We will re-examine this later.

Let us return to the longest increasing subsequence problem:

- Given a single sequence over an ordered alphabet
    - $x = x_1, \dots, x_m$
  - Find longest subsequence such that
    - $s = s_1, \dots, s_k$
    - $s_1 < s_2 < \dots < s_k$
- Here is a (deceptively simple!) algorithm that does this in time  $O(N \log N)$ , where  $N$  is the length of our original sequence:

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

**L:** 1-indexed array,  $L[1] \leftarrow w_1$   
**B:** 0-indexed array of back-pointers;  $B[0] = 0$

**P:** array used for traceback

//  $L[j]$ : smallest last element  $w_i$  of  $j$ -long longest

// increasing subsequence seen so far.

//  $w_1$  is currently a longest increasing subsequence of

// length 1.

// The L array is always sorted. **ALGORITHM**

```
for i = 2 to n {
```

```
    Find j such that  $L[j] < w[i] \leq L[j+1]$      $L[j+1] \leftarrow w[i]$ 
```

```
     $B[j+1] \leftarrow i$ 
```

```
     $P[i] \leftarrow B[j]$ 
```

```
}
```

The running time of this algorithm is  $O(N \log N)$ , since the for loop iterates  $O(N)$  times, and we can find  $j$  such that  $L[j] < w[i] \leq L[j+1]$  in  $\log N$  time and the rest of the for loop body in constant time. (L is always sorted.)

This element *will* find the longest increasing subsequence, but this is not obvious. Let us use an example to show that at least the properties that we assert on the L array are maintained.

Given the sequence: **20, 2, 10, 8, 4, 15, 1**

The L array will progress as follows:

**20**

**2**

**2, 10**

**2, 8**

(Note that, we replaced 10 with 8 because we thereby increased the possible range of the next element seen such that we would extend the longest increasing subsequence.)

**2, 4**

**2, 4, 15**

**1, 4, 15**

(Also note that, the 1 did not replace 2 as a part of the longest increasing subsequence. The actual sequence is kept track of by the B and P arrays.)

We see that  $L[j]$  is indeed the smallest last element  $w_i$  of  $j$ -long longest increasing subsequence seen so far. Also, L is always sorted. The resulting longest increasing sequence is  $\{2, 4, 15\}$ .