

CS262 Computational Genomics

Lecture 13: Phylogeny Trees and Multiple Sequence Alignment

Professor Serafim Batzoglou

Scribed by Ari Greenberg

1. Review of Phylogeny Trees from last lecture

It is often important to infer the correct phylogenetic tree between a set of organisms. There are many ways to do this. One obvious way is by morphology and placing organisms which appear to be similar closer together in the tree. For example, it is obvious that humans and monkeys are closer than humans and fish, so we could put primates together and fish outside in a tree. However, creating trees based on morphology can lead to mistakes as we saw in the last lecture.

A better alternative to morphology is to use sequence information. If we have a multiple alignment without gaps of highly conserved regions of a set of organisms, then we can place species together according to how similar the sequences are to each other.

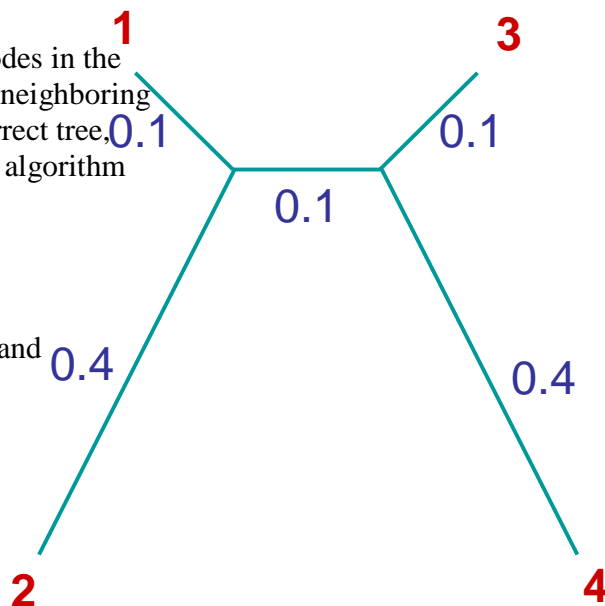
The UPGMA method can be used to reconstruct the phylogenetic tree of a set of organisms. This method requires that we have pairwise distances between each pair of organisms and that the distances obey the **ultrametric** property: for any three organisms, two of the distances are equal and larger than (or equal to) the third. These distances are related to the molecular clock -- the distance between two organisms is exactly the geological time, and current organisms all reside at the bottom of the tree (time 0). Neighbor-joining is an alternative algorithm which is guaranteed to produce the correct tree if the distances are **additive**. Distances are additive if the pairwise distances between any two organisms simply by adding up the distances of each branch along the tree. Even if the distances are not additive, neighbor-joining often produces a good quality tree.

In the tree on the right, the two closest nodes in the tree are 1 and 3, however 1 and 3 are not neighboring leaves. Neighbor-joining will find the correct tree, assuming that distances are additive. The algorithm works as follows:

$$D_{ij} = d_{ij} - (r_i + r_j)$$

where d_{ij} is the distance between i and j , and

$$r_i = \frac{1}{|L| - 2} \sum_k d_{ik}$$



Once we adjust the pairwise distances in this way, then the two nodes i and j for which D_{ij} is minimized are guaranteed to be neighbors in the phylogenetic tree. In the tree above, D_{12} and D_{34} are minimum, while D_{13} is not.

Four-Point Condition: The pairwise distances between a set of organisms are additive if and only if for every set of four leaves i, j, k, l , two of the following three sums are equal and larger than the third: $D_{ij} + D_{kl}$, $D_{ik} + D_{jl}$, and $D_{il} + D_{jk}$.

If the four-point condition is true for any pairwise distances among a set of organisms, then those distances are guaranteed to be additive, and we can use the neighbor-joining algorithm to construct the correct tree.

Neighbor-Joining Algorithm

Initialization:

Define T to be the set of leaf nodes, one per sequence
Let $L = T$

Iteration:

Pick i, j such that the adjusted distance D_{ij} is minimal
Join i and j by creating a new node k and set d_{km} for all other nodes m
to $1/2 (d_{im} + d_{jm} - d_{ij})$, $d_{jk} = d_{ij} - d_{ik}$
Add k to T , with edges of lengths $d_{ik} = 1/2 (d_{ij} + r_i - r_j)$, $d_{jk} = d_{ij} - d_{ik}$
Remove i, j from L ;
Add k to L

Termination:

When L consists of two nodes, i, j , and the edge between them of length d_{ij}

2. Parsimony

Parsimony is a popular method for creating phylogenetic trees because it is fast and simple. The problem of finding the *best* phylogenetic tree is known to be NP-hard, so a simple algorithm that gets generally good results is often preferred.

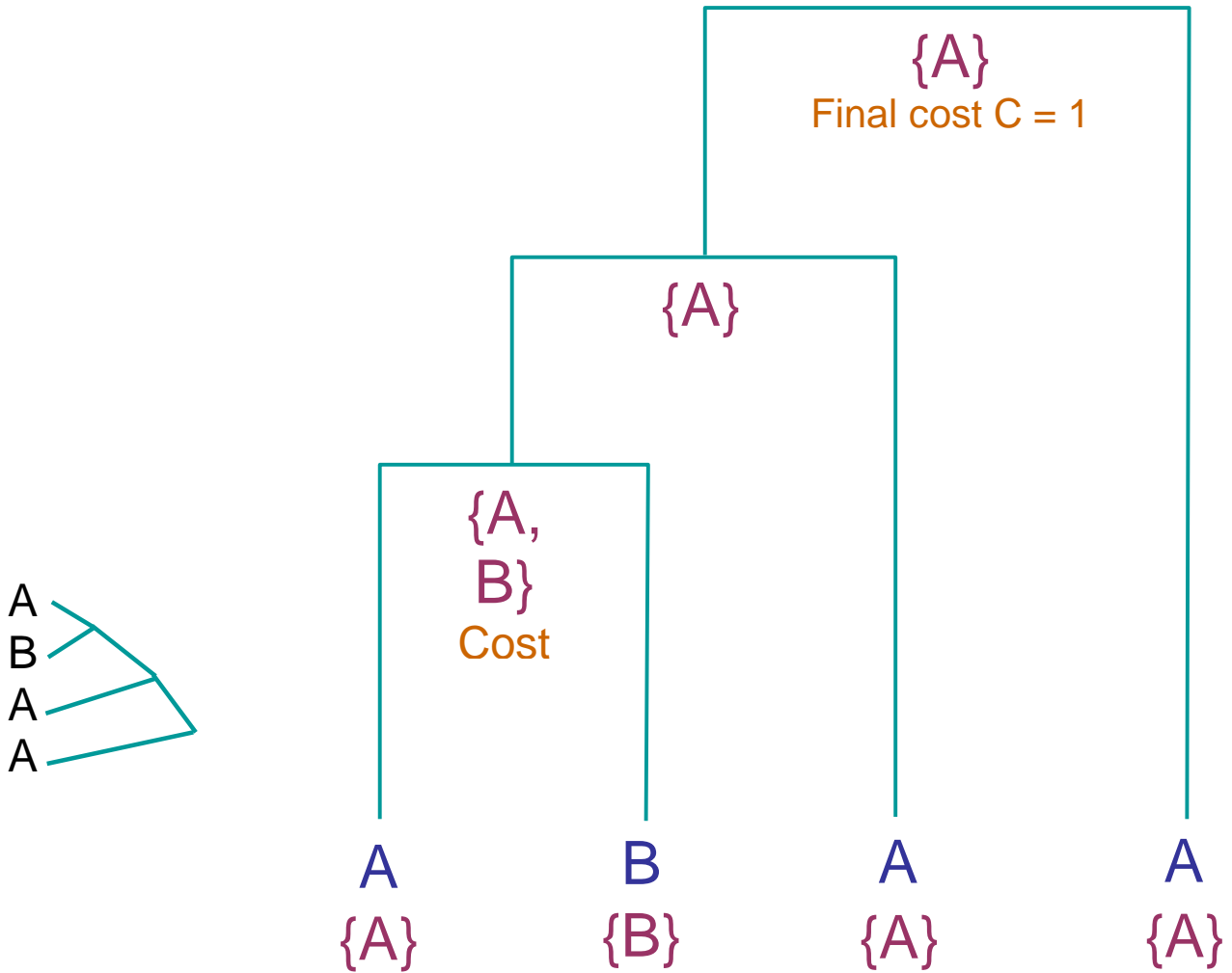
Goal: Given a multiple alignment, find the correct phylogenetic tree and the history of substitutions that led to the current species.

There are two subproblems that must be solved:

1. Given a tree and a multiple alignment, find the history of substitutions. This is an easy problem because we can look at each column in the multiple alignment independently, and calculate the minimum number of mutations that could have led to the given column.

2. Given a multiple alignment where the tree is not known, determine the tree that would give us the minimum number of mutations once we apply parsimony on that tree. This problem is **NP-hard** so we must use heuristics to search the huge space of trees intelligently.

In this example, we have a column consisting of ABAA and the tree. By inspection, we see that the minimum number of mutations that could explain this column is 1, namely a mutation from A to B in the most recent branch.



We calculate the score of the parsimony tree from the bottom upwards. Starting at the leaves, we create the sets of valid characters at each node. At a leaf, this set contains exactly one element: the character found in the column at that leaf. Walking up the tree, we form sets at each node by comparing the two sets of the children of that node. If the child sets overlap ($C_L \cap C_R \neq \emptyset$) then the set for a node is the intersection of its two child sets. This makes sense because only characters in the intersection could be passed on to both children without sustaining a mutation. If ($C_L \cap C_R = \emptyset$), then the only explanation is that a mutation occurred at this point, so we place ($C_L \cup C_R$) into the set and add one to the cost. The reason that we place the union of the characters is that we do not know at this point if the mutation occurred in the left branch or in the right branch, but in either case the cost is the same: 1 mutation. The parent could be either some character from the left child or some character from the right child. This process is repeated all the way up the tree until all nodes have been labelled and the final score has been calculated. This score is the minimum number of mutations that could explain this tree structure.

In formal terms, the algorithm is:

Given a tree, and an alignment column u

Label internal nodes to minimize the number of required substitutions

Initialization:

Set cost $C = 0$; node $k = 2N - 1$ (last leaf)

Iteration:

If k is a leaf, set $R_k = \{ x^k[u] \}$ // R_k is simply the character of k^{th} species

If k is not a leaf,

Let i, j be the daughter nodes;

Set $R_k = R_i \cap R_j$ if intersection is nonempty

Set $R_k = R_i \cup R_j$, and $C += 1$, if intersection is empty

Termination:

Minimal cost of tree for column u , = C

We also have an algorithm for tracing back to find the nucleotides in each ancestral node:

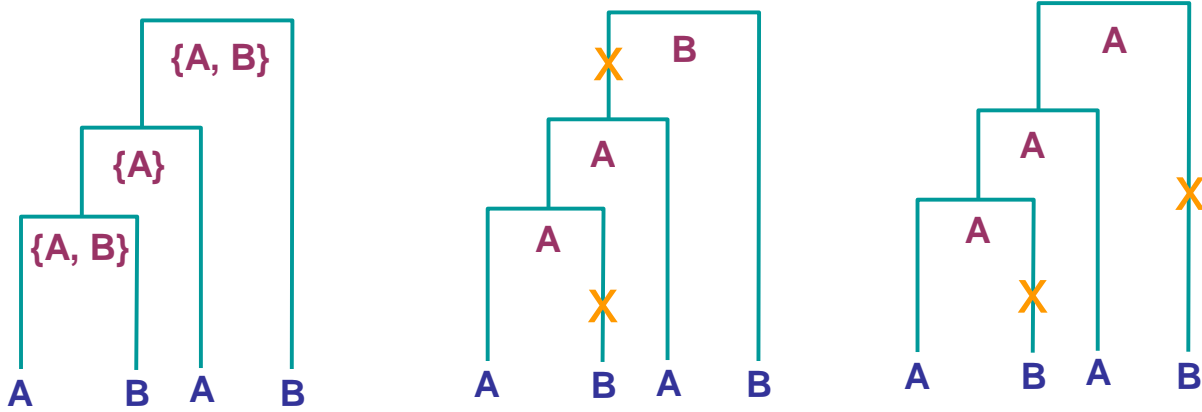
Traceback:

1. Choose an arbitrary nucleotide from R_{2N-1} for the root

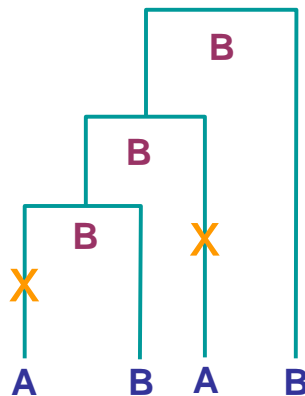
2. Having chosen nucleotide r for parent k , if $r \in R_i$ then choose r for daughter i .

Else, choose an arbitrary nucleotide for R_i .

This traceback procedure will **not** construct all possible parsimony scenarios!

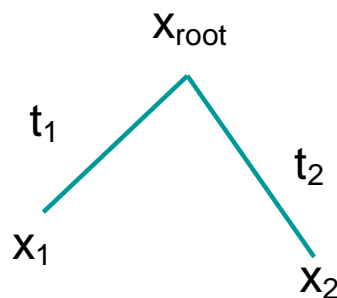


Note that we make arbitrary choices in the traceback procedure whenever we choose one out of a set of multiple characters. Thus, the simple traceback procedure can lead to many different scenarios. The tree on the left could give rise to either of the two on the right, depending on whether we initially choose A or B for the top nucleotide. However, there is a third possible scenario that traceback will never predict:



There are many algorithms for reconstructing the optimal trace but we will not be discussing them in this class. We will see some references to the literature later.

3. Probabilistic Methods:



The probability of a tree can also be calculated according to a model such as Jukes-Cantor or Kimura:

$$P(x_1, x_2, x_{\text{root}} | t_1, t_2) = P(x_{\text{root}}) P(x_1 | t_1, x_{\text{root}}) P(x_2 | t_2, x_{\text{root}})$$

This formula consists of three events which can be thought of as independent:

$P(x_{\text{root}})$ is the frequency of x_{root} in the ancestor

$P(x_1 | t_1, x_{\text{root}})$ is the probability of mutating to x_1 in t_1 .

$P(x_2 | t_2, x_{\text{root}})$ is the probability of mutating to x_2 in t_2

If we use Jukes-Cantor and assign letter values to the variables, we can plug in values and predict the scores. For example, if $x_1 = x_{\text{root}} = A$, $x_2 = C$, and $t_1 = t_2 = 1$, then the probability according to Jukes-Cantor is:

$$p_A * \frac{1}{4}(1 + 3e^{-4\alpha}) * \frac{1}{4}(1 - e^{-4\alpha}) = (\frac{1}{4})^3(1 + 3e^{-4\alpha})(1 - e^{-4\alpha})$$

Jukes-Cantor only has one model parameter (α), so maximum likelihood calculations for the branch lengths is very fast!

If we know all the characters of both the leaves and the ancestors, then we can calculate the combined probability of the tree:

Given the tree topology T and the branch lengths t ,

$$P(x_1, x_2, \dots, x_N, x_{N+1}, \dots, x_{2N-1} | T, t) = P(x_{\text{root}}) * \prod_{j \neq \text{root}} P(x_j | x_{\text{parent}(j)}, t_j, \text{parent}(j))$$

However, in general we only know the characters of the leaves, not the ancestors. (We know $x_1 \dots x_N$ but not $x_{N+1} \dots x_{2N-1}$.) Thus we must sum over every possible letter in $x_{N+1}, x_{N+2}, \dots, x_{2N-1}$:

$$P(x_1, x_2, \dots, x_N | T, t) = \sum_{x_{N+1}} \sum_{x_{N+2}} \dots \sum_{x_{2N-1}} P(x_1, x_2, \dots, x_{2N-1} | T, t)$$

There are an exponential number of combinations, but fortunately we can use dynamic programming to calculate the probability in polynomial time. This algorithm is called **Felsenstein's Likelihood Algorithm**:

Let $P(L_k | a)$ denote the probability of all the leaves below node k , given that the residue at k is a .

Initialization:

Set $k = 2N - 1$

Iteration: Compute $P(L_k | a)$ for all $a \in \Sigma$

If k is a leaf node:

$$\text{Set } P(L_k | a) = \mathbf{1}(a = x_k)$$

If k is not a leaf node:

1. Compute $P(L_i | b)$, $P(L_j | b)$ for all b , for daughter nodes i, j

2. Set $P(L_k | a) = \sum_{b,c} P(b | a, t_i) P(L_i | b) P(c | a, t_j) P(L_j | c)$

Termination:

$$\text{Likelihood at this column} = P(x_1, x_2, \dots, x_N | T, t) = \sum_a P(L_{2N-1} | a) P(a)$$

This algorithm gives us the probability of a given column, assuming that we have a model for how nucleotides mutate such as Jukes-Cantor. The parsimony algorithm gives us the minimum number of mutations in a given column, but Felsenstein's algorithm takes into account the possibility that there could be mutations of a letter to another letter and then back to itself, within a single branch. Thus, the expected number of mutations given by Felsenstein's algorithm will be larger than the number of mutations given by the parsimony algorithm.

4. Probabilistic Methods for Constructing Phylogenetic Trees

Problem: Given a multiple alignment (without gaps) of M sequences and a model for predicting mutations such as Jukes-Cantor, define the likelihood of the tree as the product of the likelihood of each column of the alignment:

$$L(\mathbf{T}, \mathbf{t}) = P(\text{Data} \mid \mathbf{T}, \mathbf{t}) = \prod_{m=1 \dots M} P(x_{1m}, \dots, x_{nm}, \mathbf{T}, \mathbf{t})$$

(All columns are assumed to be independent).

Find the tree topology \mathbf{T} and vector of branch lengths \mathbf{t} that maximizes the likelihood $L(\mathbf{T}, \mathbf{t})$.

There are many different algorithms and probabilistic approaches suggested to solve this problem, but at the moment parsimony techniques tend to be more successful. There are literally hundreds of different programs available for constructing phylogenetic trees. Here are some recommended programs for trying this yourself

§Rec-1-DCM3

<http://www.cs.utexas.edu/users/tandy/mp.html>

Tandy Warnow and colleagues

§SEMPHY

<http://www.cs.huji.ac.il/labs/compbio/semphy/>

Nir Friedman and colleagues

5. Multiple Sequence Alignments

There are two main ways that proteins evolve:

1. Species divergence -- when two species split, proteins in each species are free to evolve differently. We expect to see some similarity between proteins in the current species and the protein in the ancestor.

2. Protein duplication -- A large chunk of a genome is duplicated, resulting in two copies. One copy is free to evolve (randomly), and the organism can still survive because the other copy remains functional. The copy that evolves has a huge advantage over random DNA strings because it is already a functional protein -- the mechanism for transcribing and translating and folding the gene is already in place.

Suppose there are two very similar genes that arose from gene duplication and we are wondering if they function in the same biological process or not? If the genes are extremely similar to one another, then can we infer that they are likely to function in the same process? If both genes are used for the same function, then the only explanation for why both genes are necessary in the genome is that each gene has a different condition on which it functions.

When creating protein phylogenetic trees, it is important to know which branches arose from speciation and which branches were caused by gene duplication.

Multiple alignments are useful because they reveal to us the common evolutionary history of a set of organisms.

Definition of a multiple alignment:

Given N sequences x^1, x^2, \dots, x^N , insert gaps as needed in each sequence so that each sequence is the same length and the score of the global map between the sequences is maximized.

Multiple alignments are much more powerful than pairwise alignments. A faint similarity between two sequences may not be statistically significant, but if we see the same faint similarity between many sequences this is evidence that the sequences are indeed related.

Patterns of sequence variability can be revealing in their own right. For example, genes tend to show more variability at every 3rd position because the 3rd letter in each codon is often allowed to vary without changing the amino acid.

Scoring function:

The most common scoring function is the sum of pairwise scores:

Given a multiple alignment between X , Y , and C , the induced pairwise alignment between X and Y is the pairwise alignment between X and Y where we remove any column that contains both a gap in X and a gap in Y . A gap that is common to X and Y does not contribute to the pairwise score of X and Y .

The Sum of Pairs score for a multiple alignment is simply the sum over all pairs X, Y of the induced pairwise alignment between X and Y:

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

The scores for induced pairwise alignments can be calculated according to edit distance, affine gap penalties, etc. Each pairwise alignment has its own weight, because the multiple alignment may contain more sequences from one part of the tree than from another.

Profile Representation of Multiple Alignment

Normally, the size of a multiple alignment scales linearly with the number of sequences aligned. However, suppose we are aligning thousands of sequences and we want to represent the alignment compactly. Instead of storing each sequence separately, we summarize each column by storing the percentage of A's, G's, C's, T's or gaps in that column. Thus, we could represent a multiple alignment as 5 rows:

A		1			1			.8		
C	.6			1		.4	1	.6	.2	
G		1	.2					.2		.4
T	.2				1	.6				.2
-	.2		.8					.4	.8	.4

Where the sum of the columns in each row is 1. Alternatively, we can provide more detail about gap types, distinguishing between gap initiations, gap continuations, gap closes, and single gap positions. This would allow more information to be represented while keeping the space requirements for the multiple alignment small.

6. Algorithms for finding Multiple Alignments

Can we extend Needleman-Wunsch for solving pairwise alignments to find the optimal multiple alignment for N sequences?

Let $F(i_1, i_2, \dots, i_N)$ be the maximum score of aligning the first i_1 characters of sequence 1, the first i_2 characters of sequence 2, and the first i_N characters of sequence N.

Then $F(i_1, i_2, \dots, i_N)$ will be the maximum over all neighbors in this N-dimensional space of $(F(\text{neighbor}) + S(\text{neighbor}))$.

What is the running time of this algorithm?

Example: Suppose we are aligning 3 sequences.

$$F(i,j,k) = \max: \begin{aligned} & F(i-1, j-1, k-1) + S(x_i, x_j, x_k), \\ & F(i-1, j-1, k) + S(x_i, x_j, -), \\ & F(i-1, j, k-1) + S(x_i, -, x_k), \\ & F(i-1, j, k) + S(x_i, -, -), \\ & F(i, j-1, k-1) + S(-, x_j, x_k), \end{aligned}$$

$$F(i, j, k) + S(-, x_j, -),$$

$$F(i, j, k-1) + S(-, -, x_k),$$

At each step, there are 2^{N-1} neighbors that we must consider. There are L^N squares in our matrix, so in general the running time of this algorithm is $O(2^N L^N)$.

Note: if we want to do scoring for affine gaps, then we need 2^{N-1} gap states, so the running time would be 2^N .

Summary: multiple alignment is a HARD problem.(NP-hard)

Progressive Alignment

If the evolutionary tree is known, then we can perform pairwise alignments of sequences that are neighbors in the tree. In each step, align sequences x, y or profiles p_x, p_y to generate a new alignment with associated profile p_r . At each step of the algorithm, we perform a pairwise alignment between sequences or profiles. Profiles are just like sequences except that they have weighted frequencies for each letter at each position. We can treat

Example: For profile of the form (A, C, G, T, -):

Suppose $P_x = (0.8, 0.2, 0, 0, 0)$ and P_y is $(0.6, 0, 0, 0, 0.4)$

Then we can compute the sum of pairs score for this alignment as:

$$s(p_x, p_y) = 0.8*0.6*s(A,A) + 0.2*0.6*s(C,A) + 0.8*0.4*s(A, -) + (0.2*0.4)*s(C, -)$$

And the resulting profile p_{xy} is: the (weighted) average of the two profiles = $(0.7, 0.1, 0, 0, 0.2)$

We can also align calculate the score $s(p_x, -)$ for aligning a profile with a gap:

$$s(p_x, -) = 0.8*1.0*s(A, -) + 0.2*1.0*s(C, -) \text{ and } p_{x-} = (0.4, 0.1, 0, 0, 0.5)$$

Note that progressive alignment is not guaranteed to give us the optimal alignment.

However, the running time of progressive alignment is only $O(L^2*N)$ because there are $N-1$ pairwise alignment steps.