CS 262 Lecture Notes #9

Finding Common Gene Structure:
Suffix Trees, Sparse Dynamic Programming and Multiple Alignments

Michael Kassoff, Carla Pinon

The problem at hand is the following: given two long genomic regions, efficiently find one or more conserved genes, such that their order is preserved. Na vely, we can find the optimal global alignment using Needleman-Wunsch, and examine it for stretches of high conservation. However, as the sequences may be quite long (say, greater than 1,000,000 base-pairs), Needleman-Wunsch is unacceptable. We want to be able to do this in better than $O(N^2)$ time.

One idea is to take advantage of homology between the two genomes. Genomic regions of interest contain ordered islands of similarity (e.g, genes). Therefore we can find local alignments and chain an optimal subset of them together. This works well if the two genomes are related. We can use suffix trees to find the local alignments, and then use sparse dynamic programming to chain the alignments together. We will see that using this combination of techniques, we can solve our problem in linear time (more or less).

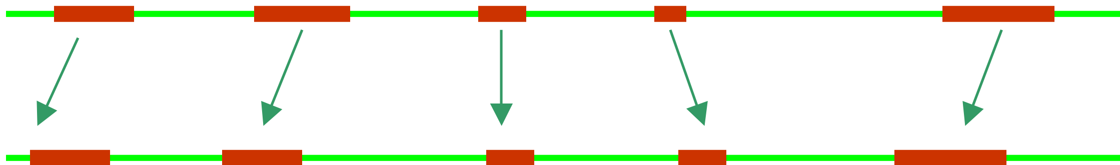


Figure 1. Ordered local alignments

Finding Local Alignments

We begin by discussing how to find local alignments. In particular, let's find common k-mers. One way to do this is sort all k-long words. That is, given sequences **x**,**y**:

1. Write down all
   (w, 0, i): $w = x_{i+1} \ldots x_{i+k}$
   (z, 1, j): $z = y_{j+1} \ldots y_{j+k}$
2. Sort them lexicographically
3. Deduce all k-long matches between x and y
4. Extend the matches to local alignments

As an example, consider all 3-mers in **x** = caggc and**y** = ggcag :

x: (cag,0,0), (agg,0,1), (ggc,0,2)
y: (ggc,1,0), (gca,1,1), (cag,1,2)

Now we sort them:

(agg,0,1), (cag,0,0), (cag,1,2), (ggc,0,2), (ggc,1,0), (gca,1,1)

We find two matches:

1. cag: $x_1x_2x_3 = y_3y_4y_5$
2. ggc: $x_3x_4x_5 = y_1y_2y_3$

There are two problems with this approach. First of all, the worst case running time is O(NM). Secondly, in practice, a large value of `k` results in a short list of matches. There is a tradeoff: as we increase `k`, we miss significant alignments, and as we decrease `k`, the running time increases.

A better way to do this is using suffix trees. Recall that suffix trees are a very efficient data structure for storing all suffixes of a string, and they support fast lookup as well.

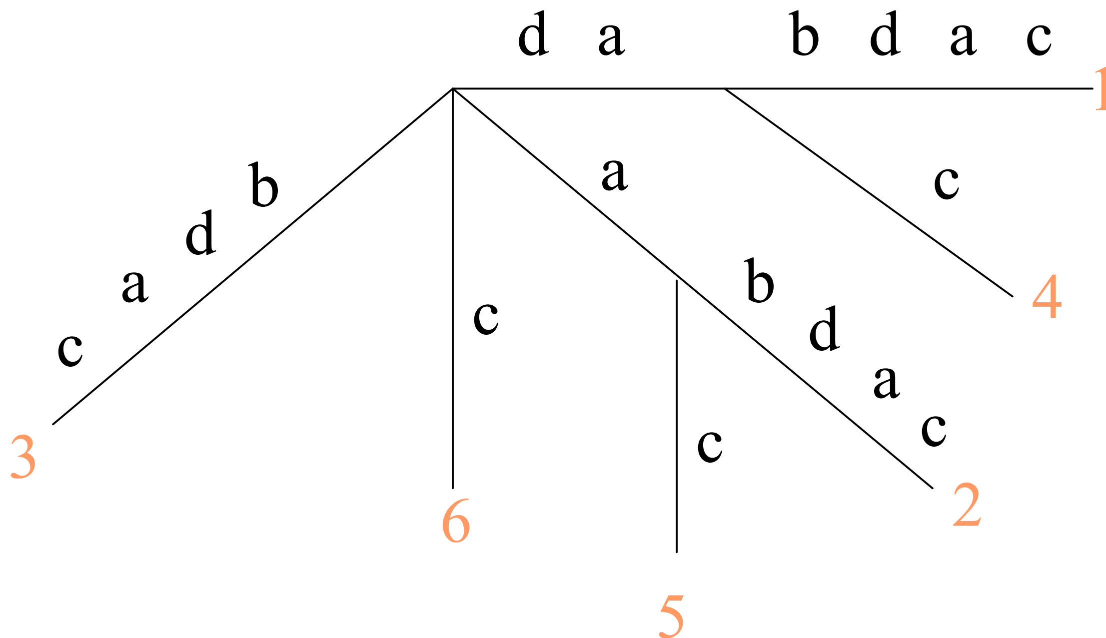


Figure 2. A suffix tree for dabdac.

The definition of a suffix tree is as follows:

A suffix tree for a string $\mathbf{x} = x_1 \ldots x_m$ is a rooted tree with m leaves labeled 1 to `m`. Each edge is labeled by a substring of $\mathbf{x}$. No two edges out of a node start with same letter. Each leaf `i` corresponds to the suffix $x_i \ldots x_m$, in the following manner: The concatenation of edge-labels on the path from the root to leaf `i` spells out $x_i \ldots x_m$.

Constructing Suffix Trees

The na ve algorithm to construct a suffix tree runs in O(N2) time. Algorithms exists that can construct a suffix tree in O(N) time. However we will not discuss the O(N) algorithms because they are highly technical and but not so deep.

The na ve algorithm for constructing a suffix tree is as follows:

Given a string $\mathbf{x} = x_1 \ldots x_m$:

1. Insert an end marker $ at end of **x**
2. Initialize the tree `T` to a single root node `r`
3. For `j` = 1 to `m`
    - Find the longest match of $x_i \ldots x_m$ to `T`, starting from `r`
    - Split the edge where the match stops, at a new node `w`
    - Create an edge (`w`,`j`), and label it the with unmatched portion of $x_i \ldots x_m$

For example, given string **x** = dabda, we would first append $ to **x**, obtaining dabda$. We would then insert the following substrings in the following order, obtaining the tree below:

1. dabda$
2. abda$
3. bda$
4. da$
5. a$
6. $

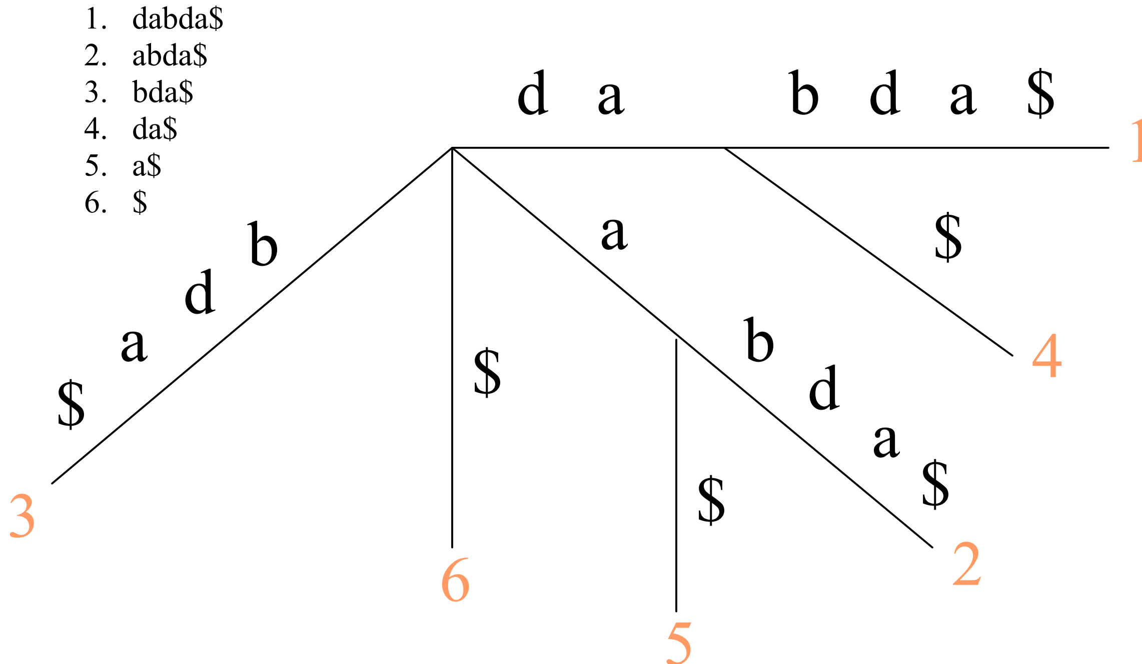


Figure 3. Suffix tree constructed from dabda$.

We can store a suffix tree in O(N) memory. Every edge is labeled with (i, j), denoting a label of $x_i \ldots x_j$. A tree has O(N) nodes, because there is exactly one leaf for each letter in **x**, and the number of leaves is greater than or equal to the number of nodes - 1.

Applications of suffix trees

One application is to find all matches between two strings **x** and **y**. We can do this as follows:

1. Build a suffix tree for **x**, mark nodes with **x**
2. Insert **y** in the suffix tree, and mark all nodes **y** “passes from with **y**

Then the path of every node marked both **x** and **y** is a common substring.

An example suffix tree containing **x** = dabda and **y** = abada is shown below.

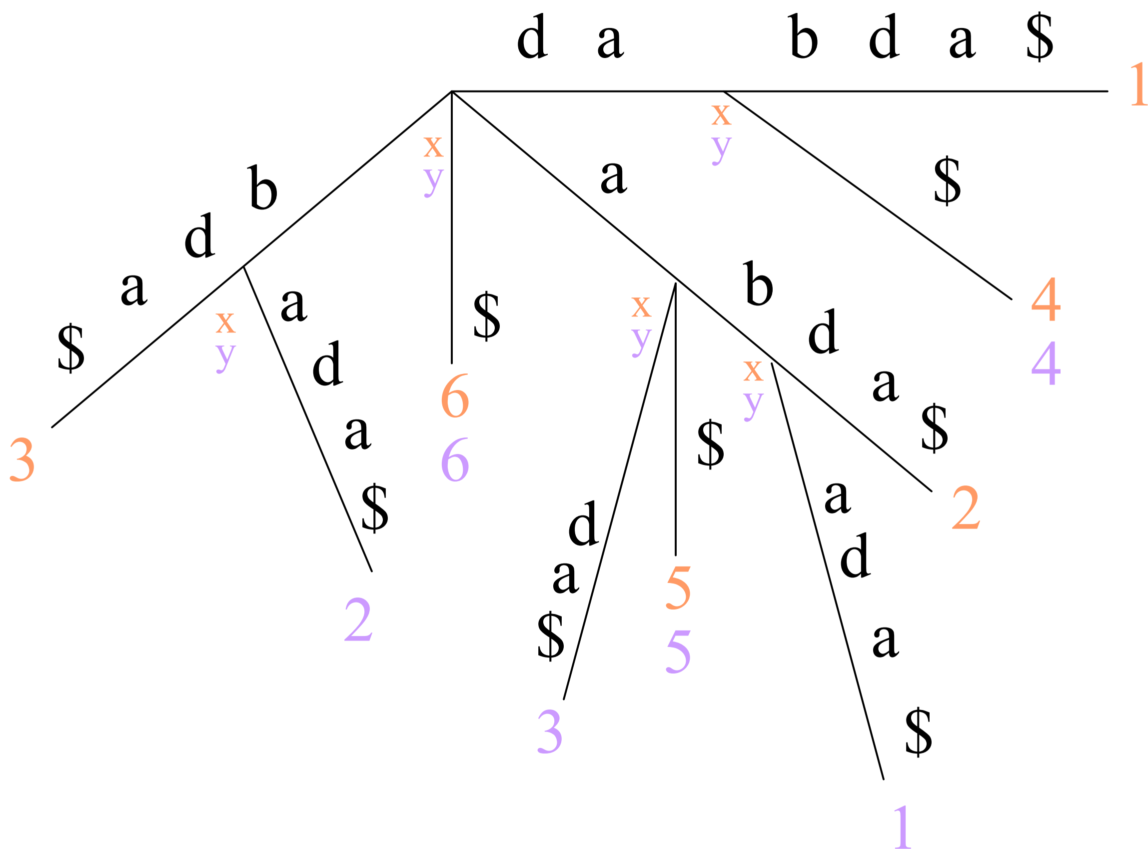


Figure 5. Joint suffix tree for **x** = dabda and **y** = abada

A second application is to perform an online search of strings in a database. Say we have a database D = { s_1, s_2, …s_n }, where s_i is, for example, a protein. Given a query string **x**,

we wish to find all matches of **x** to the database. We can accomplish this by building a suffix tree for $\{s_1,\ldots, s_n\}$, and then performing a string match as in the previous example. Then all new queries take O(|**x**|) time. If we can hold the entire database in memory, this is really fast. This is similar to the performance of BLAST.

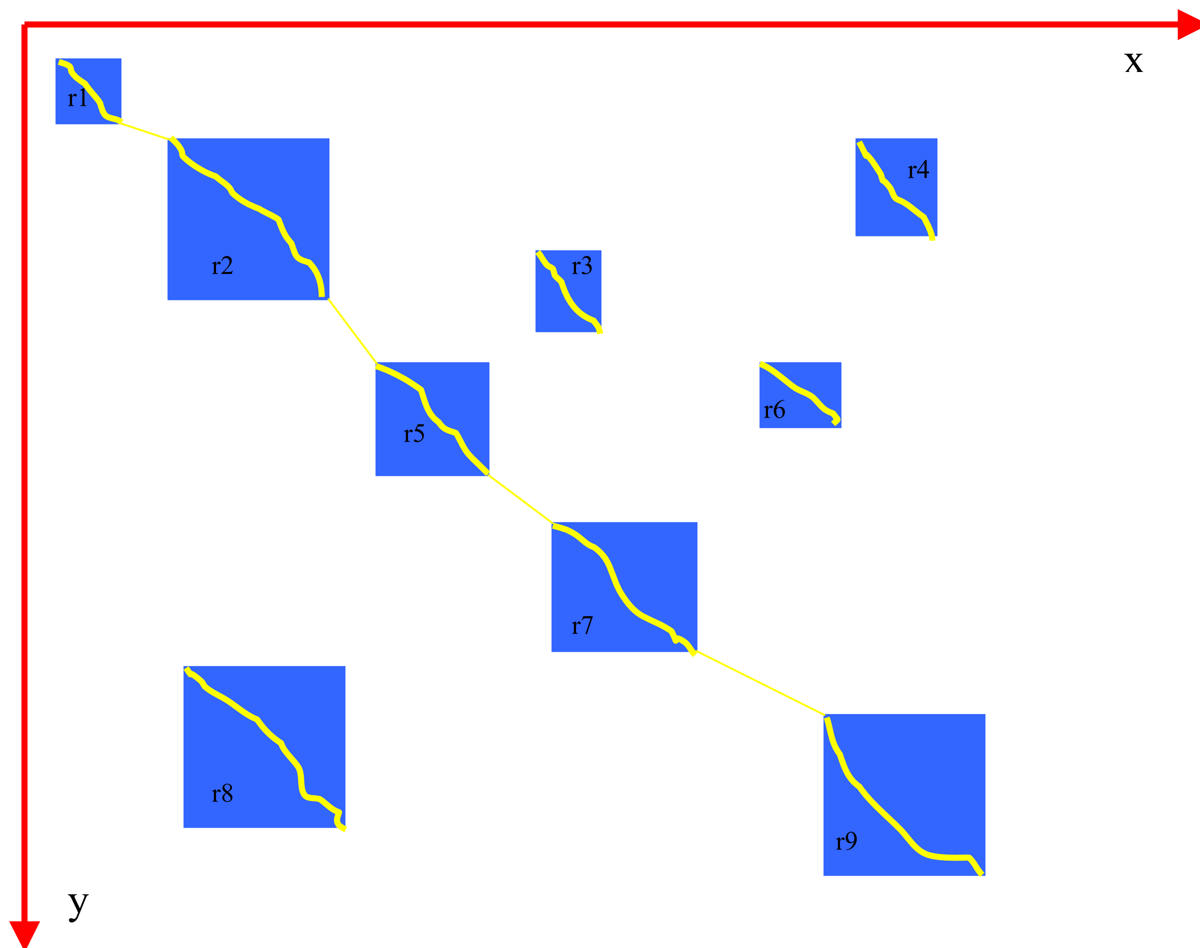
A final application is to find the longest common substring of `k` strings s_1, s_2, …s_n. We can do this as follows: Build a suffix tree for $s_1,\ldots, s_n$. Then nodes labeled $\{s_{i1}, \ldots, s_{ik}\}$ represent a match between $s_{i1}, \ldots, s_{ik}$.

Interestingly, all of these problems seemed hard until suffix trees came along.

Recall that our main problem was rapid global alignment, i.e., how to align genomic sequences in linear time (more or less). Our idea was to first find local alignments and then find the best way to merge/chain such local alignments (or an optimal subset of such local alignments). We've already discussed how to do the first part through 1) matching k-mers and 2) suffix trees. Now we turn to the second part which is the chaining of local alignments. We discuss two approaches:

1) dynamic programming
2) sparse dynamic programming

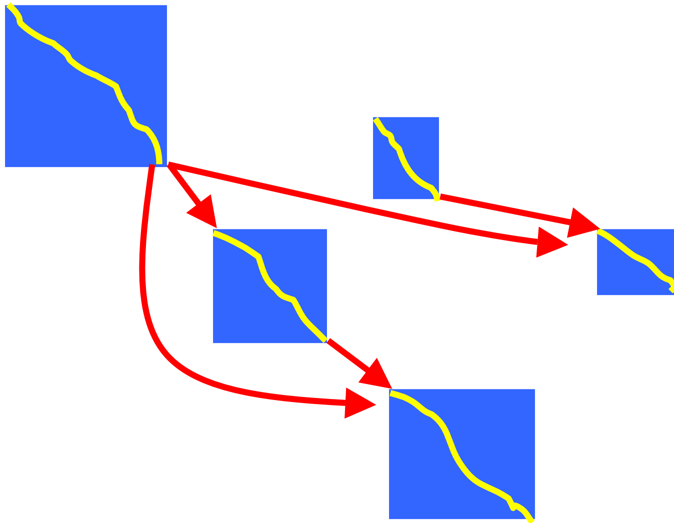
Given two genomic sequences x and y, imagine that we have identified a set of local alignments (represented by individual rectangles). Our task is to chain such local alignments optimally. Note that we can chain rectangle r2, denoted by ($x2_{tl}$, $y2_{tl}$), to rectangle r1, denoted by ($x1_{tl}$, $y1_{tl}$), only if $x1_{tl} < x2_{tl}$ and $y1_{tl} < y2_{tl}$, where (x_{tl}, y_{tl}) are the (x, y) coordinates of the rectangle's top left corner.



Thus, if we assign a weight to each local alignment/rectangle, then the problem is reduced to finding the chain of rectangles with the highest total weight. Dynamic programming is ideal for such a problem.

Dynamic Programming

We build a directed acyclic graph (DAG) whose nodes are the local alignments/rectangles (1, 2, 3, … n) and whose edges are any two rectangles that can be chained ((1, 2), (1, 3), (1, 4) … (7, 9) …).



We begin chaining by finding the start node r_0 which is the rectangle such that there are no incoming edges to it, i.e., no edge (i, r_0) $\forall$ i. Then we apply dynamic programming and observe that the score of the best possible chain that ends at a given rectangle i is the score of the best chain ending at a previous rectangle j plus the score for adding the given rectangle i to the chain. That is, for each rectangle i, the optimal chain ending in rectangle i has score:

$$V(i) = \max (V(j) + weight(j, i)), \forall \text{ previous rectangles } j$$

The optimal global chain is obtained by finding the rectangle with the best chain score and then tracing back from it:

j = argmax V(j); then trace the chain ending in rectangle j

This algorithm runs in $O(n^2)$ time, where n = number of rectangles/local alignments because it never visits a node/rectangle more than once and each node/rectangle has at most n predecessor nodes/rectangles.

Sparse Dynamic Programming

We can do better than a quadratic run time solution by preprocessing the rectangles, thereby allowing us to do a limited but focused dynamic programming as opposed to the full-blown dynamic programming above. More specifically, we sort the rectangles in increasing x coordinates and then traverse them in the same order while maintaining a list L of rectangles that we have already visited and that are on the current optimal chain. Every time we "enter a rectangle i, we pick that rectangle j in our current optimal chain list L that is the most restrictive one that we can chain to and calculate rectangle i's score based on rectangle j's score. Every time we "exit a rectangle i, we add it to our current optimal chain list L if it is less restrictive and has a higher score. We also do some bookkeeping and remove those rectangles in the current optimal chain that are more restrictive and have lower scores.

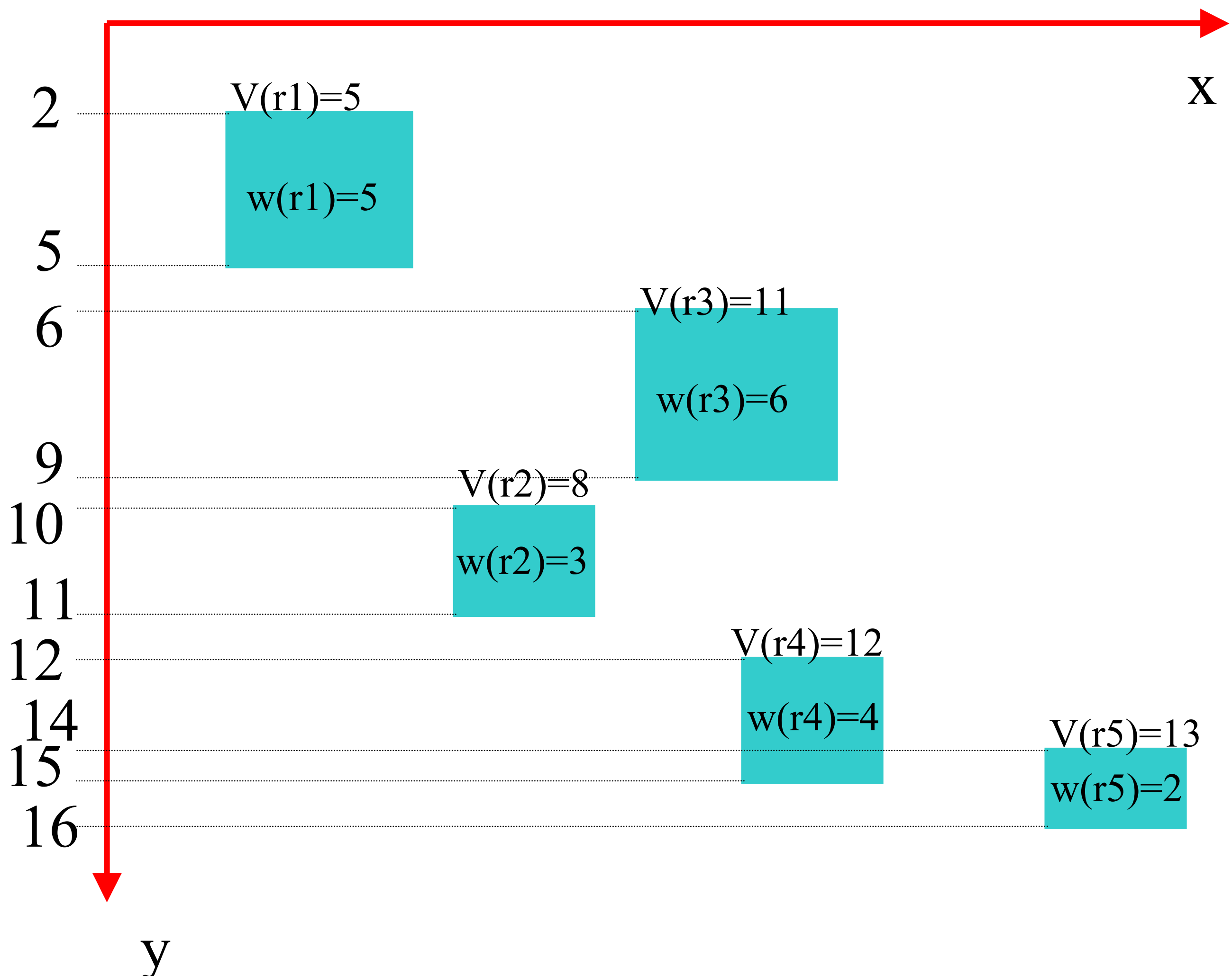
1 … n:	rectangles
(l_j, h_j):	bottom and top y-coordinates of rectangle j
w(j):	weight of rectangle j
V(j):	optimal score of chain ending in rectangle j
L:	list of triplets $(l_j, V(j), j)$; list is sorted by l_j

We traverse rectangles in increasing x coordinates. Thus, we know that every rectangle we come across can be added to our current chain only if its high y-coordinate is greater than our last low y-coordinate.

1) When we are at the leftmost end of rectangle i (i.e., we just "entered rectangle i):
   a) j = rectangle in L with the highest $l_j < h_i$ (we are picking j to be the most restrictive rectangle in our current chain that we can chain to)
   b) V(i) = V(j) + w(i)

2) When we are at the rightmost end of rectangle i (i.e., we are "exiting rectangle i):
   a) j = rectangle in L with the lowest $l_j > l_i$ (we are picking j to be the rectangle in our current chain that is next more restrictive than rectangle i)
   b) if V(i) > V(j), then
      i) insert (l_i, V(i), i) into L (we are inserting rectangle i into the current optimal chain because it is less restrictive than rectangle j – from step a – and has a better score than rectangle j)
      ii) remove all (l_k, V(k), k) from L with V(k) < V(i) and $l_k > l_i$ (we are removing from the current optimal chain all rectangles that are more restrictive and have lower chain scores than rectangle i)

Let us take the diagram below as an example and step through the sparse DP algorithm.



Enter r_i (step 1)	Exit r_i (step 2)	r_j returned (steps 1a or 2a)	V(j)	V(i) (step 1b)	V(i)>V(j)? (step 2b)	Insert into L (step 2ƀ i)	Remove from L (step 2ƀ ii)	L
r1		–	0	5				{ }
	r1	–	0	5	yes	r1	–	{ (5, 5, 1) }
r2		r1	5	8				{ (5, 5, 1) }
	r2	–	0	8	yes	r2	–	{ (5, 5, 1) (11, 8, 2) }
r3		r1	5	11				{ (5, 5, 1) (11, 8, 2) }
r4		r2	8	12				{ (5, 5, 1) (11, 8, 2) }
	r3	r2	8	11	yes	r3	r2	{ (5, 5, 1) (9, 11, 3) }
	r4	–	0	12	yes	r4	–	{ (5, 5, 1) (9, 11, 3) (15, 12, 4) }
r5		r3	11	13				{ (5, 5, 1) (9, 11, 3) (15, 12, 4) }
	r5	–	0	13	yes	r5	–	{ (5, 5, 1) (9, 11, 3) (15, 12, 4) (16, 13, 5) }

Begin with empty L = { }.

Enter r1. (step 1)

L is empty so no rectangle to return. (step 1a)

V(1) = 0 + w(1) = 5 (step 1b)

Exit r1. (step 2)

L is empty so no rectangle to return. (step 2a)

V(1) = 5 > 0 (step 2b)

Insert r1's triplet into L. (step 2b-i)

L = { (5, 5, 1) }

No rectangle is more restrictive and has a lower score than r1 so nothing to remove from L. (step 2ƀ ii)

Enter r2. (step 1)

r1 is the most restrictive rectangle in L that r2 can be chained to, so j = r1 (step 1a)

V(2) = V(1) + w(2) = 5 + 3 = 8 (step 1b)

Exit r2. (step 2)

No rectangle in L is more restrictive than r2, so no rectangle to return. (step 2a)

V(2) = 8 > 0 (step 2b)

Insert rectangle 2's triplet into L. (step 2b-i)

L = { (5, 5, 1) (11, 8, 2) }

No rectangle is more restrictive and has a lower score than r2 so nothing to remove from L. (step 2ƀ ii)

Enter r3. (step 1)

r1 is the most restrictive rectangle in L that r3 can be chained to, so j = r1 (step 1a)

V(3) = V(1) + w(3) = 5 + 6 = 11 (step 1b)

Enter r4. (step 1)

r2 is the most restrictive rectangle in L that r4 can be chained to, so j = r2 (step 1a)

V(4) = V(2) + w(4) = 8 + 4 = 12 (step 1b)

Exit r3. (step 2)

r2 is the rectangle in L that is next more restrictive than r3, so j = r2 (step 2a)

V(3) = 11 > 8 = V(2) (step 2b)

Insert r3’s triplet into L. (step 2b-i)
L = { (5, 5, 1) (9, 11, 3) (11, 8, 2) }
Remove r2’s triplet from L because it is more restrictive than r3 and has a lower score than r3. (step 2b ii)
L = { (5, 5, 1) (9, 11, 3) }
Exit r4. (step 2)
No rectangle in L is more restrictive than r4, so no rectangle to return. (step 2a)
V(4) = 12 > 0 (step 2b)
Insert r4’s triplet into L. (step 2b-i)
L = { (5, 5, 1) (9, 11, 3) (15, 12, 4) }
No rectangle is more restrictive and has a lower score than r4 so nothing to remove from L. (step 2b ii)
Enter r5. (step 1)
r3 is the most restrictive rectangle in L that r5 can be chained to, so j = r3 (step 1a)
V(5) = V(3) + w(5) = 11 + 2 = 13 (step 1b)
Exit r5. (step 2)
No rectangle in L is more restrictive than r5, so no rectangle to return. (step 2a)
V(5) = 13 > 0 (step 2b)
Insert r5’s triplet into L. (step 2b-i)
L = { (5, 5, 1) (9, 11, 3) (15, 12, 4) (16, 13, 5) }
No rectangle is more restrictive and has a lower score than r5 so nothing to remove from L. (step 2b ii)
End.

L = { (5, 5, 1) (9, 11, 3) (15, 12, 4) (16, 13, 5) }
Thus, our global chain of local alignments is r1 r3 r4 r5.

Efficiencies to note: L is maintained as a sorted list based on l_j. This gives 2 primary advantages:

1) Searching through L to obtain rectangle j (in steps 1a and 2a of the algorithm) can be done in O(log n) time.
2) Inserting (in step 2b-i) into a sorted listed also takes O(log n) time.
3) After inserting a triplet into L, deletions (in step 2b-ii) are performed precisely on those triplets that come after the newly inserted triplet because these refer to those rectangles that are more restrictive and have lower scores than the newly inserted rectangle. Thus, this means that deletions are continuous and consecutive from the point of the insertion, so again this can be done in O(log n) time.

So what is the run time complexity of this sparse DP algorithm? The preprocessing/sorting of the x-coordinates of all n rectangles takes O(n log n). We traverse the sorted x-coordinates once, from lowest to highest, thereby “visiting each rectangle once, so that translates to a total of n steps. At each such step, searching through L to obtain rectangle j takes log n time. Insertions take log n time and deletions also take log n time, as noted above. Thus, the total run time complexity is O(n log n), a far cry from the $O(n^2)$ of the full-blown dynamic programming approach.

Multiple Sequence Alignments

We can align multiple sequences, for example sequences from different organisms:

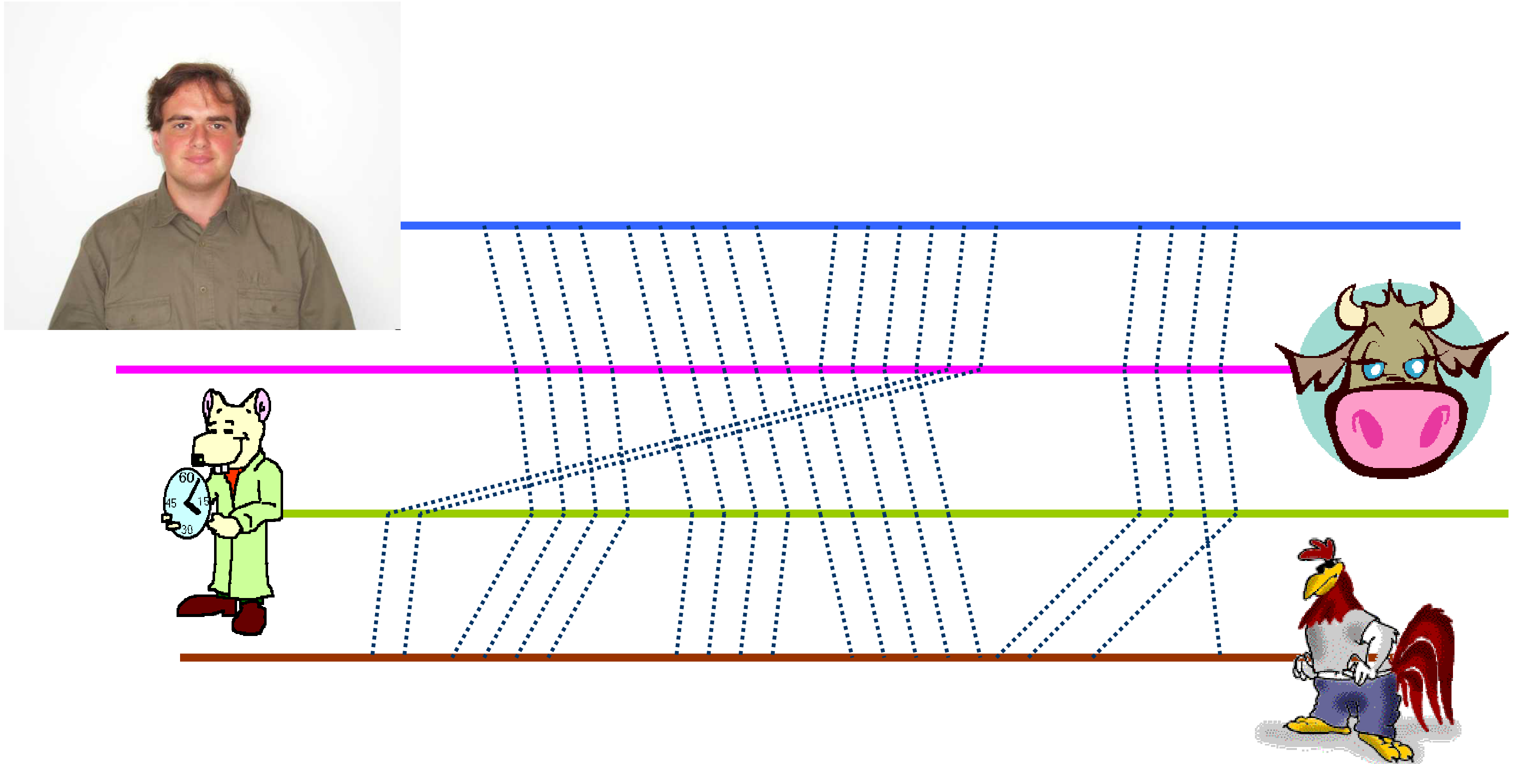


Figure 6. Perhaps Craig Venter would be more appropriate?

We may wish to do this because with many sequences, global patterns emerge, even though the pairwise alignments may be weak. That is, a faint similarity between two sequences can become statistically significant if it is present in many sequences. Multiple alignments may find conserved structure with common properties. Multiple sequence alignment has been applied to protein domains, and to finding motifs that co-regulate genes.

Formally, we define a multiple alignment as follows. Given N sequences $x^1, x^2, \ldots, x^N$, insert gaps (-) in each sequence x^i such that all sequences have the same length L and the score of the global map is maximum.

We can regard a pairwise alignment as a hypothesis on the evolutionary relationship between the letters of two sequences. The same holds for a multiple alignment.

Scoring Multiple Alignments

Ideally, we would like to find an alignment that maximizes the probability that the sequences evolved from a common ancestor.

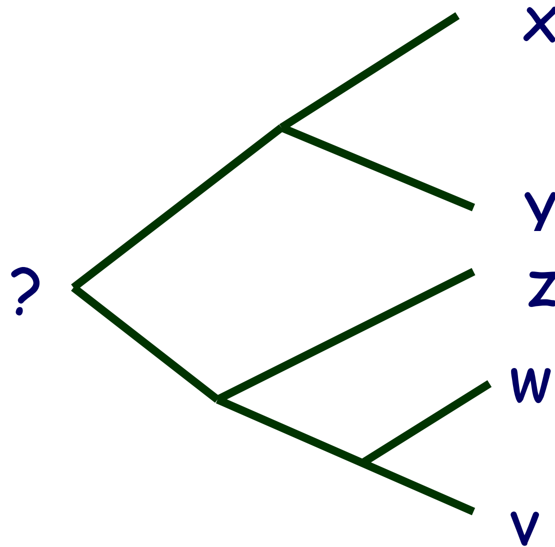


Figure 7. Our sequences evolved from some common ancestor.

Unfortunately, the evolutionary relationship between the strings is complicated. Finding a maximum likelihood alignment is an unsolved problem. There are simply too many parameters to deal with. Therefore we look at heuristics thatare not statistically sound. One common compromise is to ignore the phylogenetic tree. Another is to assume that the columns in the alignment are statistically independent. Under this assumption, our global score decomposes:

$$S(m) = G(m) + \sum_i S(m_i)$$

where `m` is the alignment matrix, and `G` is a function penalizing gaps.

An *induced pairwise alignment* be a pairwise alignment induced by the multiple alignment. For example, take the following multiple alignment:

```
x:    AC̶GCGG̶C
y:    AC̶GC̶GAG
z:    GCCGC̶GAG
```

It induces three pairwise alignments:

```
x: ACGCGG̶C x: AC̶GCGG̶C y: AC̶GCGAG
y: ACGC̶GAC z: GCCGC̶GAG z: GCCGCGAG
```

The *sum-of-pairs score* of an alignment is the sum of the scores of all induced pairwise alignments:

$$S(m) = \sum_{k<l} s(m^k, m^l)$$

where $s(m^k, m^l)$ is the score of induced alignment (`k,l`).

This is not a great approximation because it ignores the phylogenetic tree:

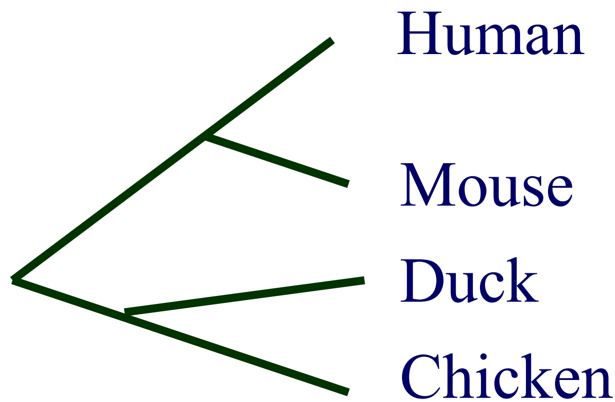


Figure 8. A phylogenetic tree. The distance between a human and a chicken is greater than the distance between a duck and a chicken.

We can heuristically incorporate the tree with a weighted sum-of-pairs score:

$$S(m) = \sum_{k<l} w_{kl}\, s(m^k, m^l)$$

where w_{kl} is the weight, which decreases with the evolutionary distance.