Part 2: Probabilities, Stochastic Models, and Evolutionary Trees

In this unit, I discuss probabilistic treatment of string data, including Hidden Markov Models and Molecular Evolution models, leading to algorithms for estimating evolutionary trees (also called phylogenies). Before reading these sections, I encourage you to read the supplemental primers on probability and stochastic processes.

Unit 7: Hidden Markov Model (HMM) of strings

In Part 1, we discussed the Finite State Automata (FSA) as a machine that accepts (= parses) certain families of strings that belongs to a *language* with *regular grammar* (remember the definition of a formal language and grammars). We can think of each FSA as a model of a family of strings. It is a pretty stringent model since every string either belongs to the family or does not belong to the family without error. On the other hand, in Chapter 7, we discussed the sequence alignment problem and how the main goal is to recover the positional homology relationships generated by a branching evolutionary process. We can also consider each branching process, mutational mechanisms, and the starting ancestral sequence as a model of a string family. If we give the evolutionary model a formal probabilistic framework, for each string, ***s***, we can talk about the probability that ***s*** belongs to some family **F**. In Unit 2, we will more explicitly discuss stochastic evolutionary models of strings. A FSA model of strings specifies the family in terms of the linear pattern of letters in the string; i.e., how the particular letters follow each other. The FSA and other formal grammar models tries to describe the regularities in the pattern of letters by rules that govern how the letters may appear in a string. Standard evolutionary models is not so much concerned about regularity of letters along a string but the regularity of letter variations amongst homologous positions; i.e., how homologous positions are expected to vary as a function of vertical transmission of genetic information. There are important utility to both kinds of models. In this chapter we will discuss the grammar type of models where the goal is to capture patterns of regularity along a string. Our model assumption is that a biological string (e.g., DNA) with some pattern along the sequence is likely to be a member of a particular biological function family. As mentioned, a strict grammar-based model like FSA is too brittle for biological modeling. Biological objects differ from idealized forms because of many reasons, not the least of which is evolutionary variations. Therefore, we would like to relax the grammar-based model a bit by stating membership in a model family using probabilistic terms rather than binary, IN/OUT, terms. A very useful model framework for this purpose is called the Hidden Markov Models (HMMs). Strictly speaking, a HMM is a type of probability model belonging to the study of stochastic processes of the Markov model type. Please read the primer on stochastic processes where I briefly discuss Markov models. I will also revisit stochastic processes below

The key idea for a HMM is that it describes multiple alternative random processes. For example, suppose that we go to a casino and play craps with two dice. It also turns out that the casino has ten sets of dice numbered from 1 to 10, each with imperfections that give a different probability to the outcome of a throw. The choice of different dice sets determines which of the alternative random processes are in play. Given a dice set, the outcomes of a throw, i.e., the pairs (1, 1), (1, 2),…(5, 6), (6, 6) are the "states emitted by dice set X." Each dice set has a characteristic **emission probability.** Also part of our model is that they all emit from a common set of possible states (e.g., pairs of numbers). Now it turns out that outward appearance of the dice sets is identical so the labels 1 to 10, indicating the dice set, are hidden from us. Only the casino operator knows which one is in play. The identity of the hidden state (the dice set label) determines which of the ten different emission probabilities govern the observed outcome. The key part of the model is that the identity of the hidden state changes as part of a hidden process and we only see the emitted states. One possibility is that the hidden state is changed deterministically. The casino operator might start with dice set 1, throw the dice, switch to dice set 2, throw the dice, etc., and repeat when dice set 10 has been thrown. Another possibility is to choose one of the dice sets from a random distribution. For example, the operator might choose one of the ten sets with equal probability and throw the dice. The Markov part of the Hidden Markov Model, is that we are going to use a particular probabilistic rule to change the hidden state. Without going into technical details, the essence of a Markov stochastic process is to specify a probability model of transitions between states. For each state, there is a transition probability of making a switch to any of the other states. For the 10 sets of dice, we might have a 10 by 10 matrix, where p_{ij} element denotes the probability of switching from the jth state to the ith state. The hidden states starts in some state, say the 5th dice set, then uses the transition probabilities to sample the next state.

To restate, a HMM has two state spaces, a **hidden state space** and an **observed state space**. Each hidden state has associated with it a probability distribution over the observed state space. This probability distribution is the emission probability distribution of the observations. There is a Markov process defined on the hidden state space specified by a set of transition probabilities. Typically, in biological models both the hidden state space and observed state space are discrete and finite.

Definition: An HMM is given by a 4-tuple (***S, O, T, E***) of hidden states ***S***, observed states ***O***, hidden state transition probabilities, ***T***, and emission probabilities ***E***

Example 1: Suppose we have two different dice. One is a loaded dice and the other is a fair dice. We will construct a hidden state space, {F, L}, to stand for the fair dice and loaded dice respectively. A throw of the fair dice results in numbers from 1 to 6 with equal probability. A throw of the loaded dice results in only the number 6. So the observed state space is {1, 2, 3, 4, 5, 6} and the emission probabilities for each of the hidden states are:

	1	2	3	4	5	6
F	1/6	1/6	1/6	1/6	1/6	1/6
L	0	0	0	0	0	1

Now we need to set up the Markov process for the hidden state. The simplest one might be that at each trial we pick up either the fair or loaded dice with equal probability. That is, the transition probability matrix looks like:

	F	L
F	½	½
L	½	½

A run of this process might look like this, where in the first line I number the throws, in the second line I show the "hidden" state sequence and in the bottom line I show the "observed" state sequence.

1	2	3	4	5	6	7	8	9	10	11	12
F	F	L	F	L	L	F	L	F	F	F	L
1	1	6	3	6	6	2	6	6	4	2	6

As in the usual stochastic processes, we want to compute things like the probability distribution of the observed states at time *t* (or in the above case the *t*-th throw). Let's consider this for, say the 1st, throw. Since our transition matrix implies that at any throw, we pick up the fair or loaded dice with equal probability we have the following possibilities for obtaining the number 1:

Prob(1) = Prob(pick up the fair dice AND throw 1) = ½ x 1/6 = 1/12

This is because if we picked up the loaded dice, we will never get the outcome 1. For number 6 we have the following possibilities:

Prob (6) = Prob([pickup the fair dice AND throw 6] OR [pickup the loaded dice and throw 6]) = ½ x 1/6 + ½ x 1 = 7/12.

Since all the other numbers have the same reasoning as the number 1, we have the distribution: P(1) = 1/12, P(2) = 1/12, P(3) = 1/12, P(4) = 1/12, P(5) = 1/12, and P(6) = 7/12.

A HMM model is, in effect, a mixture model where sometimes we draw from one set of emission probabilities, sometimes we draw from another set---all dependent on which of the hidden states are in play. As a model, we are usually interested in the hidden states and not the observed states. We think of the observed states as giving us information about the hidden states. For example, we might be interested in filing a lawsuit against the casino for playing with a loaded dice and therefore we want to know which of the dice tosses were fair and which were loaded. In a more biological context, the hidden state might be structural properties of a protein (e.g., alpha helix), which then governs the probability of which amino-acids might be "emitted" at which position. Since we are interested in the hidden states, the Markov process part of the model is important in determining the properties of the hidden states. For example, consider the waiting time until the dice switches from Fair to Loaded and vice versa for the above model. Suppose that we have just moved from F to L. What is the probability distribution for the number of tosses until we go back to F? Denoting the random variable for this waiting time as W, a little bit of thought will show that it is:

P(W=1) = ½
P(W=2) = ¼
P(W=3) = 1/8

and more generally $P(W = k) = (1/2)^k$.

We consider the more general case where the transition matrix looks like this:

	F	L
F	θ	1- θ
L	1- θ	θ

Here, θ is the probability of staying at the same state and 1- θ is the probability of changing, then the waiting time until a state switch is $P(W=k)=\theta^{k-1}(1-\theta)$. That is, the probability of staying in the same state for *k- 1* steps and then switching at the *k*th step. Note that this means that the probability of staying in any one state is a *geometrically decreasing function* of steps. So, strictly speaking a HMM model is appropriate when we expect runs of each hidden state to have a geometric distribution. However, we often use the model in contexts where this is not strictly what we expect.

HMM as a model of sequence family

To get an idea of HMM as a model of a sequence family, we will start with a simple but an exhaustive example going back to FSA. Suppose that we wanted to consider as possible input only 5 letter strings of DNA. So we have 4^5 = 1024 possibilities. We can enumerate them starting with "`AAAAA`" and ending with "`TTTTT`." Suppose what we want to do is to model the biological functional family called "promoter". We do exhaustive experiment testing of each of the 1024 strings in the possible role of a promoter and we get an outcome. If we are lucky, it may be that the outcome can be compactly described by a regular grammar like "all string with the subsequence `T.TA[A|T]`". We learned how to construct a FSA for parsing such strings. If we are not lucky, the set of positive promoters may be highly irregular so we may have to construct a table of 1024 rows for each of the possible 5-letter strings and then assign a Boolean value to each row denoting whether a given 5-mer is a promoter or not. Regardless, once the experiment has been done we will have a tool where we can feed it any string of length five and get a return value 0 or 1 indicating whether it is or is not a promoter.

Now suppose we did the experiments but each trial was not so conclusive. We might have tested the string "`ATATA`" ten times and it functioned as a promoter in only 5 cases. Or, maybe we did not have enough resources to do exhaustive experiments so we only did 100 experiments and from this we have to extrapolate to the entire set of 1024 strings. In both cases, it is not reasonable to make a "is or is not a promoter" statement, but rather "is a promoter with probability X"—either because the experimental outcomes were variable or because we only had partial information so there is inherent uncertainty to our extrapolation. In this case, instead of 0/1, it would make sense to give a number between 0 and 1 to each 5-mer. Let p_i denote the number we gave to the ith 5-mer, then we can think of this as "pick a promoter at random" then p_i is the probability the promoter's string is the ith 5-mer. While it is better to state these models in empirical terms of "pick a promoter at random", in the following we will be a bit more model-theoretical and say, "the promoter *generates* ith 5-mer with probability p_i."

We want a model of the promoter that gives a probability value for each possible string. One possible model is to enumerate 1024 numbers between 0.0 and 1.0 such that they sum to one (so that the total probability of all possible strings is one). We are allowed to put arbitrary numbers into this table—we will call such a model the "***maximal complexity model.***" It is maximally complex because we need to know as many numbers as there are possible strings. Our model will be able to account for all kinds of unexpected irregular patterns. For example, suppose we find that "`AAAAA`" and "`AATAA`" and "`AACAA`" are promoters but "`AAGAA`" is not a promoter. It would seem strange that a single base change of a particular kind would break this promoter—but with our maximally complex model it doesn't matter since we don't a priori expect any regularities. We will need to do exhaustive experiments for each of the 1024 strings to estimate those probability values.

On the other end, a completely naïve model might say all 1024 strings have equal probability of being a promoter. Since this is of no use, an alternative ***minimal complexity model*** might consist of setting probability values for each of the four nucleotides, say P(A) = 0.7, P(C) = 0.1, P(G) = 0.1, and P(T) = 0.1, and assuming that the probability of any five-letter string can be obtained by drawing five letters independently from this distribution. This model would tell us that the probability of a string "`AAAAA`" generated by the promoter family is $0.7^5 = 0.17$. This model only requires four numbers to describe the promoter family (actually three numbers since the probabilities sum to one). From a biological point of view, at best, this simple model expresses the tendency of promoter sequences to contain high frequency of certain letters and not others—but, it says nothing about the ordering of letters in the string.

We can now move to a slightly more complex model by suggesting that the promoter family can be described by the following probabilities for the first starting letter: P(A) = 0.5, P(C) = 0.0, P(G) = 0.0, and P(T) = 0.5. And, then the next letters are generated by a Markov process with the following transition probability:

	A	C	G	T
A	0.7	0.05	0.05	0.2
C	0.1	0.7	0.1	0.1
G	0.1	0.1	0.7	0.1
T	0.2	0.05	0.05	0.7

Now we have 3 + 12 = 15 different probability numbers describing the promoter family. Note that this model can be applied to any length strings.

Another way to generate a more complex model is to say that at each sequence position we have a different probability of bases and the total probability of the string is a product of these numbers. For example, one might say the probability of each base at each position is given by the following table:

	A	C	G	T
pos 1	0.5	0.0	0.0	0.5
pos 2	0.8	0.05	0.05	0.1
pos 3	0.5	0.05	0.05	0.4
pos 4	0.0	0.05	0.05	0.9
pos 5	0.5	0.0	0.0	0.5

This kind of model requires 3 numbers for each position, so we have 3x5 = 15 for length five strings. A table like above has been used extensively in computational biology and has been variously called, "profile", "position-specific weight matrix (PWM)", "weighted motif", etc. There are many databases of compilations of such profiles for various functional sequence families.

At this point, we can generate more complex models in several ways. First, we might start with the simple Markov model above and extend it to a *k*th-order Markov model. A *k*th order Markov model gives transition probabilities based on not just the current state but current state plus *k-1* prior steps. So, if we were to specify a 2nd-order Markov model for our string, we would need to first give 16 numbers for the probability of the 16 different pairs at the 1st and 2nd position of the string and then give a transition matrix of the following type:

	A	C	G	T
AA	0.7	0.1	0.1	0.1
AC	0.3	0.6	0.05	0.05
AG	0.2	0.3	0.3	0.2
AT	0.8	.	.	.
CA	.	.	.	.
⋮	.	.	.	.

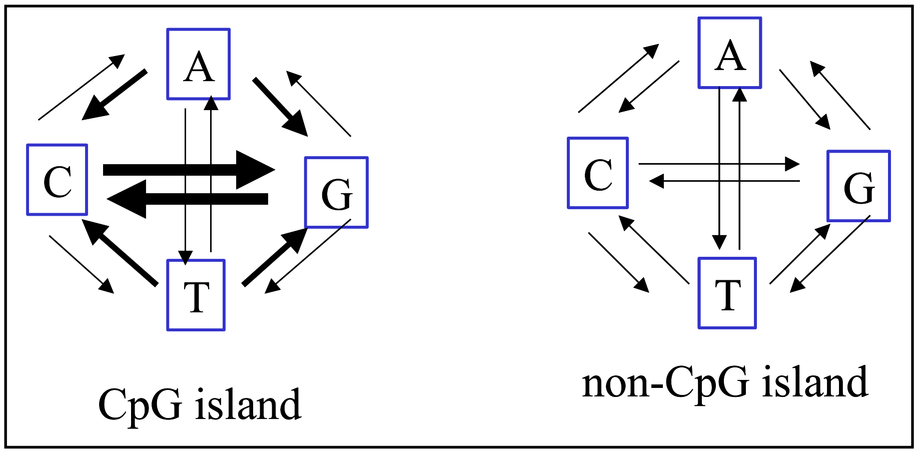
Another possibility is to give position specific Markov transition matrices. So, for the transition from the first position to the second position we might have one matrix and then for the second to third we might have a different matrix, and so on. Another possibility is to assume hidden states, say P for promoter and N for non-promoter. Then we would model one kind of probability of nucleotides for P state and another for N state.

As we can see, we can make very complex and very precise models for describing members of a sequence family. But, those models have a lot of parameters. We need the parameter values to compute, for example, the probability of a string "`AAAAA`" versus the probability of the string "`TATAT`", etc. Where would we get those numbers? In practice, we estimate those numbers using empirical observations (we will cover estimation as a general topic later). However, more complex the model, more data we need to estimate the parameters (i.e., the probability values). And, less likely a complex model will generalize to unseen data. What we want is a reasonable family of models, not too complex, not too simple—a Goldilocks model. HMMs have some nice properties that often fit this bill. We will start with a canonical example

Describing CpG islands with HMMs

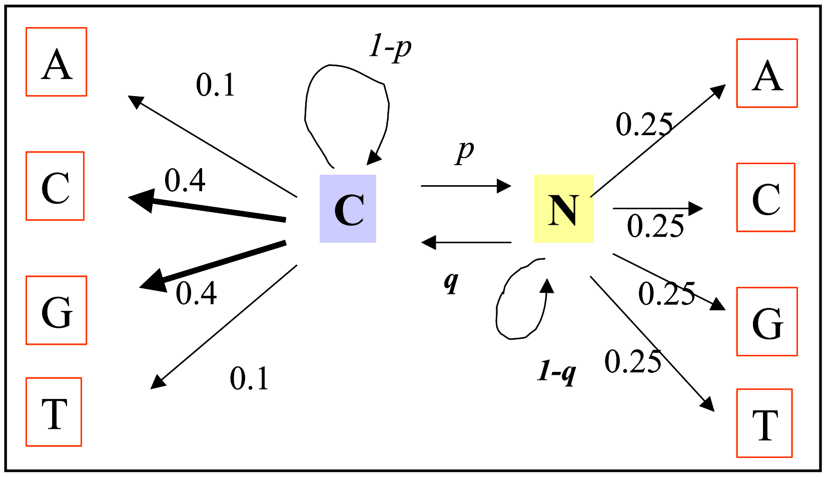
`CpG` islands are portions of the genome that have a preponderance of the dinucleotide sequence "`CG`" We use the notation "`CpG`" to indicate that the sequence of interest has a "`C`" immediately followed by a "`G`", rather than a preponderance of `C`'s and `G`'s in any order. The `CpG` islands in the genome are related to gene regulation and many promoters of genes are found within `CpG` islands.

Model 1: As a first step, we note that the main characteristic of a `CpG` island is this di-nucleotide co-occurrence. Thus, a first-order Markov model is a reasonable approach. We first build two Markov models, one describing the genomic string within a `CpG` island and another within a normal part of a genome. We depict our model with a state transition diagram like this:



In this figure, the thick lines are supposed to denote more probable type of transitions and the thin lines the less probable transitions. Under the non-CpG island model, all nucleotides are equality probable. But, under the CpG island model, there is a tendency to switch from C's to G's and from G's to C's, with the result that we will see a lot of CpG dinucleotides.

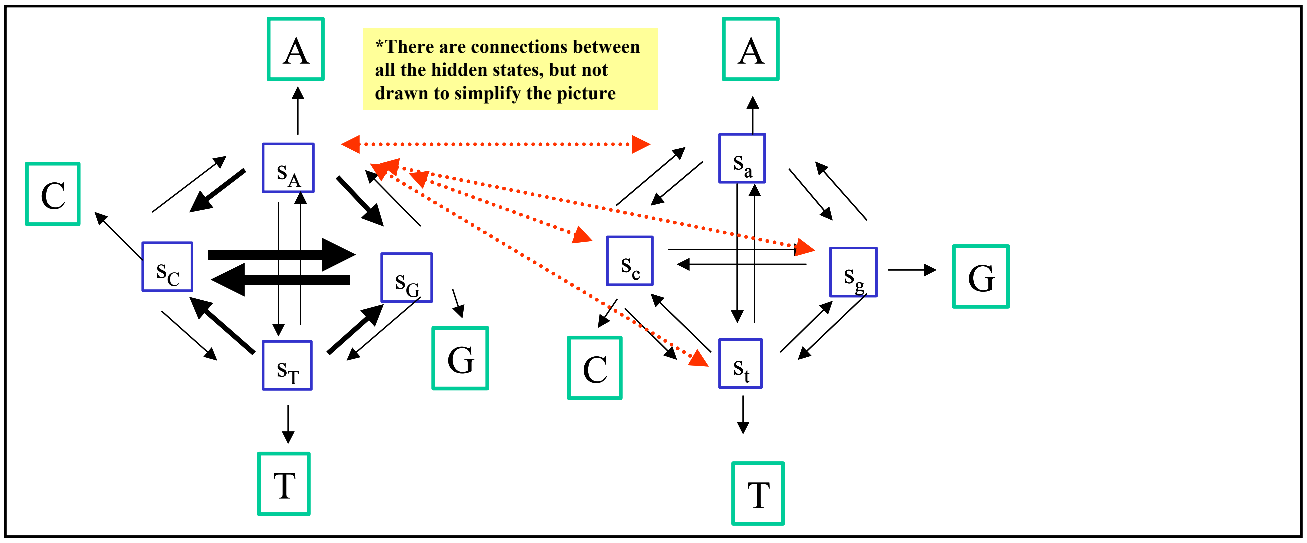
Model 2: The above model proposes two distinct models for the different genomic segments, but it does not handle the idea that a genome consists of an alteration between CpG island and non-CpG island portions. An easy way to utilize the HMM structure is to say that we have two hidden states: CpG islands (denoted C) and non-CpG island (denoted N). The genome alternates between these to hidden states. The following diagram shows a possible model:



The Markov process between the "C" and "N" states now accounts for genomic region interchanges. However, in this particular model, the observed nucleotides are emitted as independent states. Thus, we might be able to manipulate the emission probabilities of the "C" and "G" state to produce a lot of C's and G's, but we can't explicitly incorporate the dependence of C following G.

Model 3: The next model solves this problem by expanding the set of hidden states. Here we use a bit of a trick. We will use the Markov models of *Model 1* and couple them together. However, instead of the states being observed states, the states will be considered hidden. So we will have hidden states {S_A, S_C, S_G, S_T; Sa, Sc, Sg, St}, where the capital letters are the hidden states under CpG islands and the lower case letters are the hidden states under non-CpG islands. Each of the hidden states will be assumed to emit one kind of nucleotide with probability 1 and no others. The state transition diagram looks like this:

Why didn't we just take the diagram from *Model 1* and simply connect the observed states? The answer is simple, the observed states—are observed states: we can't have two different "`A`"s. The use of the hidden states allows us to model the scheme that there are in fact two kinds of "`A`"s, those in `CpG` island segments and those in non-`CpG` island segments. In this model, the hidden states are both objects of interest (because we want to know where the `CpG` islands are) and a modeling device to represent classes of observations that superficially look the same but


*There are connections between all the hidden states, but not drawn to simplify the picture
A
C
s_A
s_C
s_G
s_T
G
T
A
s_a
s_c
s_g
s_t
C
G
T

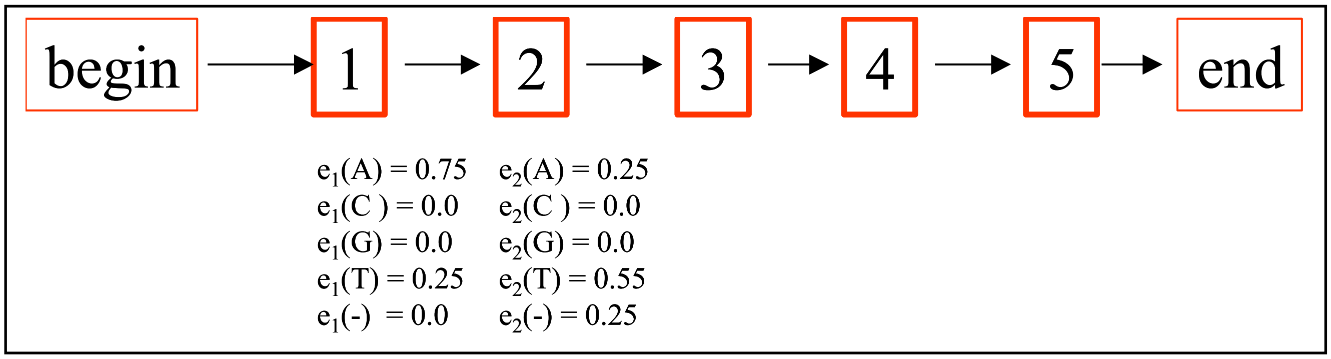
belong to different processes. A similar kind of construct can be used to model nucleotides that might show *k*-mer dependencies (i.e., probabilistic coupling in mutational changes across *k* letters) and belong to a different class dependent on the structure of the dependency.

Profile HMM

We now move onto a HMM that has been widely used in computational biology: profile HMMs. A profile is a model for sequences where we specify the model by giving position specific probability of each nucleotide—and sometimes "-" character to indicate deletions. As discussed above, the main assumption is that the frequency of each nucleotide at each position is independent of that at any other position. First, we remember that profiles looked something like this:

	1	2	3	4	5
A	0.75	0.25	0.25	0.00	1.00
C	0.00	0.00	0.00	0.25	0.00
G	0.00	0.00	0.25	0.00	0.00
T	0.25	0.50	0.50	0.50	0.00
-	0.00	0.25	0.00	0.25	0.00

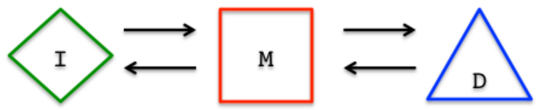
Here we also allowed for "-" characters to account for insertions and deletions (we call this indels). So this profile says that in position 1 the probability of A is 0.75, T is 0.25, and zero for all other possibilities. We can always take profiles like this and make it into a HMM. Namely we just declare each of the positions to be a hidden state. Then each of the hidden states emits the nucleotides with the probability given by the profile. That is, we have the following state transition diagram.



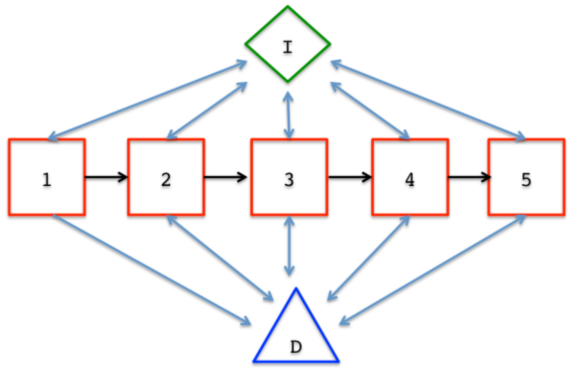
The states “1”, “2”, and so on correspond to positions in the profile table and the transitions between the states is automatic (i.e., with probability 1). At this point, the model will have a deterministic transition between the five different hidden states. So why even bother? The main thing we want to do with introducing hidden states is to include the possibility of insertions and deletions. The profile above already has indels. But, instead of modeling indels as “state” at each position, we would like to model indels as an event that may happen in a natural consecutive sequence. For example, going from left to right, a string might originally look like `ACGGCTCCTT`, then “experience” insertions and deletions such that it now looks like “`ACCTACCCT`”. We can diagram the relationship between the original string and the modified string like this:

AC--CTACCCCT-
MMDDMMIIIMMMD

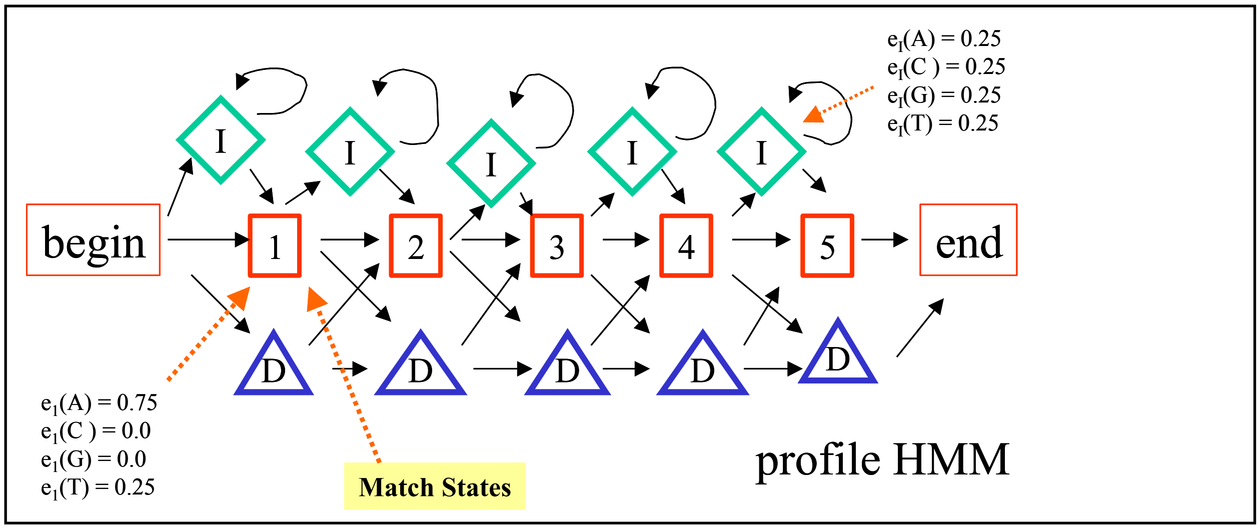
Below the string, I annotated the type of letters as `M` for Match, `D` for Deletion, and `I` for Insertion. Here, by Match, I mean that the letter matches the original string. So in the modified string there were two letters that were deleted in the first event, three letters that were inserted in the second event, and one letter that was deleted in the third event. The first key in the model is the realization that the sequence of annotation below the string can be seen as a sequence of hidden states. Each hidden state emits an observed state, either nucleotide letters or “-“. We assume that it is meaningless to consider a transition from `D` to `I` or `I` to `D`; that is, the string cannot directly make a transition between the indel states. The HMM model might look like this:



Unfortunately, there is a problem in the first match state, we want it to emit “`A`”—that was the original letter in the string. In the second match state, it should emit “`C`”, and so on. This means that the match state at each position is actually a separate state. So, we might introduce numbered match states, say 1, 2, 3,…. Now the HMM model looks like this:



This is almost good, but there is another problem. Say we went from state “1” to deletion state and then later went from state “5” to deletion state again. We want the return from each of the deletion state to be to the right match state. If we went from “5” to D to “1”, that wouldn’t do because it goes back to the beginning of the string model. This means we actually have to introduce a separate deletion state and insertion state for each match state. This is the full profile HMM model and it looks like this:



(Here to simplify the diagram I didn’t give separate subscripts for each of the I and D states; just assume that they are different hidden states.) The diagram looks a bit complicated but it is a straightforward extension of the previous HMMs. First, starting at the begin state we have possibility of making a transition to one of three hidden states. We can go to the hidden state “D”, which means to delete the nucleotide (i.e., do not emit any base). We can go to the “I” state which means to insert a random nucleotide. We typically assume that in the “I” state there is a characteristic genomic emission probability for nucleotides. In the diagram above, I just modeled this as equal probability for all nucleotides. In a real genome, one might model it from the overall base frequencies of the genome. Finally, we can go to state “1”, which is the match state 1. We can give that match state a characteristic emission probability of the nucleotides. In the above diagram we have five match states from 1 to 5 so those really determine the characteristics of this string family. The emission probability of each match state can be seen as equivalent to the profile model. All we did was add some possibility of indels.

Once we are in an insertion state, we can continue to insert nucleotides and this is shown by the loop arrow in the diagram. Or, we can then move to the next match state. When we are in the delete state, we can move to the next match state or continue to delete by moving on to the next deletion. There are no loop arrows for the delete state, because we interpret the delete states to mean: delete the base that would have been generated by the current match state.

HMM construction involves setting up the structure of the model—by which we mean the connectivities between the hidden states. And, determining the parameter values—by which we mean giving numerical values to the transition probabilities and the emission probabilities. Once we have the model, then we can examine some observed string and ask questions like:

"What is the probability that string S was generated by the HMM?"
"Given the observed string S, what is the most probable sequence of the hidden states?"
"Given the observed string S, what is the probability distribution of the hidden state at position i?"

Also, given a HMM structure, we might ask how we might use observed data to specify the parameters of the model—i.e, estimate the parameters. These are all problems of computing with HMM models. We will discuss those problems in the next chapter.

Before we end, let's put the HMM in a larger modeling context. First consider the match states. What do the emission probabilities mean? Also what do the match states 1 to 5 represent? One way to think about this is to consider an ancestral string, say with five letters. The match states 1 to 5 represents the nucleotide order of this ancestral string. Each state represents the homologous position site to this ancestral string. The emission probabilities can be seen as a model of evolutionary variability for that homologous position due to ancestral mutations, selection, drift, etc. That is, suppose we chose a member of evolutionary string family (i.e., strings that are related to each other by evolutionary descent) at random from life on Earth. Then the emission probability represents the probability of seeing various nucleotides in this random sample.

Alternatively we can construct a functional model for the match states that is not strictly an evolutionary model. Suppose each position helps to carry out a particular biochemical function. Then the states 1 to 5 might represent the individual nucleotide identity components of this biochemical function. Suppose we now randomly sample (hypothetically) all strings that carry out the same biochemical function, then the emission probability represents this sampling probability. Evolutionarily related strings tend to have similar biological function so these two models are a bit mixed.

To be clear, we already chose HMM as the basis for our models of strings. I am now talking about the conceptual model that supports the choice of HMM as the computational model. Which conceptual model we choose can have consequences for some specific aspects of the probability model. For example, if we assume the "functional model" then it is possible that we should not assume independence of emission probabilities. This is because the nucleotide combinations at different positions may have epistatic effect on function. If we assume the "evolutionary model", it is a bit more reasonable to have independent emission probabilities at each position—if we rule out selection, which might be affected by epistatic interaction of the nucleotides. On the other hand, with the evolutionary model if we have multiple strings, say 10 strings, the strings will be related to each other by evolutionary descent so we should not treat the fit to the HMM for each string independently.

The profile HMM model combines either the functional or the evolutionary model of position specific nucleotide probability distribution with a model of insertions and deletions. The real key is that we are making the assumption that insertions at any position or deletions starting at any position has a Markov model structure. That is, a geometric distribution of length of insertion or deletion. Seen this way, we can immediately see that this is one of many possible models for indels. In many ways, it is one of the simplest and its major advantage is that it has a particular model structure that makes it easy to compute probabilities. As always, we err on the conservative side with models and try to choose the simplest one that gives us a reasonable fit to the data.

Unit 8: HMM computations

In the following, I will discuss the algorithmic technique for computing the kinds of quantity we discussed in Unit 12. To see the entire structure of the computations without being confused by the algorithmic details we will work through a complete case of computation by hand, then introduce the algorithms that allow us to make the actual calculations more efficient. I will work with a simplified version of the Fair and Loaded dice example from the previous discussion. Here, we have Fair and Loaded coins as hidden states and that the observed states are Heads or Tails, which Ie will denote 1 and 0, respectively. The full model is:

Hidden States: {F,L}
Observed States: {0,1}
Emission Prob:

Table 1

	0	1
F	½	½
L	¾	¼

Transition Prob:

Table 2

	F	L
F	7/8	1/8
L	3/8	5/8

We will consider all possible outcomes for two throws, along with the respective probabilities, assuming that the initial probability of drawing F or L coin is ½. So we have the following possible combination of hidden state sequence and observed state sequence:

Table 3

FF:00 = ½ x ½ x 7/8 x ½	FL:00 = ½ x ½ x 1/8 x ¾	LF:00 = ½ x ¾ x 3/8 x ½	LL:00 = ½ x ¾ x 5/8 x ¾
FF:01 = ½ x ½ x 7/8 x ½	FL:01 = ½ x ½ x 1/8 x ¼	LF:01 = ½ x ¾ x 3/8 x ½	LL:01 = ½ x ¾ x 5/8 x ¼
FF:10 = ½ x ½ x 7/8 x ½	FL:10 = ½ x ½ x 1/8 x ¾	LF:10 = ½ x ¼ x 3/8 x ½	LL:10 = ½ x ¼ x 5/8 x ¾
FF:11 = ½ x ½ x 7/8 x ½	FL:11 = ½ x ½ x 1/8 x ¼	LF:11 = ½ x ¼ x 3/8 x ½	LL:11 = ½ x ¼ x 5/8 x ¼

The corresponding probabilities can be easily computed by following simple sequence. Consider the case of "FF" hidden state sequence emitting "00". First, we need to have the first "F" hidden state occur whose probability is ½ by our assumption of equal probability of each of the hidden state at the start. Given "F" hidden state, the probability of a "0" emission is ½ from the model Table 1 above. Given that the first hidden state is "F" the probability of making a transition to another "F" state is 7/8 from the transition Table 2. Given the second "F" state, probability of emitting another "0" is ½ . All of these events have to happen independently in sequence so the total probability is ½ x ½ x 7/8 x ½ . More formally this should be written as:

$$P(\text{"}FF\text{" } and \text{ "00"} \mid \Theta) = \frac{1}{2} x \frac{1}{2} x \frac{7}{8} x \frac{1}{2}$$

where Θ stands for HMM model including the parameter values. Since, all of our calculations assume that the model is given, we will drop the "given Θ" part from our notations. You can put it in everywhere for completeness if you want.

We can now use standard probability calculus to compute all the quantities of interest. For example,

$$P(\text{"00"}|\text{"}FF) = P(\text{"}FF\text{" } and \text{ "00"})/P(\text{"}FF\text{"}) = \left[\frac{1}{2}x\frac{1}{2}x\frac{7}{8}x\frac{1}{2}\right]/\left[\frac{1}{2}x\frac{7}{8}\right] = \frac{1}{4}$$

where P("FF") is obtained from just considering the probability of first "F" (= ½) followed by a transition to the second "F" (= 7/8).

Now we can compute the probability of the observation "00" given the model (again remember we dropped the "| Θ" notation, but you should remember that it should be everywhere).

$$P(\text{"00"}) = \sum_{Y\in F,L}\sum_{X\in F,L} P(\text{"00" } and\ XY) = 7/64 + 3/128 + 9/128 + 45/256 = 85/256$$

That is, the sum of the first row of the above table.

Given the observed states "00", we would like to know what were the hidden states. One possible approach to this problem is to ask what is the sequence of hidden states that give the highest probability given the observation. That is,

$$\max_{X,Y\in\{F,L\}} P(\text{"}XY\text{"}|\text{"00"}) = \max_{X,Y\in\{F,L\}} P(\text{"}XY\text{" } and \text{ "00"})/P(\text{"00"})$$

We can find this by examining the first row (with the "00" observed states) and finding the cell with the highest value. This is in fact the last cell with "LL" hidden states and

.

$$P(LL|00) = \frac{P(LL\ and\ 00)}{P(00)} = \frac{\frac{45}{256}}{\frac{97}{256}} = \frac{45}{97}$$

One might also want to find the marginal distribution of the first hidden state given the observed states. That is,

$$P(LX|00) = \frac{[P(LF\ and\ 00) + P(LL\ and\ 00)]}{P(00)} = \frac{\left(\frac{18}{256}+\frac{45}{256}\right)}{\left(\frac{97}{256}\right)} = \frac{63}{97}$$

$$P(FX|00) = \frac{[P(FF\ and\ 00) + P(FL\ and\ 00)]}{P(00)} = \frac{\left(\frac{28}{256}+\frac{6}{256}\right)}{\left(\frac{97}{256}\right)} = \frac{34}{97}$$

We can follow this up with similar computation for the second position

$$P(XL|00) = \frac{[P(FL\ and\ 00) + P(LL\ and\ 00)]}{P(00)} = \frac{\left(\frac{6}{256}+\frac{45}{256}\right)}{\left(\frac{97}{256}\right)} = \frac{51}{97}$$

$$P(XF|00) = \frac{[P(FF\ and\ 00) + P(LF\ and\ 00)]}{P(00)} = \frac{\left(\frac{28}{256}+\frac{18}{256}\right)}{\left(\frac{97}{256}\right)} = \frac{46}{97}$$

which due to some coincident parameter values, are exactly same as the first position. It is now interesting to try computing the product of the two cases of the "L" at the first position and the "L" at the second position. That is,

$$P(LX|00)P(XL|00) = \frac{63}{97}\frac{51}{97} = \frac{3213}{9409}$$

This value, 3213/9409 is somewhat smaller than the most probable hidden state path "LL" found above as 45/97. That is, we don't get the probability of "LL" hidden path by multiplying the marginal probability of "L" in the first position and "L" in the second position. This is as it should be. By the definition of the Markov process, the probability of the second L is conditional on the first L and the events are not independent of each other (unless the transition matrix has ½ everywhere).

We can see from the above computations that all the calculations are straightforward and only involve some arithmetic, products and sums, and keeping track of what to sum. The hard part is that with realistic length sequences the number of possibilities explode. For example, given the above F/L coin model, suppose we had a sequence of 100 throws. Then we would have 2^{100} possible hidden state sequences and 2^{100} possible observed outcomes for a total of 2^{100} x 2^{100} = 2^{200} = 10^{60}. Even the fastest computers cannot keep track of such combinatorial possibilities and do the appropriate sums.

One thing that slightly helps is that we usually want to do our computation conditional on a particular observed sequence. Thus, we don't need to consider all the possible observations but just one. For example, in the previous calculations, we did everything only with respect to the first row, i.e., the "00" outcome case. It doesn't help that much to go from 10^{60} to 10^{30}, but we can use it help prepare for the algorithms with another small example. We will use the above model and assume we have a sequence of three throws and condition on the outcome "010". So we have the following possibilities and corresponding probabilities:

Table 4

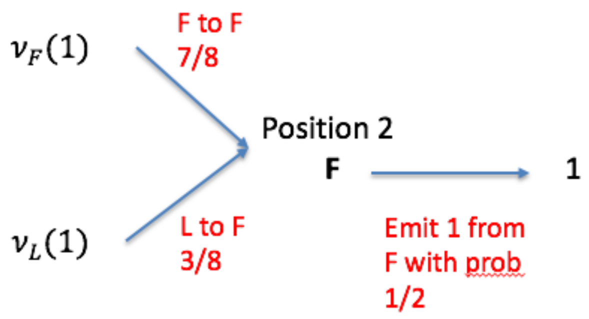
Hidden Path	FFF	FFL	FLF	FLL
Prob	$\frac{1}{2}\frac{1}{2}\frac{7}{8}\frac{1}{2}\frac{7}{8}\frac{1}{2}$	$\frac{1}{2}\frac{1}{2}\frac{7}{8}\frac{1}{2}\frac{1}{8}\frac{3}{4}$	$\frac{1}{2}\frac{1}{2}\frac{1}{8}\frac{1}{4}\frac{3}{8}\frac{1}{2}$	$\frac{1}{2}\frac{1}{2}\frac{1}{8}\frac{1}{4}\frac{5}{8}\frac{3}{4}$
Hidden Path	LFF	LFL	LLF	LLL
Prob	$\frac{1}{2}\frac{3}{4}\frac{3}{8}\frac{1}{2}\frac{7}{8}\frac{1}{2}$	$\frac{1}{2}\frac{3}{4}\frac{3}{8}\frac{1}{2}\frac{1}{8}\frac{3}{4}$	$\frac{1}{2}\frac{3}{4}\frac{5}{8}\frac{1}{4}\frac{3}{8}\frac{1}{2}$	$\frac{1}{2}\frac{3}{4}\frac{5}{8}\frac{1}{4}\frac{5}{8}\frac{3}{4}$

These computations are exactly the same as previously with the conditions that we are emitting the sequence "010" for each of the possible hidden path. We now would like to find a way to find the most probable hidden path without having to produce this entire table. Our strategy is to consider the most probable hidden state and the observed state sequence up to ith position and continue to build on that to the (i+1) position and so on.

We will keep track of the following quantity $v_k(i)$ to mean the "probability of the sequence up to ith position, up on taking the most probable path and where the hidden state ended at kth state." That is, what we are going to do is assume that we have sequentially chosen the most probable hidden state path up to the i-1 position and now $v_k(i)$ is the probability when we extend that up to the ith position and kth hidden state at that position. For this example the hidden states are F and L, so we will be keeping track of the two quantities $v_F(i)$ and $v_L(i)$.

We first start at the begin state, which we index as B, and 0th position so we would have $v_B(0) = 1$ because at this point B is the only possible hidden state. Now we need to compute $v_F(1)$ and $v_L(1)$. The quantity $v_F(1)$ represents having hidden state F at the first position and the probability is ½ x ½ ; i.e., the probability of going from B to F (=1/2) and emitting 0 with the hidden state F (= ½). The quantity $v_L(1)$ represents having hidden state L at position 1 and the corresponding probability is ½ x ¾ ; i.e., the probability of going from B to L (=1/2) and emitting 0 from hidden state L (= ¾).

In the next step we compute $v_F(2)$ and $v_L(2)$. The quantity $v_F(2)$ tracks the probability of the most probable path up to position 2, that ends up with the hidden state F. We could have ended up with F from F in the first position or L in the first position; that is, the hidden sequence could be "FF" or "LF". We compute both of these possibilities and see which gives the higher probability. Recall that $v_F(1)$ and $v_L(1)$ represents the probability of the most probable path up to position 1 that ends in F or L. So we use this previous value and then ask for the probability of F→F transition or L→F, and the emission of the 2nd observed state and see which is higher. Here is a figure of the computations:



So we have $v_F(1)$x 7/8 x ½ = ½ x ½ x 7/8 ½ = 7/64 and $v_L(1)$x 3/8 x ½ = ½ x ¾ x 3/8 x ½ = 9/128. The quantity 7/64 is bigger than 9/128 so $v_F(2) = 7/64$ with the implicit choice of "F" in the first position.

By similar reasoning we compute the possible quantities for $v_L(2)$as $v_F(1)$x 1/8 x ¼ = ½ x ½ x 1/8 x ¼ = 1/128 and $v_L(1)$x 5/8 x ¼ = ½ x ¾ x 5/8 x ¼ = 15/256. In this case 15/256 is bigger than 1/128 so we set $v_L(2) = 15/256$ with the implicit choice of "L" in the first position.

We now have $v_F(2) = 7/64$ and $v_L(2) = 15/256$ for the second position and we can go on to the third position. But to recap, this tells us that if we had a "F" at the second position the most probable sequence to that point would be "FF" and the probability would be 7/64 and if we had a "L" at the second position the most probable path would be "LL" with probability 15/256. So, if we stopped here, we would consider the "FF" path and "LL" path and we would choose the path "FF" with 7/64 as the most probable path.

We continue the computations for the third position. The choice for $v_F(3)$ is $v_F(2)$x 7/8 x 1/2 = 7/64 x 7/8 x ½ = 49/1024 and $v_L(2)$x 3/8 x ½ = 15/256 x 3/8 x ½ = 45/4096 and therefore the bigger number is $v_F(3) = 49/1024$, with the implicit sequence "FFF". For $v_L(3)$ is $v_F(2)$x 1/8 x ¾ = 7/64 x 1/8 x 3/4 =

21/2048 and $v_L(2)$ x 5/8 x ¾ = 15/256 x 5/8 x ¾ = 225/8192 and therefore the bigger number is $v_L(3) = 225/8192$, with the implicit sequence "LLL".

Comparing $v_F(3) = 49/1024$ and $v_L(3) = 225/8192$ tells us that the path ending with F has higher probability and the implicit most probable hidden path is "FFF". You can check these computations against Table 4 above.

Our general strategy is at the ith step keep track of the most probable number up to ith step for each of the possible hidden states at the ith step. Then at the (i+1) step, consider the transition possibilities and their associated probabilities and choose the maximum value (thus "fixing" the most probable state path at the ith step). More precisely we have the following algorithm

$$set \quad v_0(0) = 1$$

$$v_k(i+1) = \left[\max_j (v_j(i) t_{jk})\right] e_k(x_{i+1}) \tag{1}$$

where the subscript 0 is for the beginning state, $e_k(x_{i+1})$ is the emission probability of the (i+1) letter from the kth hidden state, and t_{jk} is the transition probability from *j* to *k* hidden state.

The above algorithm is called the Viterbi algorithm. At each step we compute *O(m²)* numbers where *m* is the number of hidden states and we do this for the length of the sequence. So, if the length of the sequence is *n*, we have *O(nm²)* computational complexity, much better than *O(mⁿ)* computations if we naively enumerated the possible hidden state sequence.

Forward Algorithm

In the above, we noted that we have S^n different possible hidden state path, where *S* is the number of different hidden states and *n* is the sequence length. We then computed the most probable path conditioning on the observed state sequence. We next would like to compute the unconditional probability of the observed sequence up to ith position, segregated into different possible hidden states at the ith position. That is, we would like

$$P(x_1 ... x_i \text{ and } \pi_i = k)$$

where we abused notation and used $x_1 ... x_i$ to denote the observed sequence up to the ith position and $[\pi_i = k]$ to mean that the hidden state of the ith position is k. Then we have,

$P(x_1 ... x_i) = \sum_{k \in \Pi} P(x_1 ... x_i \text{ and } \pi_i = k)$ where Π is the set of possible hidden states.

Basically, the probability of the string $x_1 ... x_i$ is given by summing the conditional probability of the string given some hidden state sequence over all possible hidden states. This is seen in Table 4, where the probability of "010" string is the sum of all the probabilities in the table, since each cell is the conditional probability given the particular hidden state sequence. As in the probability for the most probable hidden state sequence, we cannot compute all the combinatorial possibilities. However, we can derive an algorithm exactly like the Viterbi algorithm except for one thing: instead of computing the maximum in Eq (1), we sum over the possible transitions. So first define:

$f_k(i) = P(x_1 ... x_i \text{ and } \pi_i = k)$

then the obvious boundary condition

$f_0(0) = 1$

and

$$f_k(i+1) = e_k(x_{i+1}) \sum_{j \in \Pi} f_j(i) \cdot t_{jk} \qquad (2)$$

where the notation is as in Eq (1). This Eq (2) along with the boundary condition is known as the **forward algorithm**. As can be easily seen the computational complexity of this algorithm is the same as the Viterbi algorithm (since summing or taking the max requires the same number of operations).

If we wanted to compute the probability of a particular sequence, say $x_1 ... x_n$, then we only need this forward algorithm and we would compute $P(x_1 ... x_n) = \sum_{k \in \Pi} f_k(n)$.

Backward Algorithm

We now develop an algorithm to compute the quantity $P(x_{i+1}, ... x_n \mid \pi_i = k)$ for reasons we will see in a moment. First, note that $P(x_{i+1}, ... x_n \mid \pi_i = k)$ is the probability of the suffix of the observed string starting from the i+1 position, given that the hidden state at the ith position was k. This time, we go backwards from considering this probability when we are at the (n-1) position. Using the example in Table 4 above, suppose we condition on the 2nd hidden position being F. Then we will see the suffix "0" (of the string "010") as the union of the following events:

F goes to F and emits "0"
F goes to L and emits "0"

We can compute corresponding probabilities and then sum them up in the usual way. Once we know this probability for hidden states F and L at the second position, we can move to conditioning on F or L at the first position and computing the probability of emitting the suffix "10" (out of the string "010"), and so on. We should now see that this again suggests a recursive computation. First we define:

$g_k(i) = P(x_{i+1}, ... x_n \mid \pi_i = k)$

Then the boundary condition

$g_k(n) = t_{kE}$ where E denotes the "End" hidden state.

This boundary condition requires a bit of an explanation. Note that $g_k(n)$ is supposed to mean the probability of the (n+1) position suffix in the observed string given the kth hidden state in the nth position. However, there is no (n+1) position suffix. The string ends at the nth position, so this suffix is the null string. In a real Markov process over the positions as we model it, the process should keep going and going and there is no reason that the string ends at the nth position. Therefore, we need to explicitly introduce an "End" hidden state from which point we will assume that the emission is null and the subsequent hidden state transitions are all to the "End" state; i.e., once in "E" always stay in "E". Then, we imagine that specifying t_{jE} of transition to the "End" state is part of the model. We did not give these numbers in the tables above. One could just give some nominal constant to these numbers. For more precise formulations, unfortunately, the "End" state construction has some technical problems that we will discuss later. So, at this point, we will just say we have some numbers we can fill in for the boundary condition.

Now back to the recursive formula. We have,

$$g_k(i) = \sum_{j \in \Pi} t_{kj} e_j(x_{i+1}) g_j(i+1) \qquad (3)$$

This formula just says if we want the probability of the (i+1) suffix conditioning on the kth hidden state at the ith position, we need to have the probability of the (i+2) suffix conditioned on the jth hidden state at the (i+1) position, transition between the kth and jth hidden states and the emission of the (i+1) letter.

Eq (3) with the boundary condition is the ***backward algorithm***. We can again compute the probability of the entire observed sequence using Eq (3). We have,

$P(x_1 ... x_n) = \sum_{k \in \Pi} g_k(1) e_k(x_1) \theta(k)$ where $\theta(k)$ is the marginal probability of the kth hidden state at the first position.

Decoding

The main reason to have both the forward and the backward algorithm is because we sometimes would like to compute $P(\pi_i = k \mid x_1 ... x_n)$ as we did in the simple examples above. That is, we would like to know the marginal probability of the hidden state at the ith position given the observed sequence. For some reason, this is called "decoding" the ith hidden state. In the simple examples, we found the sums of the right combination by looking at the combinatorial possibilities of the hidden states. Again, we cannot do that for the general case. But we note the following probabilistic equalities:

$$\begin{aligned} P(\pi_i = k \mid x_1 ... x_n) &= \frac{P(\pi_i = k \ and \ x_1 ... x_n)}{P(x_1 ... x_n)} \\ &= \left[P(x_1 ... x_i \ and \ \pi_i = k) P(x_{i+1} ... x_n \mid \pi_i = k) \right] / P(x_1 ... x_n) \\ &= f_k(i) \cdot g_k(i) / P(x_1 ... x_n) \end{aligned} \qquad (4)$$

where the denominator, $P(x_1 ... x_n)$, can be computed from either the forward algorithm or the backward algorithm as noted above. Obviously, the Eq (4) can be computed in $O(m^2 n)$ time, where m is the number of hidden states and n is the length of the sequence.

Parameter Estimation

So far we have carried out all our computation with the assumption that parameter values like emission probabilities (e.g., Table 1) and transition probabilities (e.g., Table 2) are given to us. However, these values are part of our model of the sequence family and, of course, must be obtained from a careful study of exemplars of the sequence family. Therefore, we need to estimate the parameter values from data. Estimation is a complex subject. The first rule to understand is that even within a rigorous formal probability models, estimation is not a deductive procedure—it is an inductive procedure. As such, we do not have strong first principles for how to estimate parameter values from data. Rather we have some ad hoc ways of constructing estimation methods, then we can study the behavior of such methods under various conditions such as what happens to our estimate when we add more and more random samples of data...

For the HMM, the parameters we need to estimate are the transition probabilities and the emission probabilities. We first look at the simpler case of Markov models without hidden states. Suppose we were given the following sequence of DNA

"AACATTACGGTACGT..."

We now wish to estimate the transition matrix for this data. The simplest thing we can do is to count the types of observed transitions. So, we have A to A, then A to C, then C to A and so on. We can fill in a frequency count matrix like this:

	A	C	G	T
A	1	3	0	1
C	1	0	2	0
G	0	0	1	2
T	2	0	0	1

Then we can simply divide the numbers by the total number of observed transitions (= 14) and use that as our estimates of the conditional probabilities. One might say we are using the "law of large numbers", which basically states that the frequency of events converge to their respective probabilities. In this case, we can also say that we are using something called method of moments—we will discuss this again later and we will not elaborate the idea here.

One thing we notice in the above table is that some of the numbers are zero like P(G|A). If we were to strictly interpret this, we would say that there can never be a transition from A to G. However, when we look at the data, it seems like this is more likely due to the fact that our string is short and we only have a few observed transitions. One solution to this problem called the "Laplace correction" is to assume we have at least one observation in each of the possible transitions. So that the table should look like this:

	A	C	G	T
A	1 + 1	3 + 1	0 + 1	1 + 1
C	1 + 1	0 + 1	2 + 1	0 + 1
G	0 + 1	0 + 1	1 + 1	2 + 1
T	2 + 1	0 + 1	0 + 1	1 + 1

Now our estimate of the probability of A to G transition is P(G|A) = 1/(14+16) = 1/30. The Laplace correction is a special form of something called Drichlet prior for Bayesian estimation. We will discuss such priors in detail later.

Another method of estimating the parameter values is to compute the function P(D|M(Θ)) where D is the observed data and M(Θ) is the model and parameter combination. For example, we might assume that the transition matrix looks like this:

	A	C	G	T
A	1- 3a	a	a	a
C	b	1-3b	b	b
G	c	c	1-3c	c

T	d	d	d	1-3d

Given this matrix the probability of the observed string "AACATTACGGTACGT" is:

$$L = P(D \mid M(\Theta))$$
$$= P(A)\cdot(1-3a)\cdot a\cdot b\cdot a\cdot(1-3d)\cdot d\cdot a\cdot b\cdot(1-3c)\cdot c\cdot d\cdot a\cdot b\cdot c = P(A)(1-3a)a^4b^3(1-3c)c^2(1-3d)d^2 \quad (5)$$

Probability of the data given the model is a well-defined probability. However, here we are interested in the model, not the data. For the function given in Eq (5), we are interested in this as a function of the model's parameters, not so much the data. Therefore, we call this the ***Likelihood*** of the model and the parameter values. Now we can fix the values of *a, b, c*, and *d* by maximizing the likelihood with respect to possible values of *a, b, c,* and *d*. This is called the ***Maximum Likelihood (ML)*** estimate.

We next treat the special case of the profile HMM. In the HMM case, we can still use the method of moments, but unfortunately we don't have the hidden states as observations. Remember in the profile HMM at each observed letter the hidden state is either Insertion (I), Deletion (D), or Match (M) of the ith type. The heuristic idea is to use a multiple alignment to directly fill in the hidden states. So suppose we are given a multiple alignment:

```
ACCT--
A-CTTT
A-GTTC
A--CTT
A-CTT-
```

Then we assume that the aligned "-" positions directly indicate "I" and "D". Of course, a given "-" could be either I or D; so we use another rule of thumb that says if a given aligned column has majority of "-" then the cases where there are nucleotides are insertions. For example, the second column of the above alignment has C in the first string and "-" everywhere else. We just assume that the first "C" is an insertion. On the other hand, the 5th column has "-" in the first string and filled in nucleotide "T" in all other strings. Then we assume that the "-" in the first string is a deletion state. Given these kinds of rule of thumb, we can now substitute the implied hidden state for each string in the alignment:

ACCT-- $\rightarrow$ M, I, M, M, D, D
A-CTTT $\rightarrow$ M, M, M, M, M
A-GTTC $\rightarrow$ M, M, M, M, M
A--CTT $\rightarrow$ M, M, D, M, M
A-CTT- $\rightarrow$ M, M, M, M, D

In the first string, "ACCT--", we deduce that the second letter is an insertion and the last two letters are missing by deletion. In the second string, "A-CTTT", it may be confusing because it seems like we are ignoring the "-" in the second position. However, recall that for this position it is the first string that has an insertional state and therefore the "-" is not an emitted observation for any of the other strings. Thus, we do not count this position and rest of it is all M states. Since we now have the hidden states, we can again use the counting method to deduce the transition probabilities. First, in our observation from the begin state, we always make a transition to M; thus P(M|B) is estimated as 1 and all other transitions have probability zero. In the second transition, we have 1/5 probability of going to an insertion state and 4/5 probability of going to the next match state. We can continue these counting to fill in the estimated transition probabilities—and we can also use Laplace corrections here as well.

Emission probabilities are computed similarly. For example, in the first match state all of the emission seems to be A's thus one might say the emission probability in match state 1 is A with probability 1 and zero for all other letters—again we can employ Laplace correction.

Using equation (2) or (3) we can compute the probability of the observed sequence given the model. Thus, we can use the ML estimator. For example, we can numerically optimize the likelihood by trying various values of emission probabilities and transition probabilities. Numerical optimization is an advanced subject and we won't cover the material here. Many packages are available to numerically optimize a function. The main problem for optimization is that if the number of parameters is very large—as is often the case in HMM's then typical numerical optimization routines have a very difficult time. One method for optimization that is in principle straightforward regardless of the number of parameters is called the Baum-Welch algorithm, which is a particular case of a more general algorithm called Expectation Maximization (EM) algorithm. Although we do not cover it in detail here, the basic idea has two steps.

(1) Start with arbitrary parameter values
(2) Compute the conditional expectation for the hidden state transitions given the data
(3) Use these conditional expectations as the estimate for the parameter values and go back to step (2)

The EM algorithm types are robust in the sense that it can be implemented without bad numerical behavior and given enough iterations, it will converge to some local optimum. However, the convergence rate can be very slow, especially compared to some fancy gradient methods.

Box: Models, parameters, and parameter estimation.

A model is a tentative explanation of observed data. Often, we consider the model as "generating" the data; e.g., we might have a model of the process that involves converting a pool of mRNA molecules into nextgen sequence reads. Other times, a model could simply be justified as "the collection of data tends to look like a sample from this model". For example, when we model the collection of body weights of all students in this class with a normal distribution, we simply mean that the collection of body weights looks like random samples drawn from a normal distribution. We don't mean that a normal distribution is somehow connected to how our bodies grow.

Nearly all models are probabilistic. That is, have some random component. This is true for even deterministic events like a model of the trajectory of a bullet. This is because at a minimum our measurement devices have some randomness. All models are actually a family of models rather than a single model. For example, when we talk about normally distributed body weights, the family of models involves normal distributions with various means and variances. That is, the family of normal distributions is indexed by two numbers, the mean and the variance. Such numbers are called the parameters of the model.

Given some data, we wish to fit the data to some model. This means we first choose some family of models; e.g., the normal distribution. Then second, we have to choose some values for the parameters. Sometimes we are interested in the value of the parameters. For example, we might have a model that says the number of RNA sequencing reads from the gene GAPDH is modeled by a normal distribution with mean μ and variance σ^2. Then given some data, we might want to know μ as the best representation of expression levels. Conversely, we often propose a model then ask whether the model is reasonable given the data. That is, we either test the hypothesis associated with the model using the data, or try to compute the probability of some model given the data. In both cases, we try to find the parameter values that is most appropriate for the data---we use the observed data to estimate the parameters of the model; or, we say we try to choose the best fitting model by choosing the optimal parameter values.

Suppose we have a probability model $P_\theta(X)$ for observed data represented by the random variable X and the parameter set represented by θ. (I note that X could have some complicated values like a matrix of aligned sequences and θ could be also a complicated set of things like a vector of numbers.) To make things concrete, suppose we are throwing some biased coin with probability of heads equal θ and X = 1 for heads and X = 0 for tails. So, we have,

$$\begin{cases} P_\theta(X=1) = \theta \\ P_\theta(X=0) = 1-\theta \end{cases} \qquad (1)$$

A dataset is usually multiple observations, each of which is assumed to belong to the model. For example, we might have 10 flips of the coin resulting in the observation:

(1, 0, 0, 1, 1, 0, 0, 0, 0,1) (2)

We note that the above data set size, n, is 10. The size of the dataset is an important variable in our estimations we will see below.

We want to estimate θ given such dataset. There are many possible principles we can apply to estimate θ. It is important here to remember that there is no "first principles" for finding a good estimator. This is because estimation is an inductive process and not a deductive process. Classically, there are four different ideas that are commonly used:

1. Method of moments
2. Least squares, or more generally minimal risk
3. Maximum likelihood
4. Bayesian method

Method of moments

The idea here is that if I propose a probability model for the data, I can usually compute moments of the probability distribution. Then I can also compute the moments of the data. Moments of a distribution have the form $E(X^k)$ where E() means expectation and X^k is the random variable raised to the kth power. Central moments are defined as $E\left(\left(X-E(X)\right)^k\right)$, that is, *E(X)* is subtracted before taking kth power (for k > 1). You can read about moments here: https://en.wikipedia.org/wiki/Moment_(mathematics). For normal distributions whose probability density function is given as $\frac{1}{\sqrt{2\pi\sigma^2}}e^{\frac{-(x-\mu)^2}{2\sigma^2}}$, 1st moment is μ and 2nd central moment is σ^2. We can also compute the same kind of moments for the data. For example, the mean and variance of the dataset. Then we can equate the sample moments and the model moments.

For the coin flip example above, the $E(X) = 1\cdot\theta + 0\cdot(1-\theta) = \theta$. And, for the data shown in (2), the sample 1st moment (i.e., the mean) is $1\cdot\frac{1}{10}+0\cdot\frac{1}{10}+0\cdot\frac{1}{10}+1\cdot\frac{1}{10}+1\cdot\frac{1}{10}+0\cdot\frac{1}{10}+0\cdot\frac{1}{10}+0\cdot\frac{1}{10}+0\cdot\frac{1}{10}+1\cdot\frac{1}{10}=\frac{4}{10}$. We equate the sample 1st moment with the model 1st moment, so our estimate of θ is 4/10. (I will use the notation $\hat{\theta}$ for the estimate of θ.)

I should note that moments of a probability distribution are always functions of the parameters. For example, the 1st and 2nd central moments of a gamma distribution are α/β and α/β^2 where α and β are the parameters of the gamma distribution (see https://en.wikipedia.org/wiki/Gamma_distribution). In this case, we would compute sample mean and sample variance and then solve for α and β.

Least Squares

The idea of least squares is to use the model to predict values of the data. Then we measure the deviation of the prediction from the data values by computing the squared deviation. Suppose the data set is $(x_1, x_2, \ldots, x_n)$ and we model it as samples from a normal distribution. Then our best guess at the data is the mean of the normal distribution, μ. So, we compute sum of the squared deviation from our estimate: $SS = \sum_{i=1}^{n}(x_i - \mu)^2$. Least squares means we want to find the value of μ that minimizes the squared deviation. Going back to the coin flip example, we want to estimate a $\hat{\theta}$ value that minimizes,

$$SS = \left(1-\hat{\theta}\right)^2 + \left(0-\hat{\theta}\right)^2 + \left(0-\hat{\theta}\right)^2 + \left(1-\hat{\theta}\right)^2 + \left(1-\hat{\theta}\right)^2 + \left(0-\hat{\theta}\right)^2 + \left(0-\hat{\theta}\right)^2 + \left(0-\hat{\theta}\right)^2 + \left(0-\hat{\theta}\right)^2 + \left(1-\hat{\theta}\right)^2$$

We can solve this problem by using calculus to take the derivative of SS with respect to $\hat{\theta}$, which will result in $\hat{\theta} = 4/10$, the same as method of moments. If we can't solve the problem analytically, we can use numerical optimization methods to numerically guess at the value of $\hat{\theta}$
that minimizes the sum of squares.

Squared deviation of prediction from observed values is a way of measuring what we call ***loss***; that is, the loss of the prediction from the values. We can define many different kinds of loss functions. For example, $L(x, \theta) = (x - \widehat{\theta(x)})^4$ or $L(x, \hat{\theta}) = |x - \widehat{\theta(x)}|$, etc. In this notation, $\widehat{\theta(x)}$ is a function from the model that predicts the x value. A generalization of least squares is to compute the expectation of the loss function over the data: $R(D, \theta) = \frac{1}{n}\sum_{i=1}^{n} L(x_i, \theta)$. We call this function the empirical risk function for the data set *D*. We then want parameter estimate $\hat{\theta}$ that minimizes the empirical risk.

Maximum Likelihood

If we have a good probability model of the data, then we can compute $P(D|\theta)$, the probability of the data given the probability model and parameter value θ. This quantity is called the likelihood of θ. It is not the probability θ, because it is the conditional probability of the data, not θ. If we can compute $P(D|\theta)$, then it is a natural idea to try to find the θ value that maximizes the data probability. This is called the Maximum Likelihood (ML) estimate of θ. For the coin flip example above, the probability of the particular sequence of outcomes, (1, 0, 0, 1, 1, 0, 0, 0, 0,1), if we assume each throw is independent of each other. Then the probability of the data set is the product of the probability of each throw outcome,

$$P(D|\theta) = \theta \cdot \theta \cdot (1-\theta) \cdot (1-\theta) \cdot \theta \cdot \theta \cdot (1-\theta) \cdot (1-\theta) \cdot (1-\theta) \cdot (1-\theta) \cdot \theta$$

or,

$$P(D|\theta) = \theta^4(1-\theta)^6 \quad (3)$$

(I note that this is the probability of the particular sequence of 1's and 0's. If we wanted to compute the probability of 4 heads and 6 tails, regardless of the order, then we would have to consider all the combinations of so many heads and tails, resulting in $P(D|\theta) = \frac{10!}{4!6!}\theta^4(1-\theta)^6$.)

We can now try to find the maximum of (3). We do this directly but it is more convenient to maximize the log of the likelihood,

$$Ln[P(D|\theta)] = 4 \cdot Ln\theta + 6 \cdot Ln(1-\theta) \qquad (4)$$

A bit of calculus will result in the maximum likelihood estimate, $\hat{\theta} = 4/10$, which is the same estimate as method of moments and least squares.

Bayesian method

While we can compute $P(D|\theta)$, what we usually would like is $P(\theta|D)$, the probability of the model parameter value being some particular value, given the data *D*.

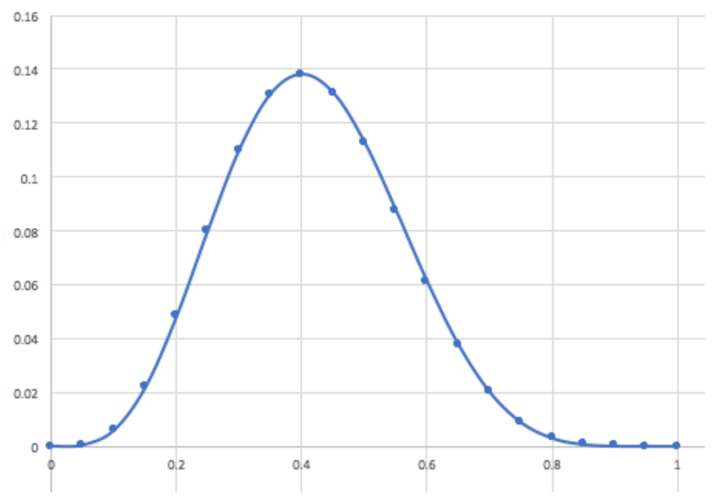
From the probability primer for Bayes' formula we have,

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{\int P(D|\theta)P(\theta)d\theta} \qquad (5)$$

where the denominator is integration of probabilities over all possible values of θ. If the probability model is discrete, then we sum over all the possible discrete values of θ. To compute the above quantity (5), we have to know $P(\theta)$, that is the probability density function of θ without any conditioning on data. This is something that does not depend on the data and therefore it is called the ***prior probability*** (of θ). The term on the left-hand side of (5), $P(\theta|D)$, probability of θ equals some value given the data, is called the ***posterior probability*** (of θ). The idea of Bayesian estimate is to compute this posterior probability of the parameter of interest. Going back to the coin flip example, $P(D|\theta) = \theta^4(1-\theta)^6$. Suppose we assume that θ ranges from zero to one with equal probability, so-called uniform probability distribution. Then $P(\theta) = 1$ for all θ. Bayes' formula becomes

$$P(\theta|D) = \frac{\theta^4(1-\theta)^6}{\int_0^1 \theta^4(1-\theta)^6 d\theta}$$

We note that $\int_0^1 \theta^4(1-\theta)^6 d\theta$ is a definite integral so it is some constant value. Let's called this 1/C. Then $P(\theta|D) = C \cdot \theta^4(1-\theta)^6$. We can plot this function out for values of θ ranging from 0 to 1.



An important point from this picture is that Bayesian estimate of θ is not a number but a probability distribution. If we want to turn this into a number, then we can compute some statistic of the probability distribution. For example, we can find the mode of the distribution, which in this case will be 4/10. Or, we can compute the mean of the distribution, which turns out to be slightly bigger than 0.4. The former is commonly called maximum a posteriori (MAP) estimate.

Unit 9: Phylogeny Estimation--Overview

All biological objects are related to each other by evolutionary history. (Evolutionary history includes recent parent-offspring relationships.) Therefore, estimating the history of the objects is both an important tool for making all kinds of inferences and an important framework for analyzing the dependence structure of biological information. The latter is especially important. For example, we might have 100 proteins that have three sequential hydrophobic amino-acids at the carboxyl end; but, it may turn out that all of them are all very closely related and the three amino-acids were present in their common ancestor (by chance). Then, instead of having 100 data points supporting potential functional role of the hydrophobic amino-acids, we have approximately one data point!

Some uses of phylogenies include:

- Estimating history, including recent history like viral infection pathways
- Classifying or grouping biological objects such as gene families
- Estimating the order of evolutionary events to understand biological processes such as development (e.g., understanding that ancestor to whales used to have four walking legs)
- Estimating the parameters of evolutionary events such as accelerated evolution of some genes in the human lineage
- Understanding the origin of genomic events such as gene fusion, horizontal transfer of genes, etc.
- Revealing dependency relationships of the objects

Alignments identify positional homologies of symbols in a biological sequence. Phylogenetic estimation methods use the information in the identified homologies to infer the genealogical history of the evolution of objects (e.g., proteins, DNA substrings, tissues, whole organisms) that display the homologous features. When a homologous feature has been identified between two or more biological objects, we will call it a (evolutionary) **character**. While DNA/protein sequence is the most common kind of character, there are many other possible character classes for phylogeny inference. For example, some possible characters are:

1. position 1 of a set of aligned sequences; more generally each ith position of aligned sequences
2. presence of a particular protein
3. presence of a particular secondary structure motif in a protein
4. a particular organ that is common between two organisms; say the heart
5. immunological response to an antibody
6. a morphometric feature, say the width of a mouth, weight of the brain, etc.

Each character typically has multiple possible values amongst the biological objects of interest. The possible values are called **states of the character**. The states of the character for the above set might be:

1. {A, C, G, T}
2. 0/1 (presence/absence)
3. 0/1
4. 2, 3, 4 chambers (ordinal values)

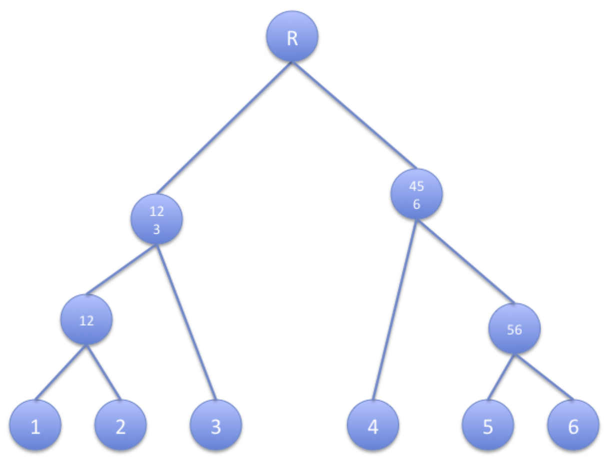
5. {weak, medium, strong}
6. real numbers

Not all identified homologous characters might have variable states within the set of objects of interest—for example, while a heart might be a homologous character the particular group of organisms you are interested in might all have 4 chambers. Or, more commonly, given a set of aligned DNA sequences, some of the positions may be **invariant** in that they all have the same nucleotide. Such invariant characters can still be informative about rate of a process if all characters are assumed to be evolving under a common evolutionary process (see later sections). Also note that "absence" of something might also be a state of a character. This is a very common situation, such as the character "has vertebrae/ does not have vertebrae". In these cases, usually one of the states (e.g., has vertebrae) is considered an **evolutionary derived state**; i.e., a character state that arose later in evolutionary history.

Some times characters are called **traits**, which is generally identical to characters. In some computer science literature, characters are called "vertex colorings" or "vertex color functions" (and the states are called "colors").

The basic phylogenies are rooted tree graphs. The vertices of the graph represent current or ancestral objects and the root is a special vertex that is considered the common ancestor to all other vertices. (Traditionally, phylogenetic trees were developed to understand organismal evolution so the vertices were called **taxons**.) The edges of the graph represent genetic lineages. Sometimes the genealogical histories of the objects are such that the graph is not a tree but a network—that is, has circuits in the graph paths. The most common example of such network relationships is the two-sex dual parent-child relationship where every child has two genetic edges to each of the parents. Other kinds of networks include cases of horizontal gene transfer, hybridization between lineages (rather common in plants), etc. For this course, we will mostly consider only tree graphs, but these other kinds of network graphs are also important and algorithms have been devised to estimate network graphs. I note here that an edge in our genetic lineages has direction in terms of time (parent-offspring relationship) but may not have direction in terms of informational relationship. That is, an offspring is clearly derived from a parent but the genetic information of the offspring given the parent and the genetic information of the parent given the offspring may be symmetrical.

For the tree graph, the most common standard case has vertices of three types: a root vertex with two edges to its daughter vertices; internal vertices with three edges, one from the parent and two leading to the daughters; and leaf vertices with one edge from its parent. Such trees are equivalently called **bifurcating trees** (each lineage bifurcates to two daughters), **binary trees**, and **trivalent trees**. We will generally call them binary trees. A tree may have vertices with more than two daughters (called an **polytomy**—"many split") but in general we will assume that such a tree can be represented by a binary tree with special edges (zero length; see next). Each degree-three vertex and the root vertex correspond to hypothetical ancestral objects while the leaf vertices correspond to extant objects (or less commonly, ancestral objects that went extinct, left no daughters, but for which we have character data).



For each hypothetical ancestor in the tree, there is the set of leaves that are all descendants of that ancestor. Such a group of leaves is called a "**clade**". A tree consists of various sets of clades. Suppose we have a tree with *n* leaves (an *n*-leaf tree). Let **L** = {*1, 2, …., n*} be a list of names for the leaves of the tree; we also call this **leaf labels**. (In an empirical example, the names might be {human, chimp, …orangutan}.) Given a tree, the clades of the tree delimit subsets of **L**. For example, for the 6-leaf tree above we have the subsets {1, 2, 3} as one clade (descendants of the vertex labeled 123) and {4, 5, 6} as another clade (descendants of vertex labeled 456). Another way to depict a tree is to think of it as a collection of the leaf label subsets, each corresponding to a clade:

{1}, {2}, {3}, {4}, {5}, {6}, {1,2}, {5,6}, {1,2,3}, {4,5,6}, {1,2,3,4,5,6} (T1)

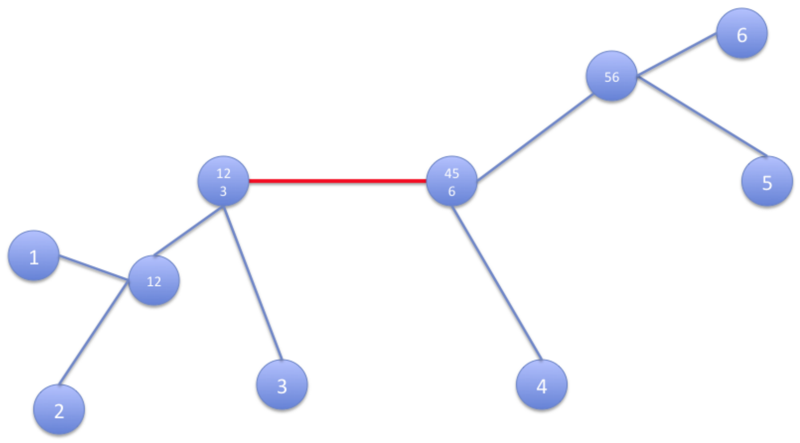
This list contains the leaves as singleton clades. It also includes the clade from the root, which is the whole set of leaf labels. The collection (T1) corresponds exactly to the clades of the tree in the figure above.

In general, let $\mathbf{2^L}$ be the power set of the leaf label set (i.e., the collection of all possible subsets). Then a tree graph can be identified with a subset, *T*, of the power set (i.e., a collection of some subsets), such that if $s, t \in T$ are two members of *T* (i.e., two subsets of the leaf labels), then either $s \cap t = \phi$ or $s \subset t$ or $t \subset s$. (See the collection in T1). This means that two subsets of leaf labels representing clades are either disjoint or one is nested within the other. In the collection (T1), for example {1,2} is disjoint from {4,5,6} and contained within {1,2,3}. This correspondence between tree graphs and set of subsets of leaf labels will be useful later in comparing two different trees. Because of this nesting property all the clades of the tree form a nested set of subsets that can be depicted by a set of nested parenthesis:

{{{1,2},3},{4,{5,6}}}

for the tree above. It is more common in the literature to see normal brackets like this: (((1, 2),3),(4(5,6))) rather than the curly brackets. We will use them interchangeably.

We will see later that inferring the root of a tree is problematic so we will mostly deal with unrooted trees. An unrooted tree can be obtained from a rooted tree by deleting the special root vertex and replacing the edges with a single edge like this:



In the above figure, I also moved around the vertices a bit so that it no longer looks directed. You should note that the graphic layout of a tree does not have special meaning unless the edges or the vertices have weight functions (see later). You should learn to recognize equivalent trees even if they are laid out differently. Given an unrooted tree, we can obtain a rooted tree by picking some edge and then replacing the edge with a special root vertex and inserting two edges. For an *n*-leaf unrooted binary tree, there are *2n-2* vertices including the leaves and *2n-3* edges. Therefore, if I have an *n*-leaf unrooted binary tree there are *2n-3* possible ways of obtaining a rooted tree from the unrooted tree. We will see later that the "finding the root" corresponds to finding a root-containing edge amongst the *2n-3* edges. An unrooted tree can also be identified with a collection of subsets of leaf labels as follows. Since this is a tree graph, if we delete an edge, the tree will split up into two graphs. This operation will also split the leaf label set. For example, in the tree above, removal of the red edge will result in the split of the leaf labels into {1, 2, 3} and {4, 5, 6}. We will write such a split as {1, 2, 3 | 4, 5, 6}. An unrooted tree graph can be identified with the leaf label splits for all possible deletion of a single edge. So the tree above has the following splits:

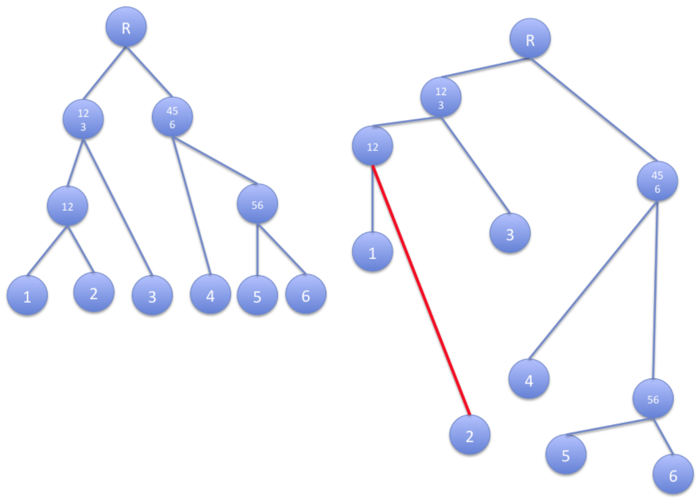
{1| 2,3,4,5,6}, {2|1,3,4,5,6}, {3|1,2,4,5,6}, {4|1,2,3,5,6}, {5|1,2,3,4,6}, {6|1,2,3,4,5}
{1,2| 3,4,5,6}, {5,6|1,2,3,4}
{1,2,3| 4,5,6}

The splits have the form {*s*|*L-s*} where *s* is some subset of *L* and *L-s* is the complement of *s* in *L*. The information in *s* and *L-s* is redundant if *L* is fixed; so we can organize the splits by choosing some arbitrary leaf label and only listing the subsets that exclude that leaf label. For example, if we choose the label "1":

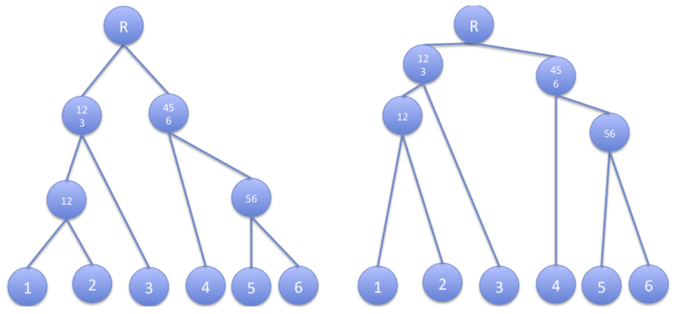
{2,3,4,5,6}, {2}, {3}, {4}, {5}, {6}, {3,4,5,6}, {5,6}, {4,5,6}

This collection now has the same property as the collection of clade subsets from a rooted tree. Namely, $s \cap t = \phi$ or $s \subset t$ or $t \subset s$ for any *s* and *t* in the collection. The correspondence between subsets of leaf labels or splits of leaf labels motivates certain class of algorithms that tries to identify such splits or clades and assemble the compatible ones (the ones that are consistent with the nested subset relationships) into a tree graph.

So far we have ignored the edges (also called branches) of the tree graph. If we don't specify anything else the edges only depict genealogical relationships and the lengths do not have any meaning. So the two trees below mean the same thing.



When we ignore the information in the edges, we say we are only interested in the "**topology**" of the tree; that is, the subset relationships of the leaf labels discussed above. It is more common to attach information to the edges in the form of "edge weights" or "edge (weight) functions". That is, for each edge *e* in the tree graph, we have an edge function *w(e)*, which maps to some information set about that edge. The most common information set is time, so that $\varpi: E \to \mathbb{R}$ maps edges to real numbers that depict either real time or number of generations that separate the two vertices connected by the edge. In this case, two trees with different graphical depiction of their edges may be different trees. For example:



When the leaves depict extant objects and the edges are proportional to time, all the leaves of the drawn tree must reach the same horizontal line. More generally, suppose we have a non-negative real-valued edge weight function $\varpi: E \to \{0, \mathbb{R}^+\}$, then path distance between any two vertices, *v* and *w*, of the tree graph is defined as:

$$d(v,w) = \sum_{e_i \in v \to w} \varpi(e_i)$$

(Eq 1)

That is, the sum of the edge weights in the unique path between vertices *v* and *w*. If the edges depict time and all of the leaves are contemporaneous we have for any triplet of vertices *v, w, x*:

$$d(x,v) = d(x,w) \geq d(v,w)$$

(Eq 2)

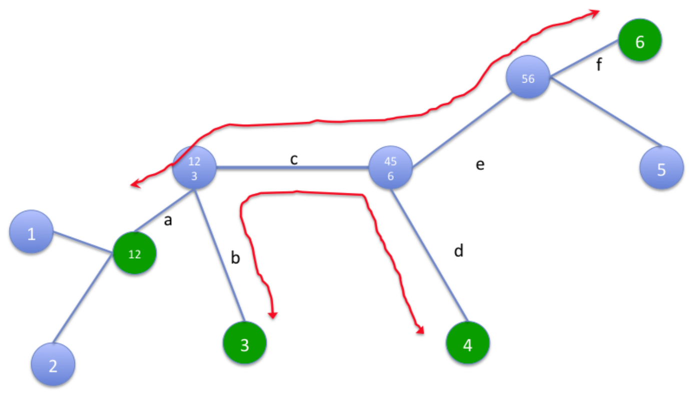
You can see this in the tree above, where you can see d(4,5) = d(4, 6) > d(5,6). When the path distances of a tree satisfy such a relationship, we call the tree an **ultrametric tree**. Sometimes we also call this a "clock tree". It is common term in the literature to speak of a "molecular clock tree", where the tree has been estimated to satisfy Eq 2 constraints using molecular sequence data.

Except under particular model circumstances, it is very hard infer time as part of the information for edges of a phylogenetic tree. More commonly, the edge weight functions depict things like the number of mutational edit operation between the two vertex objects; or under a stochastic model framework, the *expected* number of mutational edit operations. Thus, some edges might be much longer than others if the biological process represented by the edge is, say, accelerated. This is shown by the red edge in the tree above. Sometimes we might use such information. In the tree above, the edge leading to vertex 1 is relatively short compared to the edge leading to vertex 2 (the red edge). We might therefore infer something about the molecular evolution process leading to vertex 2—say, call it "accelerated adaptive evolution." [However, we will see later that simply looking at edge weights is not sufficient to make such a biological claim and we need to be more careful. (For example, how do we know it is not the edge leading to vertex 1 that is unusually slow due to lowered mutation rate?)] If the edges depict expected number of evolutionary edit operations (or more concisely, "evolutionary changes"), then the leaf vertices do not have to line up and we don't satisfy the ultrametric constraints (Eq 2). However, for any tree with non-negative real-valued edge functions, we have the following relationship between any four vertices, *w, x, y*, and *z*:

$$d(w,x) + d(y,z) = d(w,z) + d(x,y) \geq d(w,y) + d(x,z)$$

(Eq 3)

Eq 3 is called the 4-point condition. The 4-point condition holds because of the following observation of paths on a tree:



In the figure above, the vertices labeled 12, 3, 4, and 6 forms a **quartet**. We have the following 6 pairs of distances for the vertices of the quartet:

$d(12,6) = a + c + e + f$
$d(3, 4) = b + c + d$
$d(12,3) = a + b$
$d(6, 4) = d + e + f$
$d(12, 4) = a + c + d$
$d(3,6) = b + c + e + f$

The path for d(12, 6) and d(3, 4) are drawn in red in the figure. You can see that

$d(12, 6) + d(3, 4) = d(12,4) + d(3, 6) > d(12, 3) + d(6, 4).$

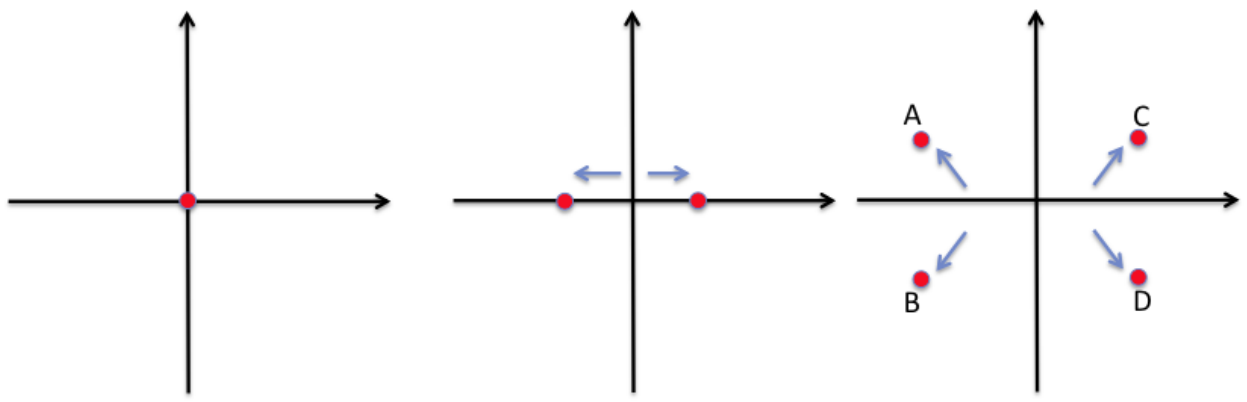
Peter Buneman proved that such a relationship holds for every quartet of vertices on any tree graph as long as the edge weights are always non-negative.

Given an *n*-leaf tree graph with non-negative real-valued edge weights, we can use the path distance to generate a *n*-by-*n* distance matrix satisfying Eq 3. Furthermore, if we have an *n*-by-*n* distance matrix satisfying Eq 3 then there is a unique tree graph with non-negative real-valued edge weights such that path distances on this tree graph exactly generate that *n*-by-*n* distance matrix. Distance matrices satisfying Eq 3 are called **Additive Distance** matrix. There is a one-to-one relationship between tree graphs with non-negative real-valued edge weights and additive distance matrices, which gives us one important class of phylogeny algorithms as we will see next.

Unit 10: Additive Distance Method

To estimate a phylogeny we are given a set of characters measured for a group of objects, such as an alignment of sequences, and we try to associate the measurements with a tree graph. We have already seen one algorithm with this kind of input/output: the guide tree generation for multiple sequence alignment. Recall that the UPGMA clustering procedure takes as input, pairwise distances, and then produces a tree graph that depicts a hierarchical similarity relationship between the objects. A tree graph of genealogical relationship between objects is generally expected to produce hierarchically structured similarity relationship between the objects. Note, that on the other hand, a set of hierarchical relationships is NOT necessarily the result of genealogical relationships. Despite this caveat, it might be reasonable to think about a clustering procedure as a method for phylogeny inference. Early attempts at computational phylogeny estimation employed various kinds of clustering procedures including the UPGMA. To see how this might work, we first start with a toy model. (A "toy model" is a technical term to denote a simple model that might be considerably removed from the actual process but still captures the broad principles of the phenomenon of interest; usually used to demonstrate an idea.)

Imagine a particle in 2D Euclidean space. Starting at the coordinate (0,0) the particle splits into two and moves East and West respectively at velocity 1 m/hr. Then an hour later, each of the two particle splits again and also acquires a 1 m/hr velocity in the 45 degree direction moving away from each other:



After two hours, the coordinates of the four particles, which we will name A, B, C, D, are $(-1-1/\sqrt{2}, 1/\sqrt{2}), (-1-1/\sqrt{2}, -1/\sqrt{2}), (1+1/\sqrt{2}, 1/\sqrt{2}), (1+1/\sqrt{2}, -1/\sqrt{2})$, respectively. The Euclidean distance matrix for A, B, C, D is (approximately):

	A	B	C	D
A	0	1.4	3.4	3.7
B		0	3.7	3.4
C			0	1.4
D				0

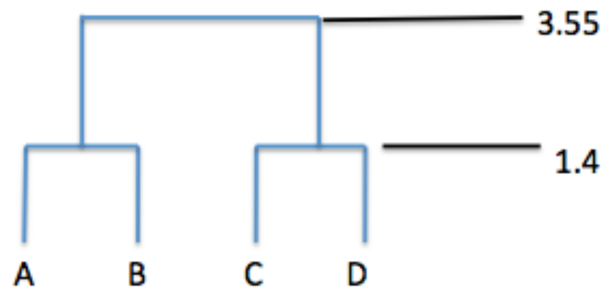
We will call the above distance matrix D0. Applying the UPGMA algorithm to D0 we get:

	A-B	C	D
A-B	0	3.55	3.55
C		0	1.4
D			0

	A-B	C-D

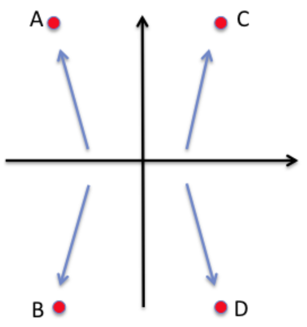
A-B	0	3.55
C-D		0

And the tree:



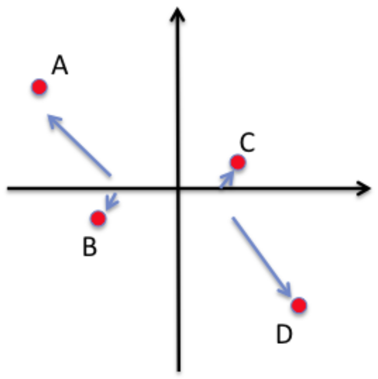
The numbers on the edges reflect the implied distances from the cluster distance matrix we calculated. The above tree and the edges reflect the history of splitting particles pretty well. The tree topology was correct albeit the edge weights are a bit off from being proportional to the actual times of split. Ideally, we would like to have had 1:1 ratio between the edges that represent the 1 hr duration of the original split and then the 1 hr duration of the second split—especially since the velocity was constant during all history. The ratio here is 2.1:1.4—not too bad, perhaps. Typically, the main object of interest for phylogeny inference is the tree topology; however, there are many situations, especially when we want to infer the degree of relationship or rates of molecular evolution, where we would like good estimates of the edge weight functions. At least for this example, the UPGMA method is giving us the correct tree topology but not giving us optimal edge weight estimates.

Now we consider an alternative scenario. In this history, the first particle splits and moves E-W with 1 m/hr velocity, but after the second split, the four particles now has 3 times the velocity in the y-axis direction compared to the x-axis direction; see the figure to the left. If we assume, 1 m/hr in x direction and 3 m/hr in y direction, then after two hours the coordinates for A, B, C, and D are (-2,3), (-2,-3),(2,3), and (2,-3) respectively with the following Euclidean distance matrix (denoted D1):



	A	B	C	D
A	0	6	4	7.2
B		0	7.2	4
C			0	6
D				0

We don't have to finish the UPGMA calculations to see that this matrix will not allow us to reconstruct the tree topology correctly by the UPGMA clustering algorithm.



Here is a third alternative scenario. Again, we have the same initial split and the 1 m/hr divergence. Then at the second set of splits, we follow the original equal velocity in x and y directions (say 1 m/hr in both x and y vector components) but the pair of particles B and C move at 1/5 the speed (0.2 m/hr in x and y components). We will have the following coordinates after, say 4 hours: (-4,3), (-1.6,-0.6), (1.6, 0.6), (4,-3), for A, B, C, and D, respectively. We will also have the following distance matrix (denoted D2):

	A	B	C	D
A	0	4.3	6.1	10
B		0	3.4	6.1
C			0	4.3
D				0

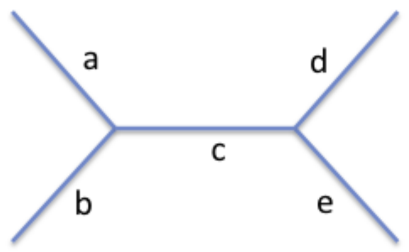
Again, we can immediately see that the UPGMA procedure will produce the wrong tree topolgy (because B and C are the closest pair).

The two scenarios that led UPGMA to the wrong answers represent two different kinds of problems that phylogenetic algorithms must try to solve. The first problem with D1 is more complex. In this case, the movement along the y direction is effectively making the particles A and C (also B and D) converge towards each other relative to their original displacement along the x direction—this is actually a complicated type of problem and we defer its discussion. The second problem arises out of the naïve UPGMA algorithm not explicitly taking into account the tree-graph generative process. That is, UPGMA is simply trying to group things into hierarchical organization without explicitly treating the objects as associated with a tree branching process. In fact, the trees constructed by the UPGMA clustering method are ultrametric (see Eq 2), whereas the process described for D2 result in a history that is not ultrametric. Therefore, we need a method that allows for non-ultrametric relationships and explicitly uses a tree-graph model as the underlying structure for the relationships.

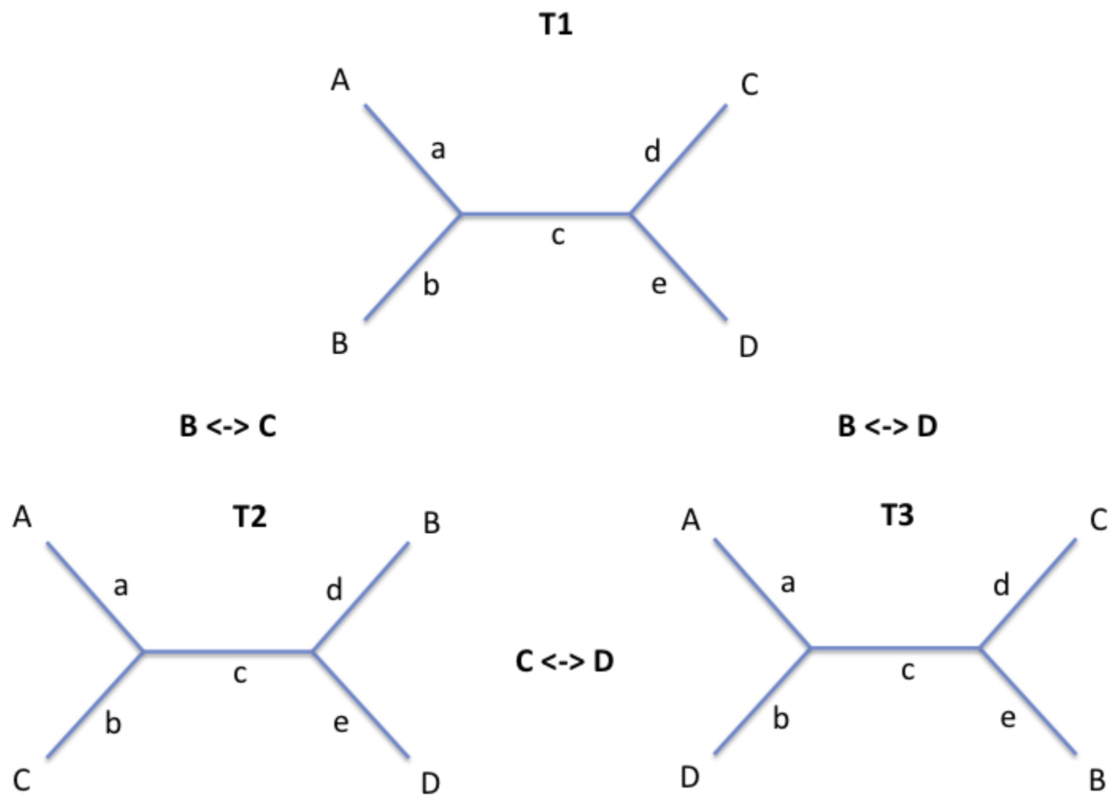
As mentioned previously, there is a bijective relationship between non-negative edge weighted tree graphs and certain kinds of distance matrices that we called additive distance matrices. Additive distance matrices are supposed to satisfy the 4-point condition (Eq 3). We can check the 4-point condition on the D2 matrix:

d(A,B) + d(C,D) = 4.3+4.3 = 8.6
d(A,C) + d(B,D) = 6.1+6.1 = 12.2
d(A,D) + d(B,C) = 10 + 3.4 = 13.4

It is easy to see that the 4-point condition does not hold. But, if the D2 matrix were generated by path distances over a tree rather than the direct leaf-to-leaf distances, the matrix should satisfy the 4-point condition. Therefore, the idea of taking into account the tree generative process is to assume that the distance matrix should be additive; and if it isn't, try to find an additive matrix that is as close as possible to the observed matrix. How do we do this? What we do is write out the elements of the distance matrix as a function of path distances on a tree graph. If we have four objects, they have to be related to each other on a binary tree like this:



where (a, b, c, d, e) denotes the edge weights. I should note that since we are simply thinking about pairwise distance relationships the position of the root vertex doesn't affect the pairwise distances. Thus, we will only consider unrooted trees. We discuss later how to estimate the root position after we infer the unrooted tree. A moment of thought will show that there are three possible arrangements of the leaf labels on the unrooted 4-leaf tree as shown below:



—

There are three 4-leaf, leaf-labeled, unrooted tree graphs in the figure; we will call the three trees: T1, T2, and T3. These trees represent all the possible ways that four named leaves can be arranged into a binary tree graph; therefore, they also represent all the possible evolutionary tree relationships of four objects (excluding the root). As can be seen from the figure, T1 and T2 are related to each other by a swap of the leaf labels B and C; T1 and T3 involve a swap of B and D; and T2 and T3 involve a swap of C and D.

It is instructive for future discussions to consider another way to enumerate these trees. Consider the original unlabeled tree in the previous figure. This is the unique binary tree with four leaves. If we have four labels (names), there are 4! = 24 ways of attaching the names to the leaves. But, some of these are tree-topologically equivalent; for example, in tree T1, if we swap A and B, it is still the same tree. Likewise, for labels C and D. Furthermore, if we switch (A, B) group with (C, D) group along the line of symmetry around the middle of the tree, we get the same tree topology. So we start with 24 possible labelings of the leaves of the tree and 2x2x2 of them are topologically equivalent so we end up with 24/8 = 3 possible distinct leaf-labeled trees. It turns out that this is a key method of enumerating combinatorial objects. We first consider some possible permutation action (e.g., the 4! different actions of putting on four labels) and then we divide by the possible symmetries.

Going back to the trees, let's take T1 as an example and write out the path distances:

	A	B	C	D
A	0	a+b	a+c+d	a+c+e
B		0	b+c+d	b+c+e
C			0	d+e
				0

(We denote this matrix, AD1.)

If a pairwise distance matrix were generated as path distance over tree T1, it should have the form of AD1. What does this mean? Take an arbitrary real-valued 4-by-4 matrix:

	A	B	C	D
A	d_{11}	d_{12}	d_{13}	d_{14}
B	d_{21}	d_{22}	d_{23}	d_{24}
C	d_{31}	d_{32}	d_{33}	d_{34}
D	d_{41}	d_{42}	d_{43}	d_{44}

There are 16 real-valued numbers in this matrix. If we want this to be a matrix of distances between four objects there are some conditions for the values of d_{ij} that comes from natural properties of distances. For distance matrices, we would generally expect all diagonals to be zero; i.e., $d_{ii} = 0$ for all i, naturally corresponding to the idea that $d(x,x) = 0$ for any x. We would also generally expect the off-diagonal values to be non-negative and for the matrix to be symmetric such that $d_{ij} = d_{ji}$. Additionally, we would also generally expect the distances to satisfy the triangular inequality; i.e., $d_{ij} \leq d_{ik} + d_{jk}$ for all i, j, k. (I keep saying "generally expect" because there are notions of "distances" that might not have these expectations--but that is beyond the subject of this text.) Thus, we start out with an arbitrary collection of 16 numbers and imposing a model of distance matrix reduces the possible set of 4-by-4 matrices to a subset of a particular form:

	A	B	C	D
A	*0*	*d1*	*d2*	*d3*
B	*d1*	*0*	*d4*	*d5*
C	*d2*	*d4*	*0*	*d6*
D	*d3*	*d5*	*d6*	*0*

where the numbers *d1,d2,d3,d4,d5*, and *d6* are non-negative and satisfy the triangular inequality. We can visualize this geometrically. We start out with a point in 16 dimensional space; i.e., $p \in \mathbb{R}^{16}$. We can think of this point as an arbitrary real-valued 4-by-4 matrix. The set of all possible real-valued 4-by-4 matrices comprise the whole $\mathbb{R}^{16}$. The restriction that the elements of the matrix have non-negative values reduces the possible values to the non-negative orthant subset. (An **orthant** is a piece of a Euclidean space that has the axes of zero coordinates as the boundary. For example, if we have $\mathbb{R}^2$ there are four quadrants. In $\mathbb{R}^3$, there are eight octants, and in higher dimensions we call the pieces orthants.) Then, the restriction that the matrix should be symmetrical with zero diagonals reduces it further to the non-negative orthant of $\mathbb{R}^6$ (which, in principles, is some subspace of the original $\mathbb{R}^{16}$). The additional restriction of the triangular inequality reduces it to a particular subset of the orthant that is a cone with flat faces (this is called a polytope). Finally, the restriction to an additive distance matrix reduces it further to a smaller subset; a subset of $\mathbb{R}^5$ (because there are five variables as edge weights on a 4-leaf binary tree) that satisfies the algebraic relationships implied by the equations in (Eq 4). It will also turn out that there are particular 5-dimensional subsets for each of the three possible trees that are joined together along a four dimensional "spine".

Side Note: Models

This view of different models as a restriction of certain quantities is a very important view. Typically models are specified by a set of numbers that we call the *parameters of the model*. The collection of all possible values of the parameters form what might be called the "world according to the model". Different models might be constructed by restricting the possible world to different subsets of possible numbers. For example, in the distance matrix above, we started with any 16 real-valued numbers and our additive distance model restricts the possibilities to a five dimensional subset. In many cases, various subsets of numbers corresponding to models may be described geometrically as in the tree examples. This may help us with their analyses or with intuitive understanding. Several remarks are in order:

(1) Models are almost never a singular construct. For example, the Newtonian model of inertial motion describes a family of possible rules for motions. The specific instances within the family are specified by numbers that we call parameters (e.g., acceleration and mass). That is, what we usually call a model is in fact a set of models, each a variation of a base model.
(2) Saying that model(s) are specified by a set of numbers that are the *parameters of the model* begs the question "what are parameters?" This can be a rather tricky question and I will answer this somewhat tautologically by saying it is a set of numbers that uniquely has a one-to-one relationship with the specific instances within the model family (i.e., elements in the model set).
(3) While I said "a set of numbers", parameters of a model don't have to be numbers. It could be a set of anything that has a one-to-one relationship to the model set. For example, the different binary tree graphs are parameters in a phylogenetic model for biological data.
(4) Different models (through their parameters) need not form convenient ensembles like the cases we have seen in our progression from 4-by-4 real-valued matrices to additive distance matrices (where we were able to describe each model as a subset restriction of the original model). For example, we might have a class of models whose parameters take value from a unique collection of numbers. A concrete example of such a model would be phylogenetic trees where we decide the edge weights can have only a finite set of possible values.

All of this discussion now gets us into more trouble by begging the question "what is a model?" I won't try to answer this question too carefully. For our purposes, we imagine that there is a set of empirical data D that draw from some intuitively comparable set of measurements, say genome-wide gene expression from neurons. If D is variable, we feel like the variation in D should be "explained" by some restricted set of models. The models of D form a set, M, of compact mathematical or computational descriptions, such that in our world view there is a function $m: D \to M$. We might call the function $\boldsymbol{m}$, the "modeling function" consisting of all the things a scientist might do to map each data set to each model object. For example, D might be the set of possible gene expression measurements and M might be named cell types (i.e., the names of cells like neurons, skin cells, fat cells, etc. form our model-view). Then we might have a classification method that takes as input the gene expression values and produces as output the cell type. This classification method is our modeling function.

Hmm, having gotten this far, I have to make a few more remarks:

(1) While we imagine the existence of a computable modeling function $m: D \rightarrow M$, we don't't' necessarily assume the converse. That is, the existence of a generating function $g: M \rightarrow D$ that generates empirical data from the model is not assumed. By this, I mean that most of the time we don't really think that the model generates the data in some direct sense. Below, we will learn some stochastic models that generate DNA strings as samples of the stochastic process. In those cases, the data is being generated in a model theoretic sense; we don't really assume that evolution follows these models as a physical/chemical implementation. This may be for the birds or the philosophers, but in my view, a model is a device to organize our understanding of data; it is not a direct generator of reality (at least for vast majority of constructs that we call models). Our models are inductive constructs. Deduction using a model can be useful for expanding on the logical consequences of the model, but it does not substantiate the validity of the model itself.

(2) It is clear that $m: D \rightarrow M$ should be a many-to-one function. If there is one model for each empirical observation, there is no generalization; in some sense, there is no information gained. (There is an exception to this view, if the models have a notion of organization or structure that is itself a reduction; that is, if there is a structure to the model set M. But, I won't try to expand on this.) The computational modeling procedure is to take different datasets and map them to the allowable points of the model, in a sense performing a **data reduction** and gaining information.

(3) Suppose we have two different models or classes of models, M and M'. For example, M could be the model for spontaneous mutation of nucleotide identity from A→ T with some probability p = 0.5, while M' could be a model for spontaneous mutation of nucleotide identity A→T with probability p = [0,1] (i.e., the range 0 to 1). We may have yet another model. M'', of spontaneous mutation of A→ "-" (i.e., deletion) with probability 0.5. We may consider $M \cap M'$ to consider shared instances between two model sets. Sometimes one model set might be wholly contained in the other; i.e., $M \subset M'$. In other cases, it may not make sense to consider the intersection. In general, an important process in science is to compare different models and accept or reject one set of models over another. Therefore, it is important to consider these kinds of set-theoretic relationship of models. Sometimes the intersection of model sets might be ***represented*** in their parameter space so for all three models M, M', and M'' the parameter space is ℝ and $M \cap M' = M \cap M'' = 0.5$; but the intersection $M \cap M''$ does not make sense in terms of "model space". Analysis of models and also trying to rationally choose between different models for the same dataset is much more principled when $M \cap M' \neq \phi$ and it make sense to take the intersection. It is especially best if $M \subset M'$ (or vice-versa); then we can think of one class of models as a strict restriction of another set of models, which is conceptually and technically easier. We will see below that phylogenetic tree models have the property $M \cap M' \neq \phi$ but not $M \subset M'$, which makes their analysis tricky.

(4) Many classes of models, in fact have a structure like $M \subset M' \subset M''$ ... and also can be described algebraically or geometrically. This is because we often construct models by considering a large number of possibilities—say all possible real-valued 4-by-4 matrices, and then start imposing restrictions from various principles that we think should apply. These restrictions often have the form of algebraic equalities—identifying (i.e., "considering to be equal") classes of models. For example, these are the algebraic identities for 4-by-4 real-valued matrices derived from the very natural idea that distances should be defined such that $d(x,x) = 0$, $d(x,y) = d(y,x)$:

$$d_{11} = 0, d_{22} = 0, d_{33} = 0, d_{44} = 0$$
$$d_{12} = d_{21},\ d_{13} = d_{31},\ d_{14} = d_{41},\ d_{23} = d_{32},\ d_{24} = d_{42},\ d_{34} = d_{43}$$

These linear equations "kill" a dimension of the ambient space of model points, reducing the points to a smaller dimensional subspace. The 10 independent linear equalities above take points in $\mathbb{R}^{16}$ and "kill" 10 dimensions and restrict the points to a subset of $\mathbb{R}^6$. The additive distance matrix has one more linear equation due to the 4-point condition that kills another dimension so that we get points in a subspace of $\mathbb{R}^5$. For T1, it is:

$$d_{13} + d_{24} = d_{14} + d_{23}$$

(You will see below the other two trees imply a different linear equation.) In other biological cases, we might have other ways of finding such constraints such as "molecule A always reacts with an equal number of B" implying [A] = [B]; or "total number ATP molecule is constant", etc. When the models and variant models are specified by such constraints, the resulting representations (usually in parameter space) typically have nested relationships $M \subset M'$ and form geometric subsets that makes them easier to analyze.

Whew! We are now ready to go back to distance matrices and trees.

From the previous discussion, if a distance matrix is obtained as path distances on a binary tree and the tree topology is T1, we expect a matrix of the form:

	A	B	C	D
A	*0*	*a*+*b*	*a*+*c*+*d*	*a*+*c*+*e*
B		*0*	*b*+*c*+*d*	*b*+*c*+*e*
C			*0*	*d*+*e*
				0

[AD1]

While the computed distance matrix from the 3rd example above is:

	A	B	C	D
A	0	4.3	6.1	10
B		0	3.4	6.1
C			0	4.3
D				0

[D2]

We already determined that D2 doesn't fit the 4-point constraint. The idea now is to see how different D2 is from a matrix of the form AD1. There are many ways to come up with a numerical notion of difference between two matrices. We will try distance measure between the two matrices that computes the sum of squared difference between each element of the two matrices. We get:

$$S = (a + b - 4.3)^2 + (a + c + d - 6.1)^2 + (a + c + e - 10)^2 + (b + c + d - 3.4)^2 + (b + c + e - 6.1)^2 + (d + e - 4.3)^2$$

S quantifies how different D2 is from AD1 but as a function of the variables *a, b, c, d*, and *e*. In order to give D2 a "fair chance" of fitting to AD1, we want to find values of *a, b, c, d*, and *e* that minimizes the difference between D2 and AD1. Here, we can use calculus and take the partial derivative of *S* with respect to each variable.

According to the usual theorem in calculus the extreme values of *S* are found at the simultaneous zero of the partial derivatives, yielding the following equations:

$$\begin{gathered}
a+b-4.3+a+c+d-6.1+a+c+e-10=0\\
a+b-4.3+b+c+d-3.4+b+c+e-6.1=0\\
a+c+d-6.1+a+c+e-10+b+c+d-3.4+b+c+e-6.1=0\\
a+c+d-6.1+b+c+d-3.4+d+e-4.3=0\\
a+c+e-10+b+c+e-6.1+d+e-4.3=0
\end{gathered}$$

(from taking each partial derivative and ignoring the common factor 2). So we have:

$$\begin{pmatrix}3&1&2&1&1\\1&3&2&1&1\\2&2&4&2&2\\1&1&2&3&1\\1&1&2&1&3\end{pmatrix}\begin{pmatrix}a\\b\\c\\d\\e\end{pmatrix}=\begin{pmatrix}20.4\\13.8\\25.6\\13.8\\20.4\end{pmatrix}$$

This system has solutions (3.8, 0.5, 2.1, 0.5, 3.8) and plugging back into *S(a,b,c,d,e)*, we get $\hat{S}=0.27$, which is a very good fit (how do we know this?). (I also assume that you remember that the zeros of the derivative is not the only thing to check when finding function maxima/minima.) We are not done yet since this was only for T1. We need to consider similar additive distance matrices for T2 and T3. We can get the matrix for T2 by switching the labels B and C and appropriately rearranging (see the T1, T2, and T3 figure above):

	A	C	B	D
A	*0*	*a+b*	*a+c+d*	*a+c+e*
C		*0*	*b+c+d*	*b+c+e*
B			*0*	*d+e*
D				*0*

	A	B	C	D
A	*0*	*a+c+d*	*a+b*	*a+c+e*
B		*0*	*b+c+d*	*b+c+e*
C			*0*	*d+e*
D				*0*

Same thing for T3:

	A	D	C	B
A	*0*	*a+b*	*a+c+d*	*a+c+e*
D		*0*	*b+c+d*	*b+c+e*
C			*0*	*d+e*
B				*0*

	A	B	C	D
A	*0*	*a+c+e*	*a+c+d*	*a+b*
B		*0*	*d+e*	*b+c+e*
C			*0*	*b+c+d*
D				*0*

Additional linear algebra yields the solution (4.475,1.175,0.075,1.175,4.475) and $S = 4.88$ for T2 and the solution (5, 5, -1.5, 1.7, 1.7) and S=57.96 for T3. Wait, for this last solution, -1.5 is a negative number, which shouldn't be possible for "non-negative" edge weights. Indeed, we want to find the minimal solution within the non-negative orthant of the variables *a, b, c, d*, and *e*. Thus, we need to solve the constrained minimization problem (constrained to be not negative) not the unconstrained problem. We will see later that finding a computational solution to optimization problems becomes harder the more complex constraints we put on the problem. In fact, solving a constrained version of what used to be an easy optimization problem might just be the biggest headache in modeling. [As a quick example, for the quadratic function S, with no constraints, it is easy application of calculus techniques to find the minimum. Now suppose we constrained the problem by saying we only want solutions such that at least one of the five variables must be zero (geometrically, we are restricting the possible solutions to the "faces" of $\mathbb{R}^5$. Then we need to now compute five separate optimizations, a much more involved problem.]

Fortunately for this problem, the unconstrained minimum at the coordinates (5, 5, -1.5, 1.7, 1.7) and $S = 57.96$ is already way bigger than the solution we found for T1 (S= 0.27) so we don't have to try to find the constrained non-negative solution---because we also know that the constrained optimum will always be bigger than the unconstrained optimum. Recall from the moving particle model above for the D2 distance matrix that the original edge weights for *a, b, c, d*, and *e* were (3, 0.6, 2, 0.6, 3). The edge weights that most closely fit D2 result from using the T1 tree (S =0.27) and yields the estimates (3.8, 0.5, 2.1, 0.5, 3.8). Thus, the fit of the edge weights is pretty good, compared to what we saw with UPGMA. In fact, if we were to find the closest additive matrix for D0 we get the solution (0.7,0.7,2.15,0.7,0.7) for the edge weights. The true edge weights from our particle movement were (1, 1, 2, 1, 1), so again we get quite good estimate of the edge weights (and the tree topology).

The minimum squared difference additive distance method for phylogenetic tree:

1. Input n-by-n pairwise distance matrix
2. For each possible n-leaf binary tree graph consider the edge weights as variables
3. For each tree, solve for the optimal edge weights whose implied path distance fits the observed n-by-n distance matrix where optimality criterion is sum of squares of the observed and expected distances
4. Choose the tree graph that has the lowest optimality value

Let's summarize what we have done before making a few more remarks about modeling. We have an input given as a distance matrix between four objects. The distance matrix is assumed to have been measured from the objects and satisfy the standard notions of distance like $d(x,x) = 0$, $d(x,y) = d(y,x) \geq 0$. Our model class involves three discrete structures, the three possible leaf-labeled unrooted binary trees and five non-negative real-valued edge weights. We can think of this as one discrete-valued parameter (the tree topologies) and five real-valued parameters. If we had more objects, we will have more possible tree topologies and more edge weight parameters. We want to associate the input distance matrix to the model that best fits the input matrix. Therefore, we have to search through the parameter set (including the tree topology parameter) until we find the best fitting model. This requires us to have a notion of "best fit" and also a computational method for searching through the parameter set. In the above, we defined the fit as the sum of squared difference of each element of the distance matrix. The search through the parameter set for the five edge weights was made relatively easy by calculus and linear algebra. (Most of the time, for problems like this, we won't be able to use such direct techniques and we would be using a set of techniques called "numerical optimization", which we will discuss later.) The search through the tree topology parameter is much harder since we don't have neat tools like calculus. Since we only have three possible values of this parameter, we employed a brute force enumeration as a strategy. Later on we will see that once the number of leaves become moderately large (say > 20), we cannot employ brute force enumeration. In sum the mathematical structure of the problem is like this:

$$argmin(\varphi(D,\vec{\theta},\Sigma))$$

where *argmin* means the "find the arguments to the function that minimizes", φ is the objective function that is measuring the fit of the model to the data, and D is the data. The model has two types of parameters: $\vec{\theta}$ a vector of real or integer numbers and Σ a set of discrete structures like the tree topology. Many problems in computational biology have this structure. For example:

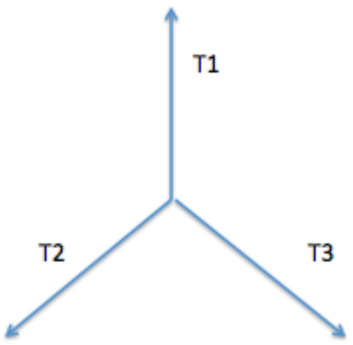
Affine gap score sequence alignment: D = set of strings, $\vec{\theta}$ = gap scores, Σ = alignments, φ = affine gap penalty function.

Protein folding: D = string of amino-acids, $\vec{\theta}$ = numbers depicting potential forces, Σ = 3D configuration coordinates, φ = free energy function.

Argmin may involve one or more of the arguments to the objective function. In the case of additive trees, we want argmin for both $\vec{\theta}$ and Σ, while for affine alignment we want argmin for only Σ. For the parameter set $\vec{\theta}$, we have all the tools of calculus, analysis, etc, that mathematicians and computer scientists have developed over 300 years. But, the set Σ is a problem. Sometimes there are ways to organize the set such that certain degree of efficiency can be gained. An example was seen in the dynamic programming method for the sequence alignment problem. In most other cases, we do not know any good approach. The key seems to be to try to organize the set in some regular way that helps create a smooth map to the objective function φ. In the next few sections, we will be discussing various functions φ for tree inference such as maximum likelihood function. Whatever the objective function, the various methods all share the problem of finding a good way to search through possible tree topologies. Now, some remarks:

(1) The parameters of the additive distance matrix can be seen as a vector in six-dimensional vector space with real-valued coordinates. We can give the coordinates a standard order like: $x_1 = d_{12}$, $x_2 = d_{13}$, $x_3 = d_{14}$, $x_4 = d_{23}$, $x_5 = d_{24}$, $x_6 = d_{34}$. Then, the objective function described above is the squared Euclidean norm of the difference between the input distance matrix vector and the parameter vector: $\|\vec{x} - \vec{d}\|^2$. That is, it is the squared Euclidean distance between the two vector points. (Recall a norm is a real-valued function of vectors that defines the length of the vector.)

(2) A more sophisticated view of parameter space for the three discrete trees is as discrete subspaces of $\mathbb{R}^6$, where each of the three subspaces satisfy one of the following three equations:



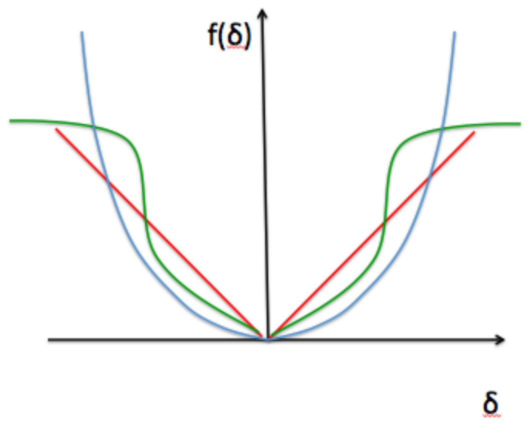
$$d_{13} + d_{24} = d_{14} + d_{23}$$
$$d_{12} + d_{34} = d_{14} + d_{23}$$
$$d_{13} + d_{24} = d_{12} + d_{34}$$

or, in terms of the coordinates we discussed above in point (1), the equations are:

$$x_2 + x_5 = x_3 + x_4$$
$$x_1 + x_6 = x_3 + x_4$$
$$x_2 + x_5 = x_1 + x_6 \qquad \text{(Eq 4)}$$

We can't visualize $\mathbb{R}^6$ but if we think about the set of points that satisfy each of the three constraint equations in (Eq 4), they might look something like the figure on the left. Here, each of the lines with arrow heads depict the subspace of $\mathbb{R}^6$ occupied by the additive distance matrices of tree topology T1, T2, and T3, respectively. The arrows actually hide a five dimensional subspace; that is, each line is actually five dimensions. The lines meet at a single point corresponding to what would happen if the middle edge of the tree graphs had weight zero—all three tree graphs would produce the same additive distance matrix. (In actuality, this point is four dimensional.) This view is interesting because on one hand, the tree topologies comprise a discrete set, seemingly distinct from the real-valued edge weight parameters. But, on the other hand, they can be seen as a particular subset in a larger, global space; the $\mathbb{R}^6$ space of distance matrices. That is, the trees can be given a common embedding in a standard vector space. We are not exactly certain if this kind of construction/view will give us better ways to efficiently compute the objective function but this kind of geometric view is very useful for understanding the properties of algorithms. We will discuss such geometric view again when we discuss probability models for phylogenetic trees.

(3) Going back to the formula $\|\vec{x} - \vec{d}\|^2$, we might ask why this particular objective function? It is clear that we want to compute some number that captures the idea of difference between the input distance matrix (now written as $\vec{d}$) and the model additive distance matrix (now written as $\vec{x}$). First, we note that even as distances between two vectors there are many variations. Here is a family called p-norms for vectors: $\|\vec{x}\|_p = \left(\sum x_i^p\right)^{1/p}$. The standard Euclidean norm corresponds to $p = 2$; sometimes this is also called L^2-norm. Some of the more commonly used norms include $p = 1$ or L^1-norm, which is the sum of the absolute value of the coordinates; and, the L^∞-norm, which is the maximum of the absolute value of the coordinates. (It should be clear that if you have a norm of a vector, the definition of the norm can be used to quantify the difference of two vectors by taking the norm of the difference vector. Therefore, a vector norm also defines a difference measure between two vectors.) Mathematically these different norms are different ways of establishing a notion of length in vector space, thereby generating different kinds of geometrical properties. However, for us, we are not so much concerned about the mathematical structure. Our question is why is it that we have $\vec{d} \neq \vec{x}$? If the model is "correct" there should be some parameter vector $\vec{x}$ at which $\vec{d} = \vec{x}$. Well, the model might not be appropriate—but we leave that to the side for now. Assuming that the model is appropriate for the data then our only rationale for $\vec{d} \neq \vec{x}$ is that there must be some source of mis-measurement that makes $\vec{d}$ deviate from its proper vector. It could be some kind of bad instrument, blurry eyes, stochastic sampling noise, etc. Since it is hard to model instrument problems or blurry eyes, we mostly concentrate on the idea of stochastic sampling noise. In the sections on probability models for phylogeny we will define the stochastic noise and resulting random events very precisely. For the moment, we just assume that $\vec{d} = \vec{x}^* + \vec{\delta}$ where $\vec{x}^*$ is a model vector that satisfies the additive distance conditions and $\vec{\delta}$ is a noise vector, gotten who knows where. How we should set up the objective function depends on what we think about the noise vector, especially how its coordinate components are distributed. Consider one coordinate of the noise vector and how different ways of computing the norm or the objective function might translate the magnitude of the noise to the magnitude of the objective function. The figure on the left shows various ways in which a noise value δ might map to values of the objective function as a "cost" component, f(δ). Three different curves are shown, linear (red), quadratic (blue), and "robust" (green).



The linear function, implemented in L^1-norm, gives proportional penalty to larger deviations, while the quadratic function, implemented in the L^2-norm, gives greater penalty to larger deviations. In a certain sense, the quadratic function is suggesting that large deviations are relatively rare compared to small deviations and therefore, if they happen, we should try to pay more attention to them. That is, when we are fiddling with the objective function, reducing large deviations will have quadratic benefits. Therefore, all other things being equal, a quadratic cost function will force the optimization to reduce large deviations more than small deviations. The green curve, which I called "robust", implements the idea that we should emphasize the large deviations—up to a point. There is a whole subfield called robust statistics that utilize these kinds of complex cost functions. Which of these and other infinitely many possible models should we use for the objective function? Similar question arise in tons of algorithm design problems. Sometimes, we can suggest a probabilistic model for the deviates—say a random deviate of Gaussian form (e.g., the normal distribution), which then suggest L^2-norm as best objective function (model-theoretic sense). But usually, there is no principled answer to this question. For phylogenetic tree estimations, it is usually the small edge weights that are difficult to correctly estimate so one could make the case that we should use something like L^1-norm. A commonly used scheme is to express the coordinate component distance as a percentage of the input distances: $\sum(d_i - x_i)^2/d_i$. This seems to be reasonable idea that approximately puts equal weight to deviations of the same percent magnitude. We can propose many such variations. Sometimes any given proposal might seem to make sense from some biological intuition. Other times, we don't really know what property the proposed methods have and it becomes a research problem in of itself to try to understand the behavior of the algorithms. In a more general view, these algorithms are trying to solve an inductive problem; i.e., estimating or inferring a model for empirical data. It is hard to justify any method from deductive principles. Therefore, in the statistical literature we concentrate more on understanding the *post hoc* properties of the algorithms such as its bias, variance, accuracy, efficiency, consistency, etc. We will discuss the meaning of these properties in later sections.

Unit 11: Maximum Parsimony Estimate and Tree Optimizations

In the previous section, we explored the "additive distance" method for phylogenetic tree inference. As we saw from a few examples, the method had a sound principle and worked quite well. So, why consider other approaches? When we discussed the string search problem, we learned many different approaches that differed in time and space efficiency, especially as a function of the particular use cases. The appropriate algorithm depended on the size of the problem, whether we were going to carry out a single search or a multiple search, available memory size, amount of mismatches, etc. For phylogenetic estimation, a similar but more complex consideration drives us to try different approaches. Most of the methods for phylogenetic estimation have a similar degree of computational complexity but have quite different inference properties. We saw the example in Unit 8 of the case of the distance matrix D1. While we didn't work out the computation, if you try it as an exercise, you will see that the additive distance matrix method will not give you the right tree topology. What this example shows is that different types of evolutionary processes (which in the case example involves the mode of the movement of participles in 2D) can have a profound effect on the accuracy of the estimate.

The goal in phylogenetic estimation is to try to recover as accurate a representation of the evolutionary history as we can. There are some computational complexity considerations as we will see later, but the focus is on the inductive inferential utility of the method rather than computational efficiency. Input data sets are empirical measurements and we want our method to do well on all possible input. But, it is very difficult to design a method that will do well on all possible input. For example, two different organisms may experience parallel or convergent evolution, at least for some characters. This is similar to the situation with matrix D1 (Unit 8) and our inference methods are likely to do poorly. In many cases, we first have problems with inferring correct homologies, including positional homologies for DNA or protein strings. Suppose an Oracle guarantees correct positional homology. Even then, we can imagine a devious Evolutionary Gremlin that scrambles the nucleotide identity in ways that completely erases history or confounds the data to positively mislead some or even all inference methods.

Fortunately, nature is not that capricious and we generally see a reasonable degree of regularity in the input data—at least if we are sufficiently judicious in the kinds of measurements we use. Nevertheless, it is also the case that we cannot predict all the possible evolutionary mechanisms and processes that govern the inputs to our phylogeny algorithms. So, what to do? If we can't even define the set of possible input, we can't evaluate the methods in terms of how they might process the input. The way around this is to frame the possible inputs in a model-theoretic framework. The model might be conceptual (e.g., an enumeration of allowable evolutionary events) or quantitative (e.g., a stochastic model of molecular sequence evolution). By necessity all these models are toy models in the sense that they are not built up from detailed atomic level physics. The hope is that the models qualitatively capture the relevant features (observables) of the data. Within such a theoretic framework, we can evaluate the methods for their performance properties. We might make the theoretical framework for the data more detailed or less detailed; and, the methods might try to incorporate more or less of the details of the models as part of their algorithmic principles. Ideally, the models, within specific conditions, should be in the same universality class as the actual empirical process such that the qualitative features converge.

What I am trying to say is that the phylogeny estimation problem is to infer the history of natural objects. There is no way to empirically guarantee correctness or even to really evaluate the performance of the algorithms. The only exceptions I can think of are experimental evolution experiments where researchers measure real-time evolution of fast evolving organisms (e.g., virus or bacteria) in the lab. However, we can adopt a general theoretical framework for the problem—the framework might use a probabilistic model of evolution or some other conceptual model. Within such frameworks, we can evaluate phylogeny estimation methods, model properties, the interaction of methods and models, etc. This is a good approach and it gives us a principled guide to what we should do in handling empirical data. We will pursue this research program here.

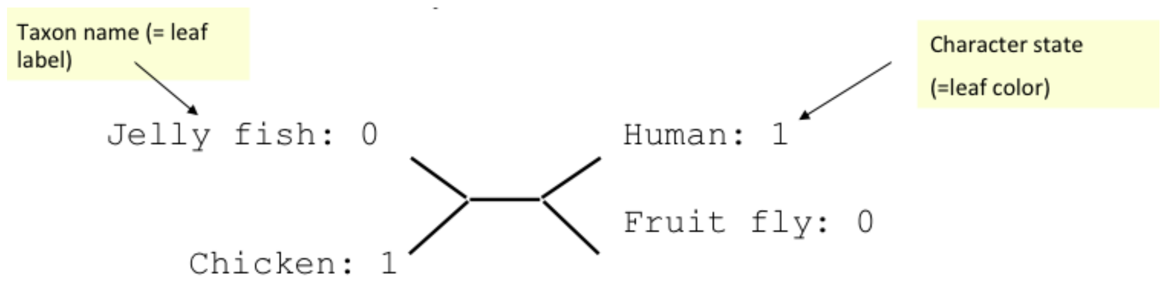
Okay, with that out of the way, it suffices to say we have many different phylogeny estimation methods because they and perform well or poorly for different examples of model theoretic inputs, each in a different manner.

Going back to the problematic example for D1 distance matrix, we can analyze the problem like this. In all of the examples of the particle split models, after the second split, the movement in the Y direction is convergent for pairs of particles, (A, C) and (B, D). The movement does tend to provide separation of A and B (C and D) but ultimately the direction of movement is convergent for A and C. Given enough time, with differential velocity, the convergent movement for A and C is greater than the divergent movements, resulting in a configuration that tends to place A and C together rather than A and B. If we were to ignore the Y axis data, we would obtain the correct tree topology—although we would be pretty severely wrong for the edge weights. However, this is an interesting idea—the coordinates of our toy model is like characters of a dataset; if we can select the right set of characters, something that reveals special tree-like relationship between the objects, we may be able to more efficiently infer the tree.

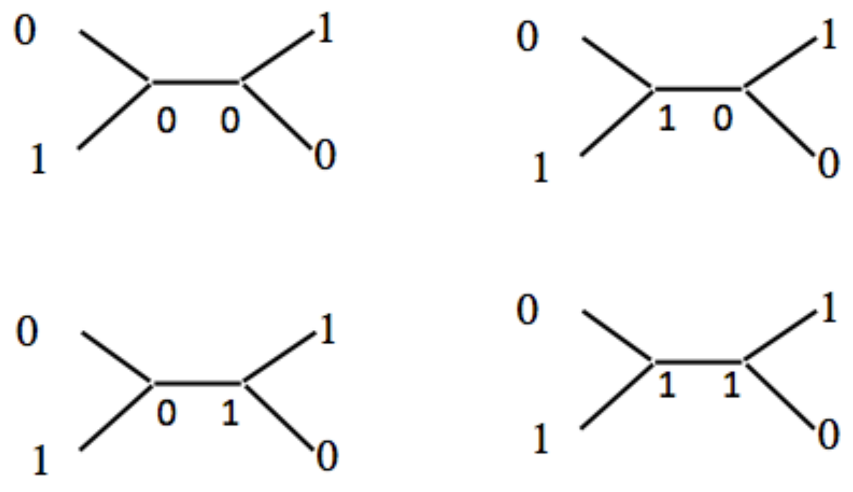
What kind of characters do we want? We want characters that identify clades or splits (cf. Unit 8). For example, consider the trait "presence/absence of backbone." If all organisms that have backbones had a single evolutionary origin, then the character identifies one clade (or, one split for unrooted trees). If we have several such characters, we can identify all the clades of a tree. Since there is a one-to-one relationship between the set of clades (set of splits) and a tree graph, we should be able to trivially estimate the tree by putting all the clades together. More precisely, suppose we have *k*-state characters. Then the distribution of the states across the leaves of the tree implies a multi-way split of the leaf labels. For example, if we have the leaves (human, chicken, mouse, worm, fruit fly, jelly fish, bacteria) and the nucleotide position with the following base distribution (A, C, A, C, T, T, T), it would imply the leaf label subsets {human, mouse}, {chicken, worm}, and {fruit fly, jelly fish, bacteria}. In this example, we are grouping together all leaves that share a character state under the assumption that they share the state because their common ancestor had that state. So a *k*-state character implies *k* clades or *k* splits. If we have *t* total leaves, then we need *t*-3 splits to define a unique unrooted binary tree (because we need to identify each one of the 2*t*-3 edges, but the edges pendant to the leaves are shared by all tree graphs). Assuming we have enough characters that implies at least *t*-3 compatible splits, we can assemble the tree in O(*t*) time by simply placing an edge between each split's subset of labels. Suppose the splits or the clades are not compatible? Similar to the Longest Common Subsequence problem we discussed in string alignment, one simple idea is to find the maximal number of compatible splits (clades) and then assemble the tree that corresponds to those compatible splits. This is called the **maximum compatibility method** or some times called the **maximum clique method**. One possible approach is to delete k characters. Then for the remaining characters, we can check for compatible splits. Since there are $C(t,k) = O(t^k)$ possible ways of deleting k characters and once we do that, we can check for compatibility in $O(t\text{-}k)$ time, for a fixed value of k, we have a polynomial time complexity algorithm. Unfortunately, we do not know the proper value of k ahead of time so we would have to try $k = 0$, $k = 1$, $k = 2$, …until we narrow it to the compatible set of remaining characters. In the worst case, this would take exponential time as a function of t (because $\sum_{k=0}^{t} C(t,k) = 2^t$). This kind of problem is called "**<u>fixed parameter tractable</u>**", meaning if we fix some parameter value (e.g., the parameter k), for that fixed value the problem has polynomial time complexity. Typically, we don't know what value to fix the parameter and therefore the problem is exponential in practice and NP-hard in theory (for many cases).

When all the characters imply compatible split/clades and we have enough of them to assemble into a binary tree, we say have a **perfect phylogeny** and the collection of the characters are called **perfect characters**. The characters for perfect phylogeny should be such that each of their possible states should have originated once in the history of the leaves. The standard biological argument is that if some character is very complex, such as a backbone, it is highly unlikely evolution would have led to multiple parallel innovations of such a complex trait—thus, such characters are good candidates for yielding perfect phylogenies. At the molecular level, insertion of small substrings of DNA into a genomic background has been considered potential perfect character because it is hard to imagine two independent insertions into a large genome occurring at exactly the same place (although molecular mechanisms for such precise insertion at the same locus in different genomes is possible, so the insertion type has be of the right kind). The idea of maximum compatibility is to allow for possible misidentification of such perfect characters. For example, a camera eye, the type of eyes that humans have, has an intricate and complex structure, but has been derived in evolution at least twice (squids and octopus, for example). Therefore, the presence/absence of a camera eye would seem at first glance like a candidate perfect character but we would be mistaken. Unfortunately, it is not easy to select such candidates for perfect characters based on a priori considerations—especially for distantly related organisms. In fact, these days most of the data we use for phylogeny estimation is biomolecular sequences like DNA or proteins. Given what we know about the mode of mutations and evolution of such characters, except for some special situations, such characters are not likely to yield perfect phylogenies. Therefore, we need a different strategy to extract a phylogeny from a set of characters that are very likely to be far from perfect.

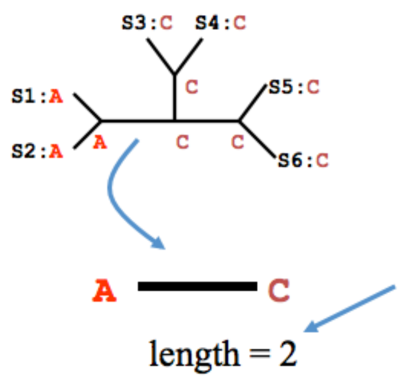
We consider a new idea by approaching the problem from the opposite point of view. Suppose we have a character like the presence and absence of backbones. If we code this as a binary character and consider the organisms (human, chicken, jelly fish, fruit fly), we will have the character states (1, 1, 0, 0). Now a tree graph can be considered a hypothesis on the number of evolutionary transitions of the character states. That is, if we propose a tree graph and look at the distribution of character states at the leaves of the tree, we can generate a hypothesis about the transitions of the character states. For example, the tree ((human, chicken), (jelly fish, fruit fly)) might be thought of as a hypothesis implying a single transition of state 1 from state 0; or, equivalently a single transition from 0 to 1 within the edge that separates (human, chicken) from (jelly fish, fruit fly). Of course, it is still possible that the common ancestor to human and chicken did not have a backbone and each evolutionary lineage independently acquired backbones—but such a scenario would be *<u>less parsimonious</u>* than the single transition scenario. To be more precise, suppose we propose the tree graph: ((human, fruit fly), (chicken, jelly fish)) shown below.



(Remember, you are supposed to distinguish between leaf labels and character states. As shown below, I will often just annotate the tree with states of some character when the leaf label context is clear.) The tree above shows one character, "backbones", with two states whose state distribution at the leaves is known. We now need to consider all the possible state distributions at the ancestors to the leaves: the two interior vertices in the unrooted tree. Obviously, there are $2^2 = 4$ possible distributions of the states. In general, if we have *k*-state characters and *t*-leaf unrooted trees, we have $k^{(t-2)}$ possible evolutionary scenarios for the state distribution of the ancestors. For the above tree, we have the following four possibilities:



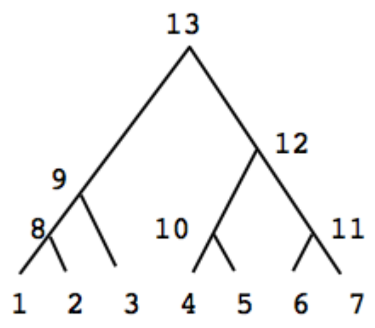
We can examine the implied transitions of the character states on each edge of the tree. You can see that the distribution of the ancestral states implies either 2 or 3 transition events across the edges of a given tree. We would claim that the scenario with 2 transitions is a more parsimonious scenario and to be preferred. We can generalize this idea with the usual construct of a cost matrix. Suppose we have an edge of a graph. Let the vertices at the end of the edge be denoted *X* and *Y* and let x_i and y_i be the states of the *i*th character for the vertices *X* and *Y*. Then the cost of the *i*th character of *X* and *Y* is given by $c(x_i, y_i)$, where *c()* is the cost matrix. We will call the value of $c(x_i, y_i)$ the "length" of the edge for the *i*th character—following the original terminology from biological systematics. If we have a full tree and a character with character states assigned for the interior (ancestral) vertices, we can compute the length of the character for the tree and its associated ancestral state distribution as the sum of the edge lengths. The tree and the cost matrix in the picture below will have the length 2 (all edges except one has length 0).



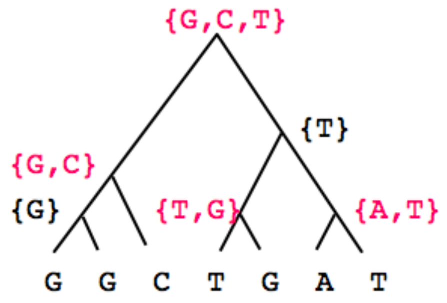
c(x,y)	A	C	G	T
A	0	2	1	2
C	2	0	1	1
G	1	1	0	1
T	2	1	1	0

The tree length just computed was for a particular set of ancestral state distribution. As noted before, we need to consider all $k^{(t-2)}$ possible ancestral state distribution. Of those, we prefer the shortest one. Why? As mentioned, one idea is that the ancestral state distribution that yields the shortest length is the ancestral state combination that generates the most parsimonious hypothesis about the evolutionary transitions of the character at hand. We will see later that we can also provide a probabilistic rationale for this optimization choice. Regardless, we call the length value obtained by minimizing over the ancestral state distribution, the parsimony length of the character over the chosen tree graph. This last clause is important. The parsimony length of a character is a function of both the character state distribution over the leaves and the ancestral vertices of a tree graph. Now we need an algorithm to efficiently find this minimal length. We have a configuration space of $k^{(t-2)}$ possible ancestral state distributions (by "**state distributions**", I mean a particular assignment of states to the vertices of the tree). Fortunately, we have an algorithm that can compute the minimal length in *O(t)* time, where *t* is the number of leaves. We will demonstrate an algorithm called the **Fitch-Hartigan** algorithm for a particular cost matrix called the Wagner parsimony cost matrix.

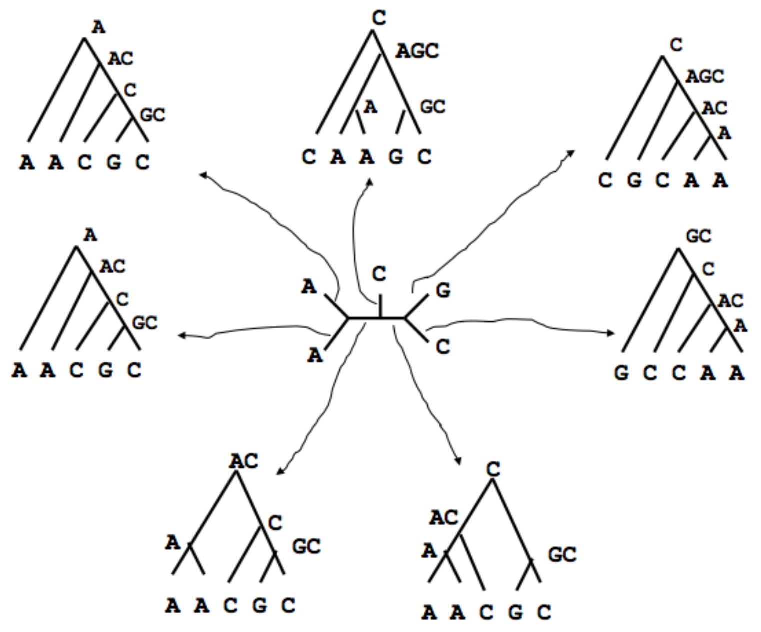
In Wagner cost matrix the diagonals are zeros and all off-diagonal elements are identical; that is, all pairwise transitions to another state have the same cost. Without loss of generality we can just say that all off-diagonal elements equal 1. Let C be a character with some state distribution for the leaves of the tree and assume we have a rooted tree. The algorithm proceeds by a depth-first search on the binary tree. Depth first search means we start at the root and move towards the leaves until you reach a leaf, then backtrack until you come back to a vertex from which you can move down the tree again, and repeat until the whole tree is visited. For the tree on the left with the internal vertices also labeled as shown, the depth-first search would take the path: 13 (root) – 9,8,1,8,2,8,9,3,9,13,12,10,4,10,5,10,12,11,6,11,7,11,12,13. One way to think of this is to imagine the tree graph as having corridors as edges. The vertices are where the corridors split into other corridors. To ensure visiting every part of the maze, we use the old maze trick and always trace along one wall.



The goal of the Fitch-Hartigan algorithm is to assign ancestral states to the ancestral vertices in such a manner that the length of the character over the tree is minimized. To do this, we need allow for the idea that a vertex may be assigned not only have a single character state but that it may be assigned a set of character states. We will consider the state sets at the leaves of the tree as sets with a single element; i.e. the known states of the character from the input data. Let L and R be the state sets at the left and right daughter vertices of a parent vertex. If $L \cap R = \phi$ then assign to the parent vertex the union of the two sets: $L \cup R$; otherwise if $L \cap R \neq \phi$ then assign $L \cap R$. Each time we assign the union, we increment the parsimony length by one. In the left figure, we show this assignment sequence for an example leaf state distribution. This tree and the character has parsimony length 4. More generally, if there is a cost matrix and we have the two daughter state sets L and R, then we search through the possible state assignments at the parent and assign all states that minimize the cost to the daughter state sets. We continue this "triangulation" up the vertices of the tree using the depth first search path, carrying out the state set assignment whenever we know the state sets of both daughter vertices. The parsimony length is the sum of the cost to each daughter state at each ancestral state assignment. The time complexity of this procedure is *O(t)*, where *t* is the number of leaves.



The algorithm above computes the parsimony length for a rooted tree. Suppose we were to unroot the tree by deleting the root vertex and then rerooting it by choosing some other edge to add the root. For the Wagner cost matrix and for any other symmetric cost matrix, it turns out that all the possible root positions result in the same parsimony length. (Try an example for the above tree.) There is no way to distinguish the scores between the different rooted trees, so we are in effect estimating the unrooted tree. The figure below shows this. In general, if there is no difference in the objective function with respect to directions on an edge, we will not be able to distinguish different rooted versions of an unrooted tree.



So far, the algorithm we have outlined computes the parsimony length for a particular tree and a particular character. The parsimony length of a dataset is computed as the sum of parsimony length for each character. Then the maximum parsimony tree is estimated by searching through all possible binary (unrooted) tree graphs, computing the parsimony length of the dataset, and choosing the tree graph with the minimal length (called the maximum parsimony tree; or MP tree). That is, we would like to solve:

$$argmin(T): \varphi = \sum_{i=1}^{n} mp(C_i, T)$$

where $mp(C_i, T)$ denotes the maximum parsimony length of character C_i for tree graph T. And, the argmin is with respect to the tree graph parameter. We stated previously that many different computational biology problems have this same structure. A few remarks are in order again:

1. For obvious reasons, if a character is invariant—i.e., has the same state for all leaves, it has zero parsimony length for all tree graphs.
2. If a character has only one state with multiplicity greater than two over the leaves then it has the same parsimony length over all tree graphs. That is, there has to be at least two leaves with one state and another two leaves with another state for this character to have different parsimony length for different tree graphs.
3. If a character has *k*-states then the obvious lower bound on the parsimony length is *k*-1 for the Wagner cost matrix.
4. The parsimony objective function is one of the oldest and most commonly used objective function. Recent advances in computational power have allowed more computationally intensive stochastic model methods to become popular.

There is a systematic way to view the maximum parsimony objective function. Suppose we have t leaves and k-state characters. Then there are k^t possible state distributions at the leaves of any tree. For example, if $t = 5$ and $k = 3$ with the states (A, B, C), we would have the following possible state distributions:

1:AAAAA
2:AAAAB
3:AAAAC
4:AAABA
5:AAABB
//
241:CCCCA
242:CCCCB
243:CCCCC

Using the indices above, we will use the notation d_1 to describe "AAAAA" and d_2 to describe "AAAAB", and so on. For $t = 5$, it turns out there are 15 possible leaf-labeled unrooted binary trees. For each of the above state distributions, say d_1, each of the 15 possible trees will give a maximum parsimony length score. For d_1, all 15 trees will give the score 0, while for d_5, some of the trees will give the score 1 while others will give the score 2. We can exhaustively enumerate these scores and denote the score for the ith state distribution and the jth tree as w_{ij}. (Imagine selecting each of the 15 possible trees in turn and using d_i as input into the Fitch-Hartigan algorithm to obtain w_{ij}.)

Given this scheme, suppose we have a dataset with n characters. Each of the characters will be one of the state distribution types; one of 243 possible types if we have 5 leaves and 3-state characters. For the dataset with n characters, we can count the frequency of each of the 243 possible distribution types. For example, suppose my aligned dataset had 5 characters like this:

	pos 1	pos 2	pos 3	pos 4	pos 5
Leaf 1	A	A	C	A	A
Leaf 2	A	A	C	A	A
Leaf 3	A	A	C	A	A

Leaf 4	A	A	C	B	B
Leaf 5	A	A	B	B	B

Then, we would have 2 of state distribution d_1, 1 of state distribution d_{242}, and 2 of state distribution d_5 while the frequency for all other state distributions would be zero. Let's denote the frequency of each of the possible state distributions for some *n*-character dataset D_n as $f_1(D_n)\ldots f_{243}(D_n)$. Obviously $\sum_{i=1}^{i=243} f_i(D_n) = n$. (In general, if we have *t* leaves and *k* states, we will have k^t indices.) The frequency spectrum of the state distribution is enough information we need from the dataset since the objective function treats each character separately; there is no information in the order of the character in the data matrix (i.e., as far as the maximum parsimony objective function is concerned). Then the maximum parsimony length of the dataset for some *j*th tree is:

$$\varphi_j(D_n) = \sum_{i=1}^{i=k^t} w_{ij} \cdot f_i\,(D_n)$$

That is, the objective function is a weighted linear sum of the frequencies of each kind of state distribution. This has some obvious consequences. For example, suppose we have two datasets D_1 and D_2. And, there is some tree graph *p*, such that $\varphi_p(D_1) \leq \varphi_j(D_1), \forall j \neq p$ and $\varphi_p(D_2) \leq \varphi_j(D_2), \forall j \neq p$. That is, the *p*th tree graph is the maximum parsimony tree for both datasets. Then we have $\varphi_p(D_1 + D_2) \leq \varphi_j(D_1 + D_2), \forall j \neq p$, where by $D_1 + D_2$ we mean add the frequency spectrum; or, equivalently, concatenate the two datasets and recompute the frequencies. In words, if we have two different datasets that have the same maximum parsimony tree then the concatenation of the two datasets has the same tree.

We can put all of this together in a geometrical picture. Given the frequency spectrum $f_1(D_n)\ldots f_{243}(D_n)$, we can rethink of this as a vector of numbers $\vec{f} = (f_1, \ldots, f_{243})$, where I dropped the D_n for notational convenience. The parsimony score for the *j*th tree can also be thought of as a vector of numbers $\vec{w_j} = (w_{1j}, \ldots, w_{243j})$. There will be 15 such $\vec{w}$ vectors for 5-leaf trees. Then the maximum parsimony length of the dataset represented by the vector $\vec{f} = (f_1, \ldots, f_{243})$ is:

$$\varphi_j = \vec{f} \cdot \vec{w_j}$$

where the dot represents the usual inner product of vector spaces. If some *j*th tree is the shortest tree for the vector $\vec{f}$, this means that this vector has the smallest inner product with $\vec{w_j}$. Furthermore, if we consider the set of possible datasets that has *j*th tree as the most parsimonious tree, this is the set $F_j = \{\vec{f} : \vec{f} \cdot \vec{w_j} \leq \vec{f} \cdot \vec{w_k}, \forall k \neq j\}$. Since this involves linear inequalities, the set F_j is obviously a convex polytope cone (recall what this is from the discussion on triangular inequalities). This geometric picture will become useful later in dissecting the properties of the maximum parsimony method.

Unit 12: Optimizing Objective Functions Over All Possible Tree Graphs

In this section we consider the problem of optimizing the objective function over the discrete parameter sets—more specifically the tree graph parameter. The problem of optimizing over the tree graphs is a shared problem for the additive distance method, the maximum parsimony method, and the probabilistic methods we will discuss later. Optimizing over a discrete set presents challenges not so much because it is discrete but because efficient optimization of a function depends critically on the following question: if we know the function value at one point, how much does it tell us about function value at other points? This in turn depends on notions of neighborhoods of a point in the domain of the function (i.e., the set of tree graphs) in relation to the values of the function. In the case of real-valued parameters, we have pretty standard notions of neighborhoods of points and a robust set of tools for understanding the structure of the objective function (i.e., how the domain of the function relates to the range of the function). With unfamiliar structures like tree graphs, we have to try out many possible ways of arranging the domain of the function and ways of searching through this structured domain. We consider four general strategies:

1. **Exhaustive enumeration:** Here we examine each and every tree graph to evaluate the objective function. In a sense, each tree graph has itself as its sole neighbor (so-called discrete topology in mathematical terminology) and no information is obtained from the evaluation of one tree graph that we can use for another tree graph. (Nevertheless, it will turn out there are systematic ways to enumerate the possible set of tree graphs whose enumeration structure can in fact give some information about the objective function value. Such a construction can be used to design what is known as a "branch-and-bound" strategy to make the exhaustive enumeration much more efficient. We will not discuss this topic in this text.)

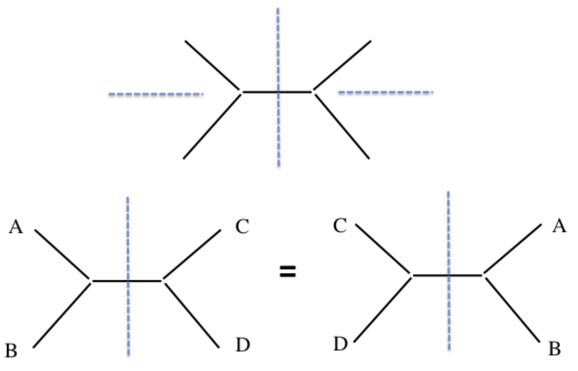
2. **Sequential construction:** As we did for UPGMA clustering algorithm, we can design strategies to build up trees from smaller trees in a locally optimal manner. We might analogize this strategy to optimizing a continuous valued multi-dimensional parameter one dimension at a time.

3. **Divide-and-Conquer:** A variation of the sequential strategy is what is called "divide-and-conquer" where a coarse set of trees with smaller number of leaves are first estimated and then refined or amalgamated into a binary tree We might analogize this to finding sets of low dimensional solutions a continuous-valued multi-dimensional parameter and refining the optimal subset.

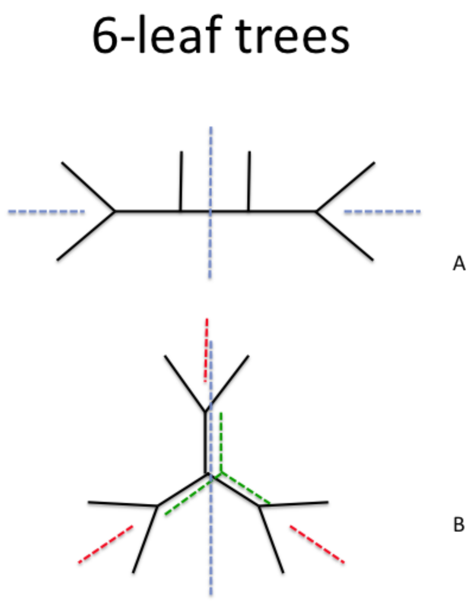
4. **Structured search over a tree graph configuration space:** Recall that a configuration space for discrete objects is a neighborhood relationship between the objects established by edit operations. We can construct various ways of transforming one tree into another tree—different kinds of edit operations will result in a different configuration space. We can then use the neighborhood information to try to efficiently search through the tree configuration space.

Exhaustive enumeration

To exhaustively enumerate over binary leaf-labeled trees we need to know what to expect--i.e., how many such trees there are, and how to systematically enumerate them. Recall that for 4-leaf unrooted trees there were three possible leaf-labeled tree topologies. A way to see this was to consider the unlabeled graph as shown below and then consider all 4! = 24 possible application of leaf labels on the tree. As seen in the figure below there are three different two-fold symmetry operations for a 4-leaf tree (shown by dotted lines) where permuting the leaf label for each of the possible symmetry results in the same leaf-labeled tree. Therefore, we have 4*3*2*1/(2*2*2) = 3 possible leaf-labeled 4-leaf unrooted trees.



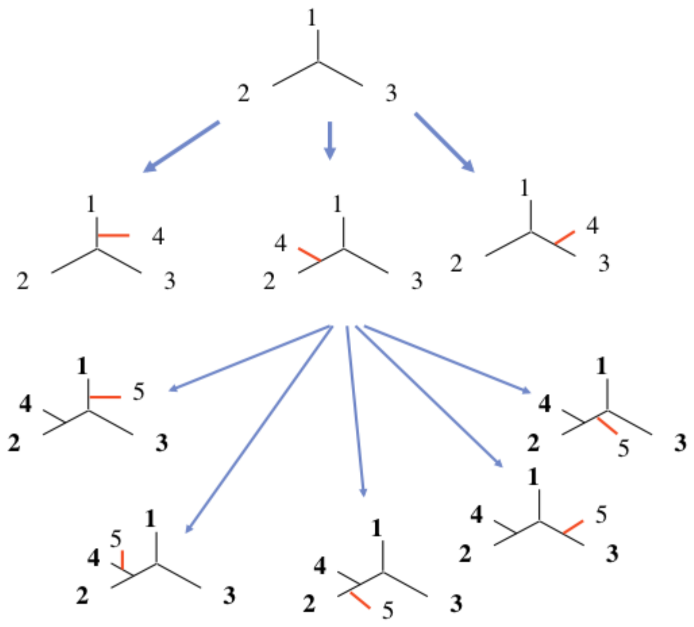
It is fun to continue this kind of counting a little bit more. We will consider 6-leaf trees. There are now two different unlabeled 6-leaf trees as shown in the figure below. Each of these trees has the symmetry operations as shown in the figure. For tree A, we have three different symmetry operations, each with two possible permutations. For tree B, we have lot more symmetry operations. We have three with two permutations (red lines), one with three permutations (green line), and another one with two permutations (blue line). Each of the trees has 6! = 6*5*4*3*2*1 = 720 possible leaf label permutations. Tree A has 2*2*2 possible label permutations that yield the same leaf-labeled tree—so we have 90 possible trees of the form A. Tree B has 2*2*2*2*3 = 48 possible permutations that yield the same leaf-labeled tree, so we have 15 possible trees of the form B. In sum we have 105 leaf-labeled 6-leaf unrooted trees.



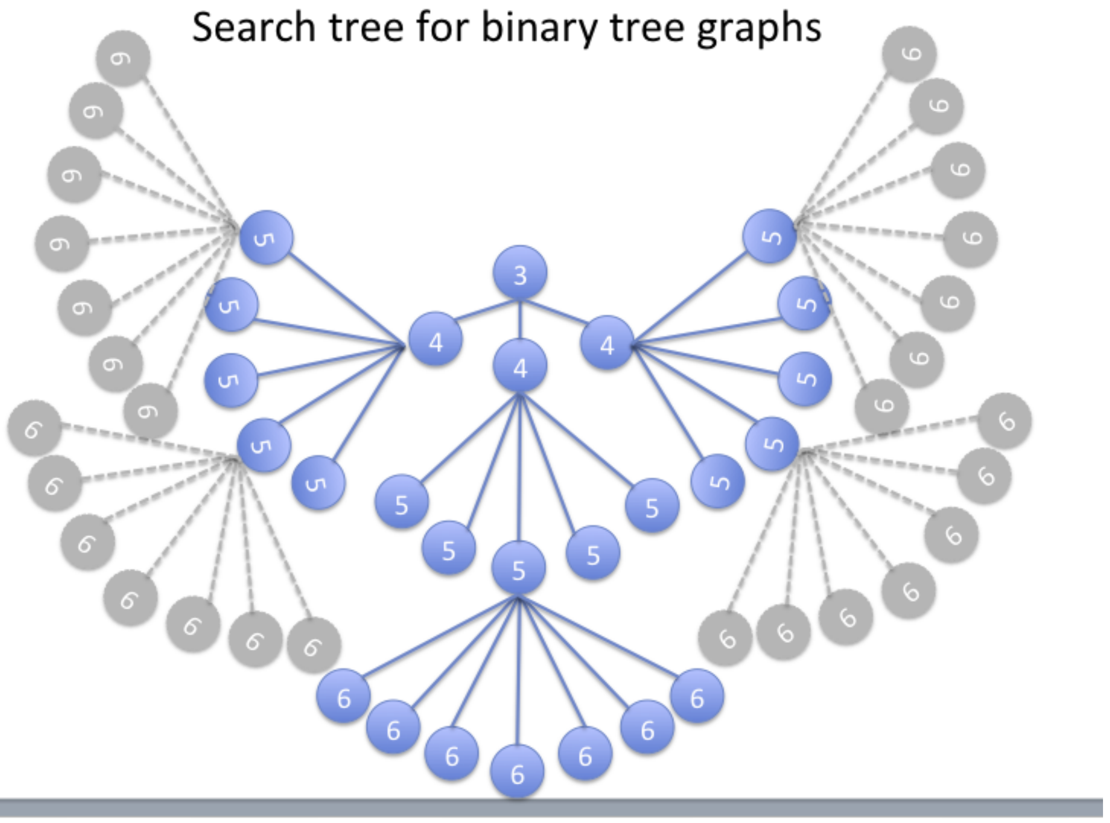
We could go on doing the kinds of counting as above but, it becomes more and more difficult—especially because it is actually a very hard problem to systematically enumerate the unlabeled tree topologies. Instead we take a different approach. Suppose we start with 3-leaf trees. It is trivial to see that there is only one possible tree topology. Now we add the fourth leaf. The 3-leaf unrooted tree has $2t$-3 = 2*3 – 3 = 3 edges to which we can add the fourth leaf. Each of the possible additions yields a unique 4-leaf tree topology (because each edge is equivalent to a unique split of the leaf labels), therefore, we have three possible leaf-labeled 4-leaf unrooted trees. Going on, there are $2t$-3 = 2*4-3 = 5 possible edges on each of the three 4-leaf trees where we can add a fifth leaf vertex. So we have 3*5 = 15 possible leaf-labeled 5-leaf unrooted trees. One more. There are $2t$-3 = 2*5-3 = 7 possible edges to add sixth leaf vertex for each of the 15 5-leaf trees so we have 15*7 = 105 leaf-labeled 6-leaf unrooted trees. This is the number we already computed above. This sequence of additions is shown the figure below. We can now see a pattern. For t-leaf trees we will have $1 \cdot 3 \cdot 5 \cdots [2(t-1)-3] = 1 \cdot 3 \cdot 5 \cdots (2t-5)$ number of leaf-labeled unrooted trees. That is, the product of odd numbers until (2t - 5). We sometimes write this as (2t – 5)!! We can use the Stirling's formula for the approximation of factorial function $n! = \sqrt{2\pi n} \cdot e^{-n} n^n$ to get a sense of these numbers:

$$1 \cdot 3 \cdot 5 \cdot\cdot (2t-5) = \frac{(2t-5)!}{2(t-2)!} = \frac{\sqrt{2\pi(2t-5)}e^{2t-5}(2t-5)^{2t-5}}{2\sqrt{2\pi(t-2)}e^{t-2}(t-2)^{t-2}} \sim O(t^t)$$

Or, the log of the number of t-leaf unrooted binary trees is $O(t \log t)$. You can do the multiplication of the odd numbers for up to say $t = 10$ and see that the number of possible trees is astronomically large. In fact, if $t > 35$ or so, the number of possible trees is greater than the estimated number of atoms in the universe. You can see why finding the optimal objective function value is so hard. Oops, that was a fallacy. If I try to find the minimum of $y = x^2$ there are infinite possible solution numbers. Hmm, so then? It is clear that the size of the possible solutions does not give a bound on the complexity of the optimization problem. I leave this up to you to ponder.



The particular method of counting we just discussed also gives us a systematic way of enumerating all possible trees. We can start from the first three leaf labels and construct a 3-leaf tree and generate the all possible 4-leaf trees; then, from each 4-leaf tree generate all possible 5-leaf tree; and, so on as shown in the figure below (in this figure, each vertex represents a leaf-labeled tree and the numbering represents the number of leaves). What you can see is that the enumeration structure is itself a tree. The 3-leaf tree is the root and it has three children nodes, each a possible 4-leaf tree. Each of the 4-leaf trees has five children and each of the 5-leaf trees has seven children and so on. This tree graph of tree graphs constitutes a search tree for systematically enumerating all *t*-leaf trees. We could do a depth-first search on this tree to enumerate all the leaf-labeled trees (actually including all the *s*-leaf trees where $s < t$)---at least for *t* not too big.



Sequential construction

The UPGMA algorithm consisted of sequentially building up the clustering tree by selecting pairs of clusters to join that are optimal at each step. Doing something optimal at each step, which may not be the optimal thing globally, is called a greedy strategy as we learned when we discussed the dynamic programming algorithm for sequence alignments. We can pursue a similar strategy with respect to the various phylogenetic tree objective functions. We can first build a 3-leaf tree, then add the fourth leaf in a stepwise optimal manner. Suppose we have a *t*-by-*n* aligned dataset *D* of *t* leaves and *n* characters. Without loss of generality we will denote the leaf labels by the set *L* = {*1, 2, …t*}. Let $S \subseteq L$ be a subset of the leaf labels. Then D_S is the aligned dataset obtained by selecting only the rows of the original dataset for the leaf labels in *S*. That is, D_S is the dataset restricted to only the leaves in *S*. Let the objective function be denoted $\varphi(D, \theta, \mathrm{T})$, where *D* is the dataset, θ is ancillary parameters (if needed), and T is the tree topology parameter. Then one possible sequential algorithm is as follows:

1. Consider all possible *C(t,3)* 3-leaf subset of *L*. Compute $\varphi(D_{\{i,j,k\}}, \theta, T_3)$ where $D_{\{i,jk\}}$ is the subset of dataset for a possible triplet of leaves and T_3 is a 3-leaf unrooted tree. Select the triplet whose function value is optimal.
2. For each of the *t -3* leaves extend the previous T_3 tree by computing $\varphi(D_{\{i,j,k,l\}}, \theta, T_4)$ where T_4 is all three possible ways of adding one vertex to the T_3 tree. Choose the leaf and the tree that gives the optimal value.
3. Continue addition of remaining leaves in the same manner.

In step 1, there are $\frac{t(t-1)(t-2)}{3!} = O(t^3)$ possible 3-leaf trees to examine. At each incremental step, suppose we have a tree of *k* leaves and *t - k* remaining vertices. Then there are *2k - 3* possible places to introduce one of the *t*

$-\,k$ remaining vertices, or $(2k-3)(t-k)$ operations. Since $k = O(t)$, this is $O(t^2)$ number of operations. We have to stepwise add vertices until we insert all t vertices so all in all we will have carried out $O(t^3)$ operations. If we assume evaluating φ takes time less than $O(t^3)$, which is generally the case for maximum parsimony, additive distance, and likelihood-based methods (next topic), then this stepwise algorithm has time complexity $O(t^3)$. The problem with this stepwise method is obvious. While at each step we make optimal decisions, this might have been too greedy and the end result might be (in fact is most likely to be) a suboptimal solution. Thus, we might think of various variations of the strategy. For example, it might be we would have a better result if we started with the "worst" triple—which in a sense forms the backbone of the tree we will build. If we start with a "long" backbone that will eventually get broken up by the added vertices, it may be we will do better. Another possibility is to alternate adding leaves with occasional removal of "worst" leaves. Of course, all of this is just a heuristic intuition and there is no guarantee any particular strategy will work better than another. On the other hand, since our goal is to optimize the objective function, we can try as many strategies as we can afford the computing time and compare the results by their objective function values. (Whether optimizing the objective function value is a good thing to do as far as the biological goal of accurately estimating the genealogical history is a different topic.)

In modern datasets the number of leaves might be very large. There are datasets with tens of thousands of aligned sequences. If the number of leaves is large an $O(t^3)$ algorithm might be already too difficult. We can construct an $O(t^2)$ time algorithm as follows:

1. For the leaf set L, determine an input order—for example, by choosing a random order.
2. Choose the first three leaves in the input order and construct a 3-leaf tree.
3. Select the next leaf in the input order and examine each of the possible edges and select the optimal edge for insertion by evaluating the objective function.
4. Continue until all leaves have been added.

(Convince yourself that the above algorithm has $O(t^2)$ time complexity.)

Divide-and-Conquer

Divide-and-conquer is a common algorithmic strategy where a given problem is broken up into smaller problems and then merged back into the larger problem. A classic example of divide-and-conquer is the sorting problem. Suppose we have an ordered set of numbers (5, 3, 6, 2, 7, 8, 1, 4) that we would like to sort from smallest to largest. We can break this up into two problems, (5, 3, 6, 2) and (7, 8, 1, 4). Each of these can be broken up further into two problems (5, 3) and (6, 2) as well as (7, 8) and (1, 4). Once we get down to pairs of numbers, we can easily sort each. So we will have (3, 5), (2, 6), (7, 8), (1, 4). The key is that we can take these smaller solutions and efficiently merge them back into the larger solution. To merge (3, 5) and (2, 6), we just have to first compare the first numbers of each. We see that 2 < 3, so we select 2 first: (2 …). Then we compare 3 with 6 and see that 3 is smaller: (2, 3…). We alternate going through each sub-solution and you can see that we only need to go through them once to merge into (2, 3, 5, 6). This class algorithm gives $O(n \log n)$ time complexity for sorting n numbers. We can approach the tree problem in a similar manner—break it up into problems of smaller size and merge the results together. Unfortunately, for these NP-hard optimization problems no known divide-and-conquer strategy results in guaranteed optimal solution. So, the current divide-and-conquer algorithms are heuristic algorithms rather than exact solutions. As in the "search over tree configuration space" to be discuss in the next section, the kind and number of divide-and-conquer strategy is too many to discuss in detail here. Constructing a fine-tuned heuristic algorithm for various objective functions for phylogeny estimation is an active research area and you should look at original research articles for details. Here, I will present just a few basic ideas.

Suppose we are trying to estimate the tree for a t-leaf dataset. And, we might somehow already know the tree for a k-leaf subset of the t-leaf labels, $k < t$. This could be from some biological prior knowledge, a different dataset, or even from first estimating a *k*-leaf smaller tree with the same dataset. We can hold the *k*-leaf tree constant and try to estimate a *t*-leaf tree that always has the k-leaf tree as a subtree. We say a tree *S* is a subtree of a tree *T*, if (1) the leaf label set of *S* is a subset of the leaf label set of *T*; (2) we remove the leaves of tree *T* except for those leaves that are also in *S*, then substitute all degree two vertices (things that look like A—X—B) with a single edge (things that look like A—B) and the resulting pruned tree of *T* has the same tree topology as *S*. (In some terminologies, we say that the tree *T* "displays" the subtree *S*.) A fixed tree that has a subset of leaves of a dataset is called a constraint tree. As mentioned, a constraint tree can come from prior biological knowledge—so using a constraint tree can be a form of integrating external information. If we have a *k*-leaf constraint tree for a *t*-leaf data set, $k < t$, then the number of possible *t*-leaf trees with the *k*-leaf fixed tree as a subtree is *(2t-5)!!/(2k-5)!!* This can be immediately seen from the consideration of the search tree of tree graphs we discussed in the exhaustive enumeration section. The fixed constraint tree must be one of the vertices in the search tree. The *t*-leaf tree must be one of the daughter vertices from the *k*-leaf tree vertex. There are *(2k-5)!!* vertices in the search tree with *k* leaves, thus we get *(2t- 5)!!* total *t*-leaf trees divided by *(2k-5)!!* If we have 10 leaves, we have 15*13*11*9*7*5*3 = 2,027,025 binary unrooted trees and if we fix a particular 5-leaf tree as a constraint tree, we reduce the possible trees to 135,135. Of course, we may have a set of constraint trees rather than a single constraint tree.

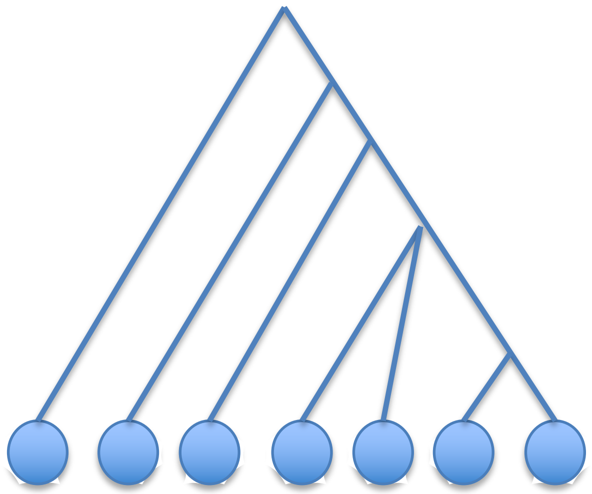
This leads us to the thought maybe we should find small trees, say multiple possible 5-leaf subsets and then glue the trees back to a larger tree. We call this problem the "super tree" problem. Given a set of input trees **P**, the supertree **Q** is a tree with leaf labels from the union of the leaf labels of **P** and where each tree in **P** is a subtree of **Q**. The problem is that when we take the same dataset then estimate smaller trees, those trees in **P** may contradict each other. What does this mean? Let *S* and *T* be two different subtrees of the leaf label set *L*. Recall the discussion on compatible clades and compatible splits. Let $S_{S\cap T}$ and $T_{S\cap T}$ be the subtrees restricted to the leaf label set that are common to both *S* and *T*. Then the two trees *S* and *T* contradict each other if the clades or the splits of $S_{S\cap T}$ and $T_{S\cap T}$ contradict each other. If the methods were perfect then there should not be any contradiction—any subset of the original dataset should correctly estimate the subtrees of the one true tree. But, real datasets and methods are not consistent like that. There will be inevitable conflicts in the various subtrees. One possible resolution of this problem is to compute some kind of notion of a **consensus tree**, like what we discussed for consensus sequence. We now need an algorithm for computing a consensus tree. Here we give one popular algorithm:

Strict consensus tree:

Let *S* and *T* be *t*-leaf binary trees that share all leaf labels in common. Let $C(S) = \{s_1, s_2 .. s_{(t-3)}\}$ be the non-trivial (i.e., non-singleton) clades of *S*, denoted as subsets of leaf labels as we discussed previously. (If *S* and *T* are unrooted we root them by the edge pendant to leaf 1 to obtain the clades, compute the strict consensus tree and then unroot the consensus tree.) Let $C(T) = \{t_1, t_2, \ldots t_{(t-3)}\}$ be the same for the *T* tree. Then the strict consensus of *S* and *T* is given by the tree implied by the clades $C(S) \cap C(T)$. That is, the consensus tree consists of the clades that are shared between the two trees. Here is an example: Let *L* = {1, 2, 3, 4, 5, 6, 7} and *S* = (1,(2,(3,(4,(5,(6,7)))))) and *T* = (1,(2,(3,((4,5),(6,7))))). Then,

C(S) = {{2, 3, 4, 5, 6, 7}, {3, 4, 5, 6, 7}, {4, 5, 6, 7}, {5, 6, 7}, {6, 7}}
C(T) = {{2, 3, 4, 5, 6, 7}, {3, 4, 5, 6, 7}, {4, 5, 6, 7}, {4, 5}, {6, 7}}

So the consensus tree has the clades $C(S) \cap C(T) = \{\{2,3,4,5,6,7\},\{3,4,5,6,7\},\{4,5,6,7\},\{6,7\}\}$. Note that this tree has less than the full set of clades—that is, a 7-leaf tree should have five non-trivial clades but we only have four. This means the tree graph corresponding to the consensus tree has ancestral vertices with more than two children. The tree is shown in the figure below. Such non-binary vertices are called polytomous (= multiple splits) vertices and such trees (and the particular vertices) are called unresolved, because they lack information about the details of some of the daughter branching sequences. The ultimate unresolved tree is the "star tree", where all leaves join together at a single ancestral vertex (shown in the figure below). All clades of this tree are trivial and it contains no information about the leaves except that they have at least one common ancestor.



In general, if we have *n* total *t*-leaf trees, $T_1, T_2, \ldots T_n$, all with the same leaf labels, we can compute the strict consensus of all the trees by considering $\bigcap_{i=1}^{n} C(T_i)$. Such trees are very likely to have low resolution. Instead of taking such strict intersections we can consider other variations that allow for partial overlap of the clades that might result in more resolved consensus trees. One commonly used algorithm is called **majority-rule consensus**:

1. Let $C(T_i)$ be the clades of *t*-leaf trees, $T_1, T_2, \ldots T_n$, all with the same leaf labels.
2. The clades of the majority-rule consensus tree $MR(T_1, T_2, \ldots T_n) = \{t_k \mid t_k \textit{ is a clade in more than 50\% of the } C(T_i)\}$.

That is, the majority rule tree is comprised of all the clades that are in greater than 50% of the trees. It needs to be proved but it turns out that if a clade is in more than 50% of the input trees, all such clades are compatible with each other. Majority-rule consensus tree can be shown to a kind of a "median" tree, in that if we look at the difference between the majority-rule consensus tree to each of the input trees, the majority-rule consensus tree is the minimal distance tree (where distance is measured by overlap of shared clades).

What if the trees do not all share the same leaf label? We can find various variations of which the simplest is to compute the consensus of subtrees that only share leaf labels. Additional leaves can be inserted into the consensus trees at the appropriate (unresolved) points corresponding to the clades in the input trees. This means, for example, we could take a large tree, break it up into smaller pieces and then compute some kind of consensus tree that are likely to have unresolved vertices. Or, alternatively we could use some kind of larger scale splitting method to create an unresolved tree with say only a few clades. This creates subproblems that are even more restricted than using constraint trees. In constraint trees we wanted to keep some kind of backbone tree constant while rest of the leaves might be inserted anywhere in the constraint tree. Here, we now have a *t*-leaf tree with some of the vertices with degrees greater than three (unresolved), but we know what leaves are attached to those polytomous vertices. The next step in the algorithm is to try to resolve the unresolved vertices. This can be a much easier problem. As an example, suppose we have the 10-leaf tree with 2,027,025 possible binary trees. Suppose we have some procedure to efficiently estimate a tree with two 5-leaf clades. So we would have a tree, ((1, 2, 3, 4, 5), (6, 7, 8, 9, 10)) where each of the 5-leaf clade is unresolved. We can now apply a more exhaustive algorithm to resolve the 5-leaf clades. There are only 15 5-leaf unrooted trees and 105 rooted 5-leaf trees. So, we only need to search through 210 trees—quite a reduction from ~ 2 million trees. Most divide-and-conquer algorithms use some kind of coarse grained procedure, say consensus trees of smaller trees, to create an unresolved *t*-leaf tree then apply a more exhaustive algorithm to resolve the unresolved vertices. This kind of strategy might be employed in a recursive manner to continuously refine the estimated tree until all non-leaf vertices are degree three vertices.

Structured search over a tree graph configuration space

The most common kind of optimization heuristic is to evaluate the objective function for a point in the domain and then examine the "neighbors" of the known point to try to "move" towards an optimum. This requires consideration of several things:

1. What are the neighbors of a point in the domain (i.e., the argument) of the objective function and what is the relationship between neighbors in the domain and neighbors in the range (i.e., value)?
2. Given the domain, the range, and the objective function map, what is the global structure of the function?
3. Given the knowledge at some point, how far and what direction to move?
4. Where to start?

Some of the above problems are easily resolved when the objective function has a real-valued domain (possibly multi-dimensional); that is, functions like $f(x) = x^2$. (Of course, we are assuming that the range is always real-valued by definition.) For a real-valued domain, we have pretty standard notions of neighborhoods and distances in the domain of the function. The main thing we are concerned about is whether the objective function is continuous (doesn't have potholes like, say tan(x)), is smooth (so we can differentiate the function), is not pathological (e.g., sin(1/x)), etc. The difficulty of problem is characterized by the number of local optima, how much of the domain of the function leads to each local optimum, etc. The optimization is easiest if the real-domain function is continuous, smooth, and unrestricted (i.e., we don't restrict the domain of the function to things like positive orthants). When this is the case, we can use a classic method like the Newton's method.

The basic idea of the Newton's method is to first approximate the original function with its Taylor series representation (look up Taylor series in a calculus textbook for more details). For a differentiable function $\varphi(x)$ a Taylor approximation at the point a is given by:

$$\varphi^*(x) = \varphi(a) + \varphi'(a)(x-a) + \frac{1}{2!}\varphi''(a)(x-a)^2 + \frac{1}{3!}\varphi'''(a)(x-a)^3 + \cdots \qquad \text{(Eq *)}$$

where $\varphi', \quad \varphi''$ means to take the first derivative, the second derivative, etc. The nice thing about Taylor approximation is that how well Eq * approximates the original function depends on the number of terms (and thus the derivatives). We can start with a linear approximation (first two terms), get a better approximation with a quadratic approximation (first three terms), etc.

[For those of you who are not used to the idea of an approximation to a function, it will be good to think a bit about what (Eq *) means. Remember that a function takes in a number as input and outputs a number. An approximation to a function is another function that takes the same input and outputs nearly the same number. Typically the approximation is much easier to compute than the original function. Most of the time the accuracy of the approximation depends on the range of input values. That is, we can approximate the function pretty nicely near some specific points, but the error in the approximation can go up further we get from those specific points. Therefore, we usually restrict approximations for a function around a particular anchor point; for example, we might be interested in approximations within a small interval around zero. The Taylor approximation in (Eq *) involves a formula that includes the anchor point a. In which case, we say that (Eq *) is a Taylor expansion of the original function around the point a. Notice that anything with a in formula (Eq *) is a number, not a variable. For example, $\varphi'(a)$ is a number that is the first derivative of φ at the point a. So if we strip out all the parts that are numbers then you can see that (Eq *) is a polynomial function in the variable x.]

The standard Newton's method approximates the function up to the quadratic approximation. We take the quadratic approximation, take the derivative with respect to x, set it to zero and find the optimal solution.

$$\varphi^*(x) = \varphi(a) + \varphi'(a)(x-a) + \frac{1}{2}\varphi''(a)(x-a)^2$$

$$\frac{d\varphi^*}{dx} = \varphi'(a) + \varphi''(a)(x-a) \qquad \text{(Eq **)}$$

$$\frac{d\varphi^*}{dx} = 0 \quad \Rightarrow \quad x = \frac{\varphi''(a)a - \varphi'(a)}{\varphi''(a)} = a - \frac{\varphi'(a)}{\varphi''(a)}$$

Okay, suppose a is our current guess for the argument value that gives the minimum to $\varphi(x)$. We use the quadratic approximation around a as shown in first line of (Eq **). This tells us that if the quadratic function is correct, the minimum will be found at the point $a - \frac{\varphi'(a)}{\varphi''(a)}$. Or, the optimal point is $-\frac{\varphi'(a)}{\varphi''(a)}$ displaced from our current guess. This is our best guess up to the quadratic approximation. We can then move our guess to this new point and then try the approximation again. Let a_0 be our initial proposed solution Newton's Method give us:

$$a_k = a_{k-1} - \frac{\varphi'(a_{k-1})}{\varphi''(a_{k-1})} \qquad \text{(Eq ***)}$$

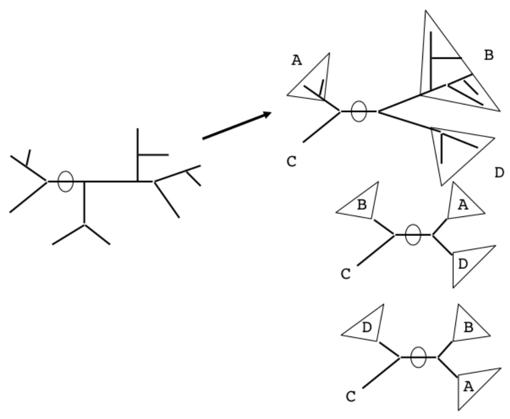
If the function is smooth and we have formulas for its derivative using (Eq ***) can give us a solution with the desired accuracy with very few function evaluations. In fact, at its best, it converges quadratically to the actual solution—by which we mean (roughly) that each iteration gains us previous accuracy squared.

Newton's method is one of many numerical optimization approaches and variations can be applied to multi-dimensional domains. Some of the other common algorithms for numerical optimization include the Simplex method, Powell method, truncated Newton method, etc. Numerical analysis and algorithms is an active field of applied mathematics research.

The main reason a method like the Newton' method works well is because the approximate function tells us how far to go and also what direction to go. (In our univariate example, we only had two directions, left or right. In the multivariate case, the method will also tell us where to go in the higher dimensions.) For tree graphs, the domain is discrete and we don't have a fixed notion of neighborhood. In particular, since we don't have a notion of derivative we also don't have a good idea how much to move and what direction (if the notion of direction makes sense). The main tool we have for this problem is the construction of the neighbor relationships by tree edit operations. We consider two types of edit operations: (1) rotating tree vertices around an edge; (2) breaking the tree and regrafting into a new tree. In general, all of the discussions will be for unrooted trees. We can always create rooted versions of the operations as an easy extension of the unrooted tree operations.

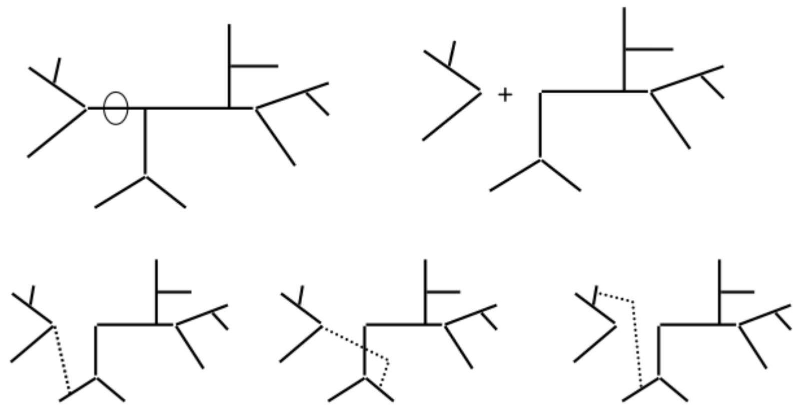
Nearest-Neighbor-Interchange (NNI) edit

The NNI edit operation is based on the fact that for any non-leaf edge, we have two internal vertices; call these X and Y. From each of the X and Y internal vertices we get two edges each leading to other vertices (i.e., other than X and Y). If we ignore what is connected to each of these edges, the overall picture looks like a 4-leaf unrooted tree. An example is shown in the figure below. In this picture, if we abstract the complex branches of the tree as a subtree (see "A" in the figure), then the whole thing looks like a 4-leaf tree. As we know, there are three possible unrooted 4-leaf trees, which involve permuting the leaves. So, we can do the same around this edge, where the subtrees are treated as if they were leaves (see the figure). For each t-leaf tree, there are *t- 3* edges that can be rotated into three possible configurations, one of which is the original. So, there will be $2(t-3) = 2t-6$ possible "neighboring trees" for each tree graph. This is one of the smallest neighbors we can make.



Tree-Bisection-and-Regrafting (TBR)

The idea here is to generate larger neighborhoods by taking a leaf-labeled unrooted tree and breaking the tree into two pieces (see figure below). Then, the two pieces can be grafted back together by "tying together" two edges. For each tree graph, there are *2t-3* edges that can be broken. Each of the two pieces have $O(t)$ edges that can be pasted together, or $O(t^2)$ possible regrafting operations. So, each tree graph will have $O(t^3)$ neighbors in this edit operation.



There are several other kinds of edit operations we can induce. For example, one edit operation is edge removal and refinement. In this operation, we select one or more edges and then contract the edges to create an unresolved vertex. Then the multiple edges of the vertex are refined back into degree three vertices.

There is a tradeoff relationship between the neighborhood size of the configuration space, computational efficiency, and the probability of finding the global optima. On one hand, the larger the neighborhood size, more likely we will find the global optima. In the extreme, every tree will be a neighbor of every other tree. Then a neighborhood search over this configuration space is the same as an exhaustive search over all possible tree topologies. We would be guaranteed to find the global optimum but the local neighborhood assessment would be very costly. On the other hand, if the neighborhood size is small, we have efficient local search but more likely to be trapped in local optimum. In principle, some kind of adaptive construction of neighborhoods might work well as a heuristic search strategy but not much has been investigated in this area. Here is a rough sketch of the neighborhood search heuristics:

1. Input: Aligned *n* by *t* dataset of *n* characters and t leaves.
2. Choose a starting binary tree (e.g., either randomly or by some sequential tree building procedure).
3. Compute objective function of the current tree
4. Examine some portion (or all) neighbors of the current tree and compute the objective function
5. Move to one (or more) of the neighboring trees if it is better or equal in score to the current tree.
6. Stop when a critical number of steps have been taken or if no more improvement is possible

7. Repeat at step #2 with a different starting tree.

As you can see, there are many points at which variations in the strategy can be introduced. We might consider many different ways of picking the initial starting tree. We can choose many different ways of defining tree neighborhoods. We might examine a very small number of neighboring tree or the exhaustive collection. We might look at one-step neighbors or *k*-step neighbors. We might dynamically change the notion of tree neighborhoods, etc. Research on this optimization suggests that effective heuristic strategy is often dependent on the input data and investigators often have to experiment with many different strategies to find better optima for any given dataset.

Unit 13: Stochastic Models of Evolution and Probabilistic Phylogenetic Estimation

We now turn to probabilistic models of evolution as the basis for estimating phylogenies. We like probability. We like probability for several important reasons. The first is that everybody intuitively understands probability. Everybody has some understanding what a batting average of 0.347 is supposed to mean. (Although maybe not quite as well as they should since people say things like: "he has a career batting average of 0.347 and he hasn't hit in 10 at bats—*he is due*.") This stands in contrast to statements like "free energy of the fold was -37kcal", which most of us don't really know what "-37kcal" means. The second is that we have rules of probabilistic inference called probability calculus that gives us an algorithmic way of compounding inferences. The most important of these probability calculus rules is the concept of **independence**. If two events **A** and **B** are independent then their joint probability can be computed as the product of the respective probabilities: P(**A** and **B**) = P(**A**)P(**B**). The main importance of this theorem is that when modeling some complicated set of events, e.g., "A and B and C and D…", if the events are independent, we only need to have a probability model for each of the marginal cases (i.e., each event considered alone). The fact that we can obtain the probability of the more complicated cases as a product of simpler cases is key to enabling modeling. If we didn't have this construction, modeling would be much more complicated and require much more data.

Of course, most realistic scenarios involve events that are not independent. For example, if we are modeling the amino-acid identity of a homologous protein from humans and mice, the probability that the second amino-acid is, say, Alanine in humans and Leucine in mice, is not independent of each other since both amino-acids are derived from a common ancestor protein. That is, the amino-acid states in each of the descendent proteins are influenced by the common ancestor. But, suppose we assume that we know the homologous amino-acid in the common ancestor, say it was Isoleucine. Then the conditional events, "Alanine in human, given the ancestor was Isoleucine" and "Leucine in mice given the ancestor was Isoleucine" can be reasonably modeled as independent events and we can write P(Alanine and Leucine|Isoleucine) = P(Alanine|Isoleucine) P(Leucine| Isoleucine). [Read P(**A**|**B**) as "probability of **A** given **B**".] It turns out that such conditional independence modeling is critical to stochastic processes and the evolutionary models discussed below.

Conditional probability and probability calculus can be used to enable the deduction of (seemingly) complicated events. As an example, here is a classic problem in probability: "Suppose I have drug test that correctly identifies drug users 99% of the time, if they are drug users (false negative 1%) and gives a false positive result 2% of the time if somebody is not a user. In general, 3% of the people in the population are drug users. If some random individual tests positive, what is the probability that he or she is in fact a drug user?" You can look at the **Probability Notes** (use Bayes' formula to compute P(drug user/positive test)) to see that, surprisingly, the probability is only ~0.6. The computational rules of probability calculus tell us how to put various probability statements together to derive other probabilities, similar to deterministic logical models where prepositional calculus helps deduce compound inferences. Lastly, most natural data have some kind of uncertainty either due to measurement uncertainties or due to factors inherent to the process. Probability is the formal framework we have developed to model such uncertainties (not without arguments about its axioms or applicability; cf., "God does not play dice"). For some of the basics of probability, please see the **Probability Notes**.

The goal of this section is to (1) propose a model for sequence evolution through time, (2) extend this model to a joint probability model of sequences over a tree graph, (3) derive a sampling distribution for how the probabilities relate to finite samples of empirical data, and then (4) construct a probability model-based estimator of the model and tree. There are three key concepts to digest first and then we will discuss what is known as the Markov model of sequence evolution.

The first key concept to understand is **sampling** from a probability model. We all know that the probability of heads in a fair coin throw is ½. But, that is just theory. To get actual data, we would have to physically throw the coin. The outcome is either head or tail—we don't get ½ head. The model theory is that in the long run, if we throw the coin billions of times, the relative frequency of heads will converge to ½ (by what is known as "**law of large numbers**"). But, the result for any **finite sample** of the model is different. In fact, the model can be used to compute the probability of possible outcomes of a finite sample. Suppose we throw the coin three times. Then the possible outcomes are "HHH", "HHT", "HTH", "HTT", "THH", "THT", "TTH", and "TTT". To be more precise and use formalism, we can say there is a **random variable** called C_3 (three coin tosses), which has values in the set {"HHH", "HHT", "HTH", "HTT", "THH", "THT", "TTH", "TTT"}. The probability of each of the possible outcome might be modeled as 1/8 each from an independence assumption of each throw. So we can compute P{C_3="HHH"} = 1/8, for example. This is the **sampling distribution** of three coin flips; or, the probability distribution of the random variable C_3. What this means is that if we were to repeat groups of three throws a billion times, we would see "HHH" about 1/8th of the time. We can also compute the probabilities of a **statistic** of a sample. A statistic is some mathematical function, usually real-valued, of the **outcomes** of a **probability trial**. For example, one statistic is the number of heads in the sample; we will call this S_H. For the eight possible outcomes listed above, we have: S_H = 3, 2, 2, 1, 2, 1, 1, 0 (in order). We can compute the probability of S_H taking the values 0, 1, 2, 3. More generally, we can compute the probability that if we throw a fair coin *n* times, assuming independent throws, the probability that the number of heads will be some value $0 \leq k \leq n$:

$$P\{S_H = k\} = C(n,k)(\tfrac{1}{2})^k(\tfrac{1}{2})^{n-k} \qquad \text{Eq 10.1}$$

which, of course, is the binomial distribution with p = ½. We can compute other statistics like the probability of at least one H in the sample, which is 7/8; and, the probability of at least one T, which is also 7/8. The sum of these two probabilities is greater than one, because they are not disjoint events (see Probability Notes). If we were to do the three-coin toss a billion times, the expected proportion of heads is:

$$3/3 \times \frac{1}{8} + 2/3 \times \frac{1}{8} + 2/3 \times \frac{1}{8} + 1/3 \times \frac{1}{8} + 2/3 \times \frac{1}{8} + 1/3 \times \frac{1}{8} + 1/3 \times \frac{1}{8} + 0/3 \times \frac{1}{8} = \frac{1}{2}$$

which corresponds to the knowledge we had about the long-term frequency of a single toss.
The point of all this is that given some probability model, the actual data are a finite sample of the possible outcomes of the probability model. Finite samples have various possible configurations dictated by the original model. The original probability model also determines a **sampling distribution** of the possible configurations of the finite samples. That is, the sampling distribution is the probability distribution of the finite sample configurations. Typically we need to know this sampling distribution in order to associate probability models with empirical data. There are many kinds of sampling processes that are important for biological models. One of them is sampling from a bounded uniform interval. Most computer programs come with a method to sample from a uniform distribution over the interval [0,1]. For example, in MS Excel, you can type the function RAND() and you will receive a number between 0 and 1. This is again a sample. We can use a sample from a uniform distribution over [0, 1] to create other kinds of samples. For example, a sample from a biased coin with

probability of heads p, can be derived from the uniform sample by setting: "outcome = heads if RAND() value is < *p*, otherwise tails." We will see many examples of more complicated derivations later.

The second key concept to know is that of **joint probability distributions**. The sampling distribution of three coin tosses requires us to consider the joint events of combination of each of the three tosses. Therefore, we are interested in the **joint distribution** of the three coins comprised of eight possible outcomes. In fact, we could say that a sampling distribution for finite samples of size *n* is the joint probability distribution of *n* samples drawn independently and identically from some base probability model (i.e., a model like a flip of a coin). The concept of "**independent and identically distribution**" is so important we usually say "***iid***". Regardless, the concept of joint distribution is broader than just an *iid* sampling distribution. A joint distribution of $\boldsymbol{X_1 \ldots X_n}$ random variables is the probability distribution defined over all the possible joint events of the *n* different random variables. The random variables may represent *i*th sample drawn independently from an identical distribution or they may represent entirely different set of events. The example calculations I gave for the coin toss assumed that each of the coin toss was *iid* and that we tossed a fair coin with probability of heads = 1/2. I used the computational rules for independent events to obtain quantities like Eq 10.1. If we weren't able to make *iid* assumptions, the probability distribution of the eight configurations, "HHH", "HHT" and so on, might be some arbitrary set of eight numbers, $p_1, p_2, \ldots p_8$, that sum to 1. This arbitrary model might represent such empirical scenarios such as coins connected by thin thread (thereby affecting each other's outcomes), unfair coins and so on. In general, we want the joint distribution of Toss-1, Toss-2, and Toss-3, meaning a probability number for each of the eight different joint configurations, regardless of whether each probability is a function of a single parameter (i.e., in the case of *iid* tosses) or seven different parameters.

The third key concept is that of a stochastic process. Formally, a stochastic process is a probability distribution that is indexed by time. That is, for each value of time, *t*, there is some probability distribution *P(t)*, that tells us the probability of each kind of outcome at that time point. The set of possible outcomes is called the **state space** of the stochastic process. We also say the stochastic system is in some state *S* at time *t*. As an example, suppose we are tossing a coin again and we compute the statistic S_H each time. We might have the coin toss sequence:

HHTTHTHTTTH

Then the value of S_H is:

1, 2, 2, 2, 3, 3, 4, 4, 4, 4, 5

Denoting each toss trial as time steps, we have the following possible values of S_H at each time step:

1: {0, 1}
2: {0, 1, 2}
3: {0, 1, 2, 3}
4: {0, 1, 2, 3, 4}
…
t: [0, t]

That is, the state space is [0, t] for this stochastic process. Assuming a fair coin with *iid* tosses we have the following probability distribution for *k* Heads at time *t*:

$$P\{S_H = k, t\} = C(t, k)\left(\frac{1}{2}\right)^k\left(\frac{1}{2}\right)^{t-k} \qquad \text{Eq 10.2}$$

As another example, consider a "random walk" on an integer line (= state space). We imagine a particle starting at position zero. Then we flip a coin whose probability of heads is *p*, and move +1 if we get heads and -1 if we get tails. In general, at time *t*, we have [-*t*, *t*] as the possible outcome positions. Suppose we had *H* heads and *T* tails with *H*+*T* = *t*. Then our position will be *H* − *T*. So, after *t* steps, the probability we will find ourselves at position $x \in [-t, t]$ can be computed by using the fact that *x* = *H* − *T* = 2*H* − *t*, so we need *(t+x)/ 2* heads and *(t-x)/2* tails, if they are both integers. If (t+x)/2 is not an integer then that particular position is impossible for that time step. For example, after one step, it is impossible to be at position zero. If |t + x| is an even number then *(t+x)/2* is an integer (|t − x| is even when |t + x| is even). Let *X(t)* be the random variable for the position at time *t*. Then

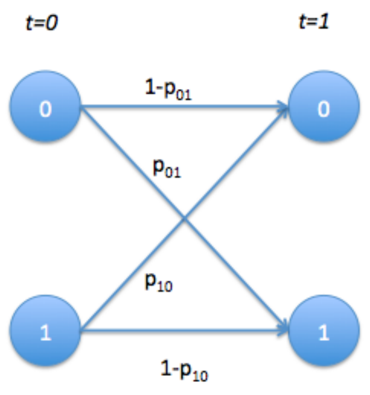
$$P\{X(t) = x\} = \begin{cases} C\left(t, \frac{t+x}{2}\right) p^{(t+x)/2}(1-p)^{(t-x)/2}, if\ |t+x|\ is\ even \\ 0, \quad if |t+x|\ is\ odd \end{cases}$$

Stochastic processes are built upon elemental probability events that over time cumulatively create probabilistic trajectories of outcomes. Often these outcomes involve numerical values, such as positions on a line discussed above. But, the state space can be a finite set of states such as 0/1 binary states or A, C, G, and T, DNA letters. Suppose we have two possible outcomes, 0 and 1. We consider a stochastic process that starts at 0 or 1 with some fixed probability $\pi(0)$ and $\pi(1)$. At the first time step, the state goes from 0 to 1 with probability p_{01} and goes from 1 to 0 with probability p_{10}. If we start at state 0, after one time step the probability that the state is 0 is $(1-p_{01})$ and the probability the state is 1 is p_{01}. A little bit more generally, let *X(t)* be a random variable that takes values in {0, 1} and *t* is the index for time. $Prob\{X(0) = 0\} = \pi(0)$ and $Prob\{X(0) = 1\} = \pi(1)$ (this is part of the model definition). After time step 1, the probability distribution of *X(1)* is:

$P\{X(1) = 0\} = (1-p_{01})\, \pi(0) + p_{10}\, \pi(1)$
$P\{X(1) = 1\} = p_{01}\, \pi(0) + (1-p_{10})\, \pi(1)$

We can breakdown $P\{X(1) = 0\}$ as these two probabilities:

[prob. X(0) = 0 and prob. of remaining in 0 starting from 0]
+
[prob. X(0) = 1 and prob of switching to 0 starting from 1] (*)



We can also draw a state switching diagram as shown below. This shows the structure of the probability model. When we are in one of the states, say 0, there is a probability of staying in the same state or switching out to the other state. These transition probabilities are shown as variables on the arrows. Since this exhausts the possible probabilistic events from the state zero, the two transition probabilities sum to one. We can interpret the transition probabilities as conditional probabilities: $P(X(1) = 1 \mid X(0) = 0) = p_{01}$, $P(X(1) = 0 \mid X(0) = 0) = 1 - p_{01}$, and so on. Or, more generally, there is a set of parameters specifying $P\{X(1) = s \mid X(0) = t\}$ for all states *s* and *t* in the finite state space. And, we have $\sum_{s \in S} P\{X(1) = s | X(0) = t\} = 1$, where *S* is the finite state space. That is, from any given state t, the probability of switching to any of the other states (including itself) sum to one—as it should. All of these transition probabilities are part of the model specification.

Now, if we want to know the probability of some state at time *t = 1*, then we sum up the probability over all the possible ways of getting to that state from the previous time point. For example, the respective probabilities of each path to state 0 are given by the terms in brackets in (*) above.

So far, we discussed the model in terms of transition from time 0 to time 1. We can then assume that the transition probabilities hold the same for all time (this is called a "time homogeneous model"). Then the main thing we want to compute is the probability distribution for *X(t)* for any value of *t*. In fact, if it is well defined, we often want to also know the probability distribution of *X(t), t* = ∞. The distribution at *t* = ∞, if it exists, is called the **equilibrium distribution** of the stochastic process. Infinity is a long time. What we really mean is that while the stochastic process has a probability distribution, *P(t)*, that is a function of time; after a sufficiently long time, the probability distribution can become invariant; i.e., *P(t) = P(t+1)* for *t* big enough. We would like to know what this is because it represents the probability distribution if we forget about the process for a while and then look in (say forget about it for a few million years of evolution, then look at it). I mentioned "if it exists". Depending on the stochastic model funny things can happen. For example, the system may oscillate between a set of probability distributions. Or, it might blow up.

We can make notations cleaner by using matrices. We can write:

$$\mathbf{P} = \begin{pmatrix} p_{11} & \cdots & p_{1k} \\ \vdots & \ddots & \vdots \\ p_{k1} & \cdots & p_{kk} \end{pmatrix}$$

where p_{ij} is interpreted as the probability of transitioning to *i*th state from *j*th state. Note that the rows of this matrix have to sum to one by definition. A square matrix whose elements are within [0, 1] and whose rows sum to one is called a **stochastic matrix**. We can also write the probability of the states at some time t, as a vector:

$$\vec{\pi}(t) = \begin{pmatrix} \pi_1(t) \\ \vdots \\ \pi_k(t) \end{pmatrix}$$

You can verify for yourself with some simple examples that we have:

$$\vec{\pi}(t+1)' = \vec{\pi}(t)' \cdot \mathbf{P} \quad \text{(Eq 10.3)}$$

where $\vec{\pi}(t)'$ means transpose of the column vector $\vec{\pi}(t)$. [Setting p_{ij} to be the transition probability from *i*th to *j*th state is just a convention. We could set it up so that p_{ij} means transition probability from *j*th state to *i*th state. In that case, we would do pre-multiplication so we get $\vec{\pi}(t+1) = \mathbf{P}\vec{\pi}(t)$, where now the vectors are column vectors and we are not taking transposes. In some texts, they use this alternative convention. As long as we know what the probabilities mean, we can keep track in a consistent manner.]

By induction,

$$\vec{\pi}(t)' = \vec{\pi}(0)' \cdot \mathbf{P}^t \qquad \text{(Eq 10.4)}$$

It is also clear that for any $0 < s < t$,

$\mathbf{P}^t = \mathbf{P}^s\mathbf{P}^{t-s}$ (Eq 10.5)

which is also known as Chapman-Kolmogorov equations. [More precisely, the Chapman-Kolmogorov equations involves "marginalization" of joint probability distributions. Here, the joint probability distribution refers to the joint of the random variable at different times; and, marginalization means to compute the joint as a product of marginal times (e.g., *s* and *t-s* times). Assertion of Eq 10.5 can be used to define the Markov property (see below).]

The stochastic process we have just studied is called a discrete time Markov Chain. The discrete time is because we have been using integers for the time index, rather than real numbers. And, a Markov Chain is a special kind of a Markov process where the process is called a chain if the state space is discrete—like {A, C, G, T}. The key part of the Markov process is its "memoryless" property where what happens in the future only depends on the current state and not any past states. That is, at some time *s*, the probability of an event in some future time *t*, only depends on the state at *s* and no time prior to *s*. Formally, a discrete time Markov process has the property:

$P\{X(t+1) \mid X(t), X(t-1),\ldots X(0)\} = P\{X(t+1) \mid X(t)\}$ (Eq 10.6)

where I abused the notation a bit (when I write $P\{X(t)\}$, I mean $P\{X(t) = x\}$).

Markov processes are kind of a stochastic analog of many deterministic models where we also specify a model using one-time step dependence between future states and current states. For example, the population size of USA at time *t*, $N(t)$, might be modeled as $N(t) = b\,N(t-1) - d\,N(t-1)$ where *b* is the birth rate and *d* is the death rate.

As mentioned, we presume that **P** is given by model assumptions and $\vec{\pi}(0)$ is also given. We want to calculate $\vec{\pi}(t)$ using (Eq 10.3). This requires us to compute $\mathbf{P}^t$, which is generally time consuming. One way is to compute using iterations of matrix squares. That is:

$$\mathbf{P1} = \mathbf{P} \cdot \mathbf{P}$$
$$\mathbf{P2} = \mathbf{P1} \cdot \mathbf{P1}$$
$$\vdots$$

In general, we would only need to do O($\log_2$ t) number of matrix multiplications. However, if **P** has non-zero determinant, a standard linear algebra theorem tell us that there exists another invertible matrix **Q** such that **QQ' = I** and **QPQ' = Δ** where **Δ** is a diagonal matrix. Then we have:

$$(\mathbf{QPQ}')^t = \boldsymbol{\Delta}^t$$
$$\rightarrow \mathbf{QP^tQ}' = \begin{pmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda^k \end{pmatrix}^t$$
$$\rightarrow \mathbf{QP^tQ}' = \begin{pmatrix} \lambda_1^t & & \\ & \ddots & \\ & & \lambda_k^t \end{pmatrix}$$

Therefore, we can compute $\mathbf{P^t = Q'\Delta^tQ}$ where $\boldsymbol{\Delta}^t$ being a power of a diagonal matrix can be obtained by taking powers of its elements as shown above. This seems like a nice way to compute things but in practice finding **Q** requires computing the eigenvalues of **P**, which takes some computational time and also creates numerical errors (i.e., errors coming from the fact that computers represent number with finite precision). Thus, using iterated matrix squares can be a better way to compute things for some matrices and reasonable values of *t*.

The Markov chain model is determined basically by the transition matrix. We can fix the transition matrix to have particular values—that is, actual numbers, or make the elements of the matrix be variables where the variables are now the parameters of the model. Then, a given model class is determined by the form of the transition matrix. For example, consider the following transition matrix for the four states of DNA:

$$\begin{pmatrix} \cdot & p & p & p \\ p & \cdot & p & p \\ p & p & \cdot & p \\ p & p & p & \cdot \end{pmatrix} \qquad \text{(JC)}$$

(In my notations, I will often skip the diagonals of stochastic matrices, since we know that the rows sum to one. The matrix above has diagonal values $1 - 3p$.) The matrix name (JC), represents a model called the Jukes-Cantor model where all possible transitions are equally likely. The Jukes-Cantor model was first studied when we didn't know much about DNA mutations and it was simple to assume that any nucleotide can change to any other nucleotide with equal probability. The nucleotides of DNA form two different chemical groups called purines (A and G) and pyrimidines (C and T). There are two within chemical group changes (A $\leftrightarrow$ G and C $\leftrightarrow$T). This type of changes is called *transitions*. There are four possible between group changes (e.g., A $\leftrightarrow$ C). This type of change is called *transversions*. It is an empirical fact that DNA seems to show two classes of mutation rates that are distinct between transitions and transversions (up to a certain level of accuracy; in detail there might be many other rates). To incorporate the assumption that transitions and transversions might have different probability we can construct the following model:

$$\begin{array}{c|cccc|} & \mathrm{A} & \mathrm{C} & \mathrm{G} & \mathrm{T} \\ \mathrm{A} & . & p & q & p \\ \mathrm{C} & p & . & p & q \\ \mathrm{G} & q & p & . & p \\ \mathrm{T} & p & q & p & . \end{array}$$

In the above, transversion probabilities are denoted as p and transition probabilities are denoted as q. This model is called the Kimura 2-parameter model (also called K2P model).

One of the possible model considerations is whether the probability model is such that processes are symmetric for forward and backward time increments. That is, do the processes look the same whether time flows forward or backward? At first glance, it is not so clear that we would want the processes to look the same. However, if we think of each time step as a very small increment in some physical-chemical process, we might decide fundamental processes should not look different whether time goes forward or backward. If this is the case, then the model must satisfy $\pi_i p_{ij} = \pi_j p_{ji}$ for some probability distribution of the states π, where π_i is the probability of state i and p_{ij} is the probability of transition from i to j. In matrix form:

$$\boldsymbol{\pi}' \cdot \mathbf{P} = \mathbf{P}' \cdot \boldsymbol{\pi}$$

Transition matrices satisfying this time reversible constraint can be constructed for 9 free parameters for 4-states and this is called the General Time Reversible (GTR) model.

While we have been discussing models with integer time index, it is more common to consider a real-valued time index—in practice there isn't a huge difference between the two types; but there can be some technical differences as we will discuss later. To construct a continuous time finite-state Markov model we start by assuming that changes in probability are governed by the following equation:

$$\vec{\pi}(t+h)' = \vec{\pi}(t)' + \vec{\pi}(t)' \cdot \mathbf{R} \cdot h \qquad \text{(Eq 10.7)}$$

Eq 10.7 means that the probability vector of states at time $t + h$ can be obtained as the probability at time t plus a transition term $\vec{\pi}(t)' \cdot \mathbf{R} \cdot h$. The transition term involves a matrix **R,** time interval h, and the probability vector from the previous time t. The elements of the matrix **R**, q_{ij} denotes a quantity that represents an instantaneous rate of transition from state i to j. To make sure that Eq 10.7 represents probabilities, the rows of **R** must sum to zero—analogous to rows of stochastic matrices summing to one. Furthermore, $q_{ii} \leq 0$ and $q_{ij} \geq 0$ for $i \neq j$. Thus, after h amount of time, we assume $q_{ij}\, h$ amount of change from state i to j. Taking the product with $\vec{\pi}(t)'$ is exactly analogous to the operation we did for (Eq 10.2)—where we were summing up all the state transitions that "flowed" into the jth state. [I note that all of this is simplified formalism aided by the fact we are working with finite states with non-pathological cases (which is appropriate for standard biological modeling).]

So, this is our model: In a small interval of time, the probability of the *j*th state changes by a small amount which is the sum of total amount of flow from other states into *j* and flow out of *j*th state. The flow into *j*th state is obtained by multiplying the probability of other states at previous time with numbers q_{ij} and the amount of time; that is, the flow is proportional to time passed. We can rewrite (Eq 10.7) and let *h* go to zero and obtain:

$$\vec{\pi}(t+h)' - \vec{\pi}(t)' = \vec{\pi}(t)' \cdot \mathbf{R} \cdot h$$
$$\lim_{h \to 0} \frac{\vec{\pi}(t+h)' - \vec{\pi}(t)'}{h} = \vec{\pi}(t)' \cdot \mathbf{R}$$
$$\frac{d\vec{\pi}(t)'}{dt} = \vec{\pi}(t)' \cdot \mathbf{R}$$

That is, we obtain a linear (homogeneous) differential equation for the vector $\vec{\pi}(t)'$. Standard methods for solving differential equations give us the solution:

$$\vec{\pi}(t)' = \vec{\pi}(0)' e^{\boldsymbol{Rt}} \qquad \text{(Eq 10.8)}$$

where $e^{\boldsymbol{Rt}}$ is called a matrix exponential. The exponential of a (well-behaved) matrix is well defined as discussed in the **Notes on Vectors and Matrices**. We can compute $e^{\boldsymbol{Rt}}$ by diagonalizing **R** as we did for the discrete time case. Then we have:

$$e^{\boldsymbol{Rt}} = \boldsymbol{Q}'^{e^{\Delta t}}\boldsymbol{Q} = \boldsymbol{Q}'\begin{pmatrix} e^{\lambda_1 t} & & \\ & \ddots & \\ & & e^{\lambda_k t} \end{pmatrix}\boldsymbol{Q} \qquad \text{(Eq 10.9)}$$

The Markov memoryless condition for continuous time models is stated a bit abstractly compared to (Eq 10.6). Let *X(t)* be the random variable of a stochastic process over discrete state space. Then *X(t)* is a continuous time Markov chain if for any time points $p_1 < p_2 < p_3 < \ldots < s < t$:

$P\{X(t)\ |X(s), X(p_1), X(p_2), \ldots X(p_n)\} = P\{X(t) \mid X(s)\}$ (Eq 10.10)

Eq 10.10 says that if we take any collection of time points prior to *t* then the conditional probability of the state at the *t* only depends on the most recent time point prior to *t*.
We now imagine observing the system at some *i*th state. Then, after some waiting time *w*, the stochastic process might make a transition to a different state. The distribution of the waiting time—that is, how long you wait before a transition happens, will be an exponential distribution with parameters that are the sum of the off-diagonal elements of the *i*th row of **R**. In general, the waiting time until a change in state happens will be exponentially distributed with parameters that are functions of rows of **R**. Furthermore, let $\vec{d} = -Diag(\mathbf{R})$ and $r = \sum_{i=1}^{k} d_i \pi_i$ where π_i is the equilibrium probability of the *i*th state. Then, given that the state of the system is in equilibrium, the expected number of transitions in a time interval *t* will be *rt*.

If we evaluate a continuous Markov chain at regular intervals, we obtain an embedded discrete time Markov chain. That is, if we evaluate (Eq 5) at points 0, d, 2d, … td, we get e^0, $e^{\mathbf{R}d}$, $e^{\mathbf{R}2d}\ldots e^{\mathbf{R}td}$ so that if we let $\mathbf{P} = e^{\mathbf{Rd}}$ then the process iterates $\mathbf{P}^0, \mathbf{P}^1, \mathbf{P}^2 \ldots \mathbf{P}^t$. The technical part is that if we have some discrete time Markov chain whose transition probabilities are given by $\mathbf{P}^0, \mathbf{P}^1, \mathbf{P}^2 \ldots \mathbf{P}^t$, we can't always obtain this as an embedding from some continuous time process. Being able to relate to continuous time process involves some restrictions on the form of **P**. This kind of technical restriction has some consequences for advanced problems.

[I should note that in textbooks there is a description of embedded discrete time Markov chain within a continuous time model that is different from the notion I discussed in the previous paragraph. Here, given a continuous time Markov process with a rate matrix **R**, we construct a new discrete time transition probability matrix **P** by setting each off diagonal element $p_{ij} = r_{ij}/(\text{sum of } r_{ij})$ where "sum of r_{ij}" is the row sum excluding the diagonal element. We also specify a series of "jump times", which is when the transitions happen in continuous time. These jump times are drawn from exponential distributions. Thus, the continuous time model can be restated as two coupled processes: (1) an embedded discrete time model of state transitions using the matrix **P**, and (2) a set of jump times which maps each discrete time step to real valued times.]

Probabilistic Model Computations

From the computations in the previous sections we can compute the probability of a multiple aligned dataset with for a *t*-leaf tree; i.e., $P(D|T, \boldsymbol{\theta})$, where T is the tree topology and $\boldsymbol{\theta}$ is the parameters of the tree and rate matrix **R**. To use this computation in an estimation framework, we can now apply either the maximum likelihood (ML) or the Bayesian framework.

ML estimation

1. For each tree topology T, numerically maximize $P(D|T, \boldsymbol{\theta})$ for all allowable values of $\boldsymbol{\theta}$.
2. Use a tree topology search procedure to maximize over T, maximizing $\boldsymbol{\theta}$ for each T.
3. The tree topology T and the particular values of $\boldsymbol{\theta}^*$ that maximizes $P(D|T, \boldsymbol{\theta})$ is called the maximum likelihood estimate.

Bayesian estimation

1. To compute the probability of some tree topology T and particular values of $\boldsymbol{\theta}$, assume a prior distribution of T (typically uniform) and $\boldsymbol{\theta}$ (typically a distribution like the gamma distribution).
2. Compute $P(D|T, \boldsymbol{\theta})P(T, \boldsymbol{\theta})$. Typically we assume independent prior distribution for the tree topology and the $\boldsymbol{\theta}$; therefore, this is $P(D|T, \boldsymbol{\theta})P(T)P(\boldsymbol{\theta})$.
3. Find some way to integrate $P(D|T, \boldsymbol{\theta})P(T, \boldsymbol{\theta})$ over the set of possible trees and vector values of $\boldsymbol{\theta}$.

For both of the estimation methods, we need to compute $P(D|T, \boldsymbol{\theta})$, which took exponential time in the enumeration algorithm I gave previously where we had to sum over the $O(k^t)$ number of possible ancestral state assignments (*t* number of leaves, *k* number of states). For the Bayesian estimation we also need an efficient way to compute the posterior distribution by integrating over the set of possible trees and $\boldsymbol{\theta}$ values. We will first discuss an efficient dynamic programming algorithm for $P(D|T, \boldsymbol{\theta})$ and then discuss an approximation algorithm for posterior distribution computation.

Dynamic programming algorithm for P(D|T, θ).

Here we assume that we have known numerical values for $\boldsymbol{\theta}$ and a fixed tree T, so in the notations I will drop the conditioning on T and $\boldsymbol{\theta}$; i.e., when I write $P(D| v = s)$ what I mean is $P(D | v = s, T, \boldsymbol{\theta})$.

Suppose some ancestral vertex *v* has *n* leaves in the clade that are descendants of *v*. Let $p[s_1..s_n | v = u]$ denote the probability that the *n* leaves of the clade has state assignments $[s_1..s_n]$ given *v* started with the state *u*. Now suppose that the vertex *v* has left and right daughter vertices that we will call *x* and *y*. Furthermore, *x* has *k* leaves in its clade with the state assignments $[s_1 \ldots s_k]$ and *y* has *n-k* leaves in its clade with the state assignments $[s_{(k+1)} \ldots s_n]$. Therefore, if we were to concatenate the leaf assignments in the clade from *x* and *y*, we will get $[s_1 \ldots s_n]$. We want to compute $p[s_1..s_n | v = u]$ as a function of the two daughter conditional probabilities $p[s_1 \ldots s_k | x = s]$ and $p[s_{(k+1)} \ldots s_n | y = t]$. We first compute:

$$p[s_1 \ldots s_k | v = u] = \textstyle\sum_{s \in S} p[s_1 \ldots s_k | x = s] p_{v=u \to x=s} \qquad \text{(Eq 11.1)}$$

where $s \in S$ are all possible states and $p_{v=u \to x=s}$ is the probability of transition from state *u* to state *s* in the vertices *v* to *x*. Eq 11.1 says that to get the probability of the [$s_1 \ldots s_k$] state assignments in the clade descending from the left daughter of vertex *v*, conditioning on *v* = *u*, we have to consider all the possible states at the daughter vertex *x* and the transitions from *v* to *x*. The equation is, in fact, from the Chapman-Kolmogorov equations we discussed before. We can now compute probability along the other daughter:

$$p[s_{k+1} \ldots s_n | v = u] = \sum_{s \in S} p[s_{k+1} \ldots s_n | y = t] p_{v=u \to y=t} \qquad \text{(Eq 11.2)}$$

Recalling the conditional independence of the probabilities for a Markov chain over a tree graph we can multiply (Eq 11.1) with (Eq 11.2) to get the joint probability:

$$p[s_1 \ldots s_n | v = u] = \left[\sum_{s \in S} p[s_1 \ldots s_k | x = s] p_{v=u \to x=s}\right]\left[\sum_{s \in S} p[s_{k+1} \ldots s_n | y = t] p_{v=u \to y=t}\right] \text{(Eq 11.3)}$$

We now define the marginal case of *p*[$s_1 .. s_n$ | *v* = *t*] if *v* is a leaf vertex. In this case, we can have only one possible state value so we want to define *p*[*s* | *v* = *t*]. It is clear that we will have:

$$\begin{cases} if\ s = t, p[s|v = t] = 1 \\ else,\ p[s|v = t] = 0 \end{cases} \qquad \text{(Eq 11.4)}$$

That is, at the leaves the conditional probability is defined as 1 for *p*[*s* | *v* = *s*] and zero for all other cases.

With (Eq 11.3) and the marginal case (Eq 11.4), we can now define a post-order traversal of the tree graph, recursively applying (Eq 11.3). Once we reach the root of the tree, we will have computed p[$s_1 \ldots s_t$ | *r* = *s*] for a t-leaf tree. We then apply the marginal probability of each state at the root of the tree to obtain:

$$p[s_1 \ldots s_t] = \sum_{s \in S} p[s_1 \ldots s_t | r = s] \pi(s) \text{ (Eq 11.5)}$$

Eq 11.5 is then used in the multinomial sampling distribution to yield P(D | T, **θ**) as previously. It is important to note that if we examine the form of Eq 11.3, the quantities *p*[$s_1 \ldots s_k$ | *x* = *s*] and *p*[$s_{(k+1)} \ldots s_n$ | *y* = *t*] have already been computed in the daughter vertices. Therefore, we only do the summation and product for each of the possible transitions from the two daughters to the current vertex. For each non-terminal edge in the tree, the algorithm computes O($|S|^2$) transition probabilities, where S is the set of possible states. So the algorithm has time complexity O($|S|^2 t$) for a *t*-leaf tree.

Maximum Likelihood estimate of ancestral states

Previously, we learned maximum parsimony estimate of ancestral states. We can use likelihood calculations to also estimate ancestral states. Assume that the tree topology is known and also we now know all the parameters of the Markov chain over the tree. We also assume that the Markov model is time reversible. Let vertex *v* be an ancestral vertex whose states we want to estimate. Then we can set *v* to be the root of the tree and compute the marginal conditional probability *p*[$s_1 \ldots s_n$ | *v*=*u*] using the dynamic programming algorithm (Eq 11.3). We can compute the conditional probability for each of the possible states and choose the state with the highest conditional probability of the data, i.e., maximum likelihood. Alternatively, we can use Bayes' formula to compute posterior probabilities, given some assumption about prior probabilities of the ancestral states. The computations are straightforward since we only need to integrate the denominator over the set of states. We can compute these ancestral state estimates for each vertex in turn. I note that estimates obtained in this manner are not independent of each other nor are they a simultaneous estimate of all the ancestral vertex states. Rather, we can think of each ancestral state estimation obtained in this manner as a marginal estimate integrating over all the possible states at all other ancestral vertices.

The dynamic programming algorithm solves the problem of efficiently computing the likelihood. We now need an algorithm to relatively efficiently approximate the posterior distribution for Bayesian estimates.

Markov Chain Monte Carlo (MCMC)

Markov Chain Monte Carlo, commonly called MCMC, is a technique for sampling from a probability distribution in a way to obtain an approximation of the distribution. We first consider the simple idea of sampling from a probability distribution. There are many cases in which we can sample from a probability distribution with reasonable efficiency (and accuracy) but we may have harder time exactly computing the distribution itself. As somewhat of an artificial example, the normal distribution with mean 0 and variance 1, N(0, 1), is given by the probability distribution function $p(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2}$. If we wanted to use it to know Prob{X < c} we would have to integrate *p(x)*, which turns out to be not so easy. But, we might have a method to sample from the N(0,1) distribution. We could sample, say a million times and the resulting frequencies (typically summarized as a histogram) gives us an approximation of the exact distribution. More samples we have more accurate will be our estimate.

We give an algorithm for drawing an approximate N(0, 1) random sample:

1. Let U ~ [0, 1] be a uniform random number between 0 and 1. Most computer programming languages have a function for generating a uniform pseudo-random number in the [0, 1] interval.
2. Draw from U~[0, 1] twelve times and sum up the result. Let X be the sum.
3. Return Y = X – 6.0

This algorithm relies on the central limit theorem where the sum of random variables converges to a normal distribution. The mean of 12 uniform random numbers in [0, 1] is 6.0. The variance of U ~ [0, 1] is $\int_0^1 (x - \frac{1}{2})^2 dx = \frac{1}{12}$, so the variance of X, the sum of 12 such random numbers is 1; therefore the standard deviation of X is 1. Therefore, Y = X – 6.0 is an approximate sample from N(0, 1). We can now sample Y some suitable number of times, say a 1000 times. We get an empirical histogram from this sampling to approximate *p(x)* and we can easily approximate Prob{Y < c}.

Returning to the problem of Bayesian posterior distributions, we want to generate samplings from the posterior distribution such that with enough samples we can get a good idea of the posterior distribution. The idea here is to generate such samples using an iterative process. Recall the definition of an equilibrium distribution of a Markov chain. The goal of MCMC algorithm is to construct a Markov chain such that its equilibrium distribution is, in fact, the probability distribution that we want. At this point, I have to emphasize that the Markov chain part of this algorithm has nothing to do with the Markov chain that we used to model sequence evolution over a tree. It just happens that we are going to use a Markov chain to sample from a desired distribution, but the technique itself is a general technique not related to sequence evolution models.

Let's start with something incredibly simple. Suppose we have a two state A/B Markov process. At each transition event, we toss a coin and move to the other state with probability ½. Suppose we start in state A. You can think of this as a marginal distribution with probability 1 in state A and zero in state B, denote this by π'(0) = (1,0). After the first step, we have π'(1) = (0.5, 0.5). After the second step we have π'(2) = (0.5, 0.5) again, so we've quickly reached the equilibrium distribution. Let's see what happens with the actual events. We start with A. At the next step, we might be in B (with probability 0.5). The next step we might be again in B, and so on. So, we might come up with a sequence ABBBABBBAABABAA... and so on. Immediately after the first step there should be an equal probability of occurrence of A and B, so counting A's and B's should give us a good approximation of the equilibrium distribution of (0.5, 0.5). Let's change the model slightly to drive this point home. Suppose at each step we make a change to the other state with probability ¼ and stay in the same state with probability ¾. We again start in state A and the marginal distribution starts π'(0) = (1, 0). After one step we have π'(1) = (0.75, 0.25). Step 2, π'(2) = (0.625, 0.375). Step 3, π'(3) = (0.563, 0.437). Step 4, π'(4) = (0.531, 0.469). Step 5, π'(5) = (0.516, 0.484). We can see this will soon go to π' = (0.5, 0.5). Now let's do an actual trial:

ABBBBBAAAABABBBBAAAAABBBBAAAAAABAAAAAAAABBBBBAAABBBAAB........

While it may not seem that obvious, what our calculation tells us that when we look at this sequence of letters, after some initial steps, the proportions of A's and B's should be equal—that is, the marginal distribution should be (0.5, 0.5). Of course, for any short segment, we tend to have runs of letters because the process tends to stay in one or the other state. However, the long run proportion of the letters will still be 50:50. The result of our construction is a Markov chain whose equilibrium distribution is the target distribution, Prob{H = 1/2)}. If we ever had a problem computing the probability distribution of a flip of a fair coin, we could have used the above process and counted the long term frequency of A's and B's.

As we just mentioned, this long run frequency can be only used after the system has reached (or nearly reached) equilibrium. The period prior to equilibrium is called the "burn-in" period. In a simple model like the above, the burn-in period is short. In a complex model, it is not clear and this is one of the most difficult issues in using MCMC. We know that we can use frequency counts after some initial period, but what is the duration of that initial period? There is no easy way to answer this question—but people use various rules-of-thumb.

Now suppose we want an estimate of some *pdf*, how do we get a Markov chain to converge to this distribution? There are many different ways to do this. One straightforward way is called the Metropolis-Hastings algorithm. The algorithm involves two components. The first component is the state space for the Markov chain (MC). For phylogenetic trees, the state space is the tree topologies and the sequence evolution model parameters. The second component is a way to generate transitions within the state space with a probability model that guarantees convergence to the desired equilibrium distribution. Recall back to our discussion of Markov models. The essence of the Markov model is a set of conditional probabilities for transition events between any two points in the state space. Such conditional probabilities can be explicitly stated as they were for models like the Jukes-Cantor model. Or, we may have a way of generating a sample transition from the Markov model without having an explicit formula for the transition (i.e., conditional) probabilities. The Metropolis-Hastings algorithm gives us a way of generating such sample transitions, such that the distribution of the samples will converge to the desired probability distribution in the long run (i.e., the equilibrium distribution).

Metropolis-Hastings algorithm starts the MC in some state; say, a particular tree topology and a combination of edge parameters. Then it samples from the transition probability using two steps: (1) a step to propose a new state; (2) a step to probabilistically accept or reject the proposed step. We will denote by *q(x,y)* the probability of proposing a new state *y*, given that the MC was in state *x*. We will call this the ***proposal function***, meaning that this probability function governs the proposal of a new state from the current state *x*. One possible proposal function is *q(x,y) = q(y)*—that is the proposed new state does not depend on the current state. For example, in a phylogenetic tree case, one could propose any possible binary tree with some probability *q(y)*. This is called an independence chain. More commonly, the proposed new state depends on the current state. For example, we could propose one of the Nearest Neighbor Interchange trees with equal probability.

In the second step, we accept the new state with the following probability:

$$\Pr ob\{Accept\ state\ y\} = \min\{\frac{p(y \mid data)q(x,y)}{p(x \mid data)q(y,x)},1\} \qquad \text{(Eq 11.6)}$$

First note that the reason we have *min(A,1)* on the right hand side is because the ratio *A* may be bigger than 1 and we want this to be a probability and bounded by 1. The numerator of the ratio is comprised of the product of the posterior probability of the state *y* (i.e., *p(y|data)*) and the proposal probability of changing to *y* from *x*. The denominator is the complement of this: it is the product of the posterior probability of the state *x* and the proposal probability of changing to *x* from *y*. If *q(x,y) = q(y,x)*, that is the probability of proposing *y* from *x* is the same as proposing *x* from *y*, then the ratio is simply the ratio of the two posterior probabilities. Remember that the posterior probabilities can be obtained from the Bayes' formula. Wait, but doesn't Eq 11.6 involve the posterior distribution? Didn't we say this is what we are trying to compute with the MCMC algorithm? Fortunately, what we need is the ratio of the two posterior probabilities, not the individual posterior probabilities. Thus from Bayes' formula the integrals in the denominators cancel and we only need to compute:

$$\frac{p(y \mid data)}{p(x \mid data)} = \frac{p(data \mid y)p(y)}{p(data \mid x)p(x)} \qquad \text{(Eq 11.7)}$$

where *p(data|x)* is the likelihood and *p(x)* is the prior distribution. We especially note that for the formula in Eq 11.7, we only need the likelihood and the prior probability for a particular state value *x* (or *y*). Thus we are not trying to compute the full range of the function but just one value. This is especially important when the state space is extremely large such as for binary trees. It is easy to compute the likelihood for a particular tree, hard for all possible trees.

To apply the Metropolis-Hastings MCMC method for trees, we start at some random initial binary tree with particular transition matrices on the edges, denote this (*T*, ***θ***), where T represents the tree topology and ***θ*** represents the **R** matrix and edge parameters. We first compute the likelihood of the data *p(data | T,* ***θ****)* using the dynamic programming algorithm. We also assume we have a prior probability distribution from which we can compute *p(T,* ***θ****)*. Then we propose a new tree and set of edge transition matrices *(T',* ***θ'****)* from some proposal function. We compute the likelihood *p(data | T',* ***θ'****)* and prior probability *p(T',* ***θ'****)*. Now, we compute

$$\frac{p(data \mid T',\theta')p(T',\theta')q(\{T,\theta\},\{T',\theta'\})}{p(data \mid T,\theta)p(T,\theta)q(\{T',\theta'\},\{T,\theta\})} \qquad \text{(Eq 11.8)}$$

That is, the product of the ratio of the posterior probability and the ratio of the proposal probabilities. We draw a random number from 0 to 1 and then accept the new state if the random number is lower than Eq 11.8. We repeat this over and over again. (Millions and millions and millions of times.)

After a sufficient burn-in time interval, we collect the states (trees) visited by the MC—say hundreds of thousands of them. We may collect with some spacing between the samples so that we avoid the problem of correlated states within any finite sampling interval. This is an approximate sample from the posterior distribution (if the burn-in time was sufficient). We may find the most probable tree among this collection or represent the whole collection by taking a majority-rule consensus tree we previously discussed.

Several remarks are due:

1. For possibly complicated MCMCs, it is hard to know how long to wait before we finish the "burn in" period. To see how difficult this might be, think of the problem of a multinomial distribution with a very large state space—say a trillion possible outcomes, each with probability p_i, $i = 1$ to *trillion*. We have some way of taking samples from this model and we will use the sample to estimate p_i. On one hand, it would seem like we should take at least a few trillion samples to get a good idea of the various p_i values. On the other hand, maybe only ten outcomes have significantly large p_i and everything else is near zero. In which case, if the MCMC tended to "hover" around those ten cases, then we would get a very good estimate of the posterior distribution. For the phylogeny tree problem, the state space who probability distribution we want to estimate is the binary trees and the scaled time parameter for each edge (assuming that the rate matrix **R** identical for every edge by assumption). As we discussed previously we have $O(t^t)$ number of binary *t*-leaf trees, so the size of the state space can be very large. Whether the

situation is like ten major probable states out of trillions or 100's of billions out of trillions depends very much on the data.

2. If a tiny proportion of the trees have most of the probability mass, the posterior distribution would be most efficiently estimated if we could have the Markov chain hover around those states. The actual occupancy of the states should be proportional to the relative posterior probabilities as given in Eq 11.6, so the main determinant of whether we would be sampling those states with high frequency depends on the proposal function. In fact, the "black art" of MCMC algorithms is designing the proposal function. As mentioned before, a possible proposal function for trees is something like considering neighboring edit trees with equal probability. This is generally a good thing since neighboring trees are likely to be similarly probable. So, if we have a tree with high probability then we would probably want to sample pretty well around it. But, we do not want our Markov chain to be stuck in some local region of the state space without getting to some other region that also has high probability. We would get stuck if the transition probability was low from one set of states to another set of states—that is, if there were some very low probability trees surrounding high probability trees. A good proposal function should allow us to jump through such barriers. In sum, as you can see, getting a good sample of the posterior distribution has all of the same issues as getting a good optimization of an objective function.

3. One other, not so minor, issue is that even if the Markov chain is somehow guaranteed to be in an equilibrium distribution, it may be still difficult to get a good sample of the posterior distribution. First, there might be very many high probability states, say millions. We generally need a sample size that is some large multiple of those high probability states so we might need a very large number of samples. Furthermore, the Markov chain may spend a large percent of its time in one subset of the probable states and then switch very rarely to another subset of the probable states and so on. This will create strong correlation in the successive state samples, which will require even more samples to alleviate.

4. If the Markov chain is not in equilibrium or if we don't have enough samples, then we will get a very biased estimate of the posterior distribution. On the other hand, if we were solving the maximum likelihood optimization problem and if we were stuck at some local optimum for the maximum likelihood methods, we would also get a very biased estimate. So, such problems are not unique to the Bayesian-MCMC methods. Unfortunately, these kinds of situations are rather hard to diagnose.

All of this discussion suggests we need to discuss properties of estimators next.

Unit 14: Properties of Estimators

Natural science is the act of systematic application of inferential methods. All measurements have some kind of uncertainty, whether it is inherent to the process or because of the limits of our measurement devices. Therefore, when we apply inferential methods to derive model-theoretic statements (i.e., associating data with specific models), we would like to know what to expect from using a particular method. There are three basic characteristics of inferential methods that concern us. First, we know that the inference method depends on the input data so we would like to know how the method might behave as more and more data is made available. Is the method **self-consistent** in the sense that more data forces the method to converge on a particular model? Second, the main concept that distinguishes natural law from subjective observations is the concept of **replication**. Therefore, we would like to know how the inferential method might behave if we were to repeat the process, say with a new set of observations. Finally, we always want to know how broad is the domain of any theory or model so we would like to know how the inferential method might behave with unusual types of data or observations. All of the properties of inference methods discussed below are our attempts to operationalize and answer these questions. Obviously, there are many different frameworks for inference. Here, I will only discuss the case of inferences under a probabilistic framework and call the inferential methods, **estimators**.

Ideally, questions like what would happen if we were to apply the same estimator to new data should be investigated with actual new data. Sometimes we are able to do this and, of course, ultimately that is the main verification of our model inferences. However, we want to understand, even hypothetically, what would happen if we were to use one particular estimator versus another without having to have "infinite" amounts of data. A key part of operationalizing our investigation of properties of estimators is to assess their properties under the assumption that data is derived from some model-theoretic rules—that is, from a probability model.

Let D_n be a dataset of size n, assumed to be samples from some probability model family $\boldsymbol{M}(\boldsymbol{\theta})$ where $\boldsymbol{\theta}$ represents the parameters of the model (not necessarily just numbers, cf., tree topologies). We will say $\phi(D_n)$ is an estimate(s) of some subset of the parameters, which we will denote by $\boldsymbol{\mu}$. So, we wish for $\phi(D_n) = \boldsymbol{\mu}$. There will be many different methods for computing $\phi(D_n)$ such as maximum likelihood methods, Bayesian methods, least-squares methods, etc. I will adopt a somewhat abstract notation of indexing the methods by α and then say $\phi_\alpha(D_n)$ is an estimate computed using method α. Since D_n is a random sample from a probability model and the estimate $\phi(D_n)$ is a function of D_n, it is also a random variable and has a sampling distribution. (Remember I previous said a function of the data is called a "statistic"; so, an estimate is also a statistic.) Therefore, most of the properties of interest are the properties of the sampling distribution of $\phi(D_n)$. Here are some traditional properties of interest for an estimator:

Consistency
Variance
Bias
Accuracy
Confidence
Efficiency
Robustness

Not all of the above properties that have been traditionally studied in statistics can be simply applied to the kinds of estimators used in computational biology; especially phylogeny problems. In the following, I will describe the main idea, traditional definitions, and then some idea of extensions to phylogeny estimation cases. Unless noted otherwise, when we are using D_n to estimate some parameter set $\boldsymbol{\mu}$ of a probability model, $\boldsymbol{M}$, we will assume that D_n is, in fact, a sample from model $\boldsymbol{M}$ with parameter $\boldsymbol{\mu}$.

Consistency: This is the simple idea that (1) as the size of the input data increases the estimate should converge to something; and (2) it should converge to the parameter value that it is trying to estimate. If the estimate is a real number we define a method as consistent if the following holds:

$P\{|\phi_\alpha(D_n) - \mu| < \epsilon\} \to 1,\ n \to \infty,\ \forall(\epsilon > 0)$ (Eq 12.1)

This just says that the probability that "the difference between the estimate and the true value is effectively zero" goes to 1 as the size of the data goes to infinity. We have to make the probabilistic statement because the estimate is a random variable. (In fact, there are many different kinds of probabilistic convergence but we won't try to decipher all the technical details here.) For parameters like tree topology, we have to replace the notion of arithmetic difference with something else like $d(\phi_\alpha(D_n), T_\mu)$ where T_μ is the true tree topology and $d()$ is some notion of tree distance. For example, we could use an index derived from the strict consensus algorithm discussed before as a measure of tree distance. If we are not too concerned about "steady convergence" to some fixed tree, then we might define a tree estimation method to be consistent if $P\{\phi_\alpha(D_n) = T\} \to 1,\ n \to \infty$. Eq 12.1 and any analog asks whether an estimate is consistent for a particular value(s) of the parameter. But, of course, what we really want is for the method to be consistent for all possible values of $\boldsymbol{\mu}$. Sometimes a method will be consistent for a restricted range of $\boldsymbol{\mu}$; i.e., a restricted class of models. As an example, a classic result in phylogeny estimation concerns the consistency of the maximum parsimony estimator.

Consider a 4-leaf tree and two-state characters (0/1). We will label the leaves, A, B, C, D. For the four leaves of the tree, we will have $2^4 = 16$ possible characters:

```
A: 0000000011111111
B: 0000111100001111
C: 0011001100110011
D: 0101010101010101
```

We want to consider computing a maximum parsimony tree. By the rules of parsimony computations we discussed before, the only characters that matter are:

```
A:000111
B:011100
C:101010
D:110001
```

That is, characters with exactly two 0's and two 1's. If we consider the Wagner parsimony score, then the MP score of [0,0,1,1] and [1,1,0,0], the binary complement, are exactly the same. So for the MP score of a dataset, we will be only concerned with the relative frequency of these three possible characters:

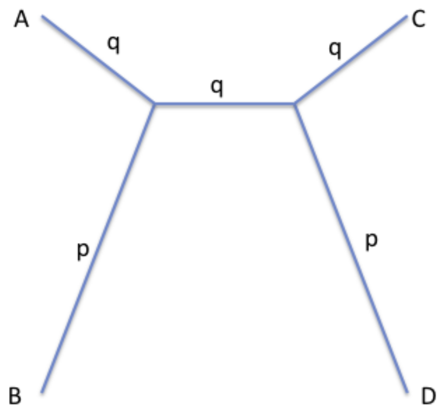
```
A:000
B:011
C:101
D:110
```

Let's consider the three possible tree topologies for 4-leaf trees: T1 = ((A,B),(C,D)); T2 = ((A, C), (B, D)); T3 = ((A, D), (B, C)). The parsimony length of each of the tree characters on each of the three tree topologies is given in this table:

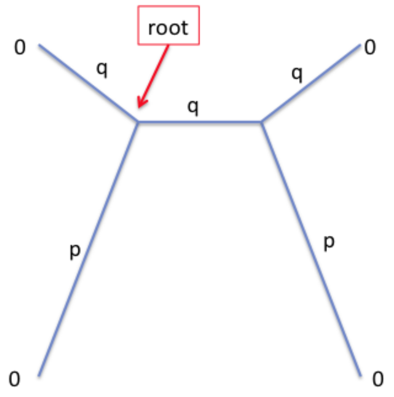
	Char 1	Char 2	Char 3
A	0	0	0
B	0	1	1
C	1	0	1
D	1	1	0
T1	**1**	**2**	**2**
T2	**2**	**1**	**2**
T3	**2**	**2**	**1**

So, if a dataset has more characters of char 1 than either of char 2 or char 3, then the MP algorithm will return T1. Similar reasoning holds for the other characters.

We now consider a Markov chain model of character evolution of a tree graph. We will assume that each transition probability matrix is symmetrical so that the probability of making $0 \rightarrow 1$ and $1 \rightarrow 0$ transition is identical. We also assume that the ancestral probability of state 0 and state 1 are identical at ½. We consider a tree that looks like this:



In this tree, there are three edges marked "q" and two edges marked "p". These two parameters represent the probability of $0 \rightarrow 1$ or $1 \rightarrow 0$ transitions on the respective edges. Note that this tree is tree topology T1 by the notation above. We wish to find the probability of the characters [0,0, 1, 1], [0, 1, 0, 1], [0, 1, 1, 0] on the leaves of this tree as a function of p and q. Since the ancestral probabilities are at equilibrium where both 0 and 1's probability = 1/2, we can just set any ancestral vertex as the root of the conditional probabilities. We will just set the left vertex as the root like this:



We then consider the four possible cases of the state assignments to the hidden ancestral vertices: [0, 0], [0, 1], [1, 0], [1, 1]. Here [x, y] means the root has state x and the other vertex has state y. As before we use the notation p[wxyz] = P{A = w, B = x, C = y, D = z}

$p[0011 \mid r = 0] = (1-q)(1-p)(1-q)q\,q + (1-q)(1-p)q(1-q)(1-p)$
$p[0011 \mid r = 1] = q\,p\,(1-q)(1-q)(1-p) + q\,p\,q\,q\,p$

$p[0011] = p[0011|r=0]*0.5+p[0011|r=1]*0.5$
$= ½*[2\,p^2q^2 - p^2q + q^3 - 2q^2 + q]$

Similar computations yield:

$p[0101] = ½*[2\,p^2q^2 - 3\,p^2q + p^2 - q^3 + q^2]$.

We note that, by Law of Large Numbers, p[0011] is the expected proportion of characters with "0011" state assignments at the leaves when we have infinite sized datasets. Likewise for p[0101]. So if the following is true:

$p[0101] - p[0011] = (1-2q)(p^2 - (1-q)q) > 0 \rightarrow p^2 > q(1-q)$

Then there are more characters of state assignment "0101" than "0011" so the maximum parsimony method will return tree T2 when presented with an infinite sized data drawn from tree T1. Thus, the maximum parsimony method is not consistent for this subset of models; i.e., the subset of models such that the edge parameters satisfy $p^2 > q(1-q)$. Using advanced techniques, we can show that such conditions can be found for many other Markov chain models over trees for the maximum parsimony method as well as for other estimation methods.

Variance: When we apply the estimator to *n*-sized data, we get one number. We want to know how different the estimate might be if we applied to a new set of *n*-sized data. That is, we would like to know $\mathrm{Var}(\phi_\alpha(D_n))$ where variance is computed with respect to the sampling distribution of D_n, the repeated sampling of same sized data. It is obviously desirable that the estimator returns similar estimates when applied to new data sampled from the same model. But, I note having low variance is not necessarily a good thing with respect to accuracy of the estimate—which is what we are ultimately concerned with. For example, an "idiotic" estimator that returns the same estimate regardless of the input data will have low variance but, like a stopped clock, will most surely be wrong. The same kind of reasoning will be true for bias, which we discuss below—low bias may or may not help with respect to accuracy.

If *$\phi_\alpha(D_n)$* was a real-valued random variable, variance would be defined as $E[\left(\phi_\alpha(D_n) - \overline{\phi_\alpha}(D_n)\right)^2]$ where $\overline{\phi_\alpha}(D_n)$ is the expectation of the estimate for *n*-sized samples. If *$\phi_\alpha(D_n)$* was a vector of real-valued numbers then we might be interested in the covariance matrix of *$\phi_\alpha(D_n)$* or some marginal property like the trace of the covariance matrix (i.e., the sum of the variances of each coordinate). If *$\phi_\alpha(D_n)$* is a discrete set like tree graphs then the definition has to be suitably modified. We can replace $\left(\phi_\alpha(D_n) - \overline{\phi_\alpha}(D_n)\right)^2$ with something like $d^2(T, \bar{T})$, i.e., the square of the distance between two trees. Using squares rather than just the distance preserves the idea that not only should we measure deviations as positive quantities but that we should emphasize the larger deviations. The problem is with $\bar{T}$, the notion of an average tree or an expected tree. How should we compute an average tree topology? Majority rule consensus tree is a kind of a centroid tree for a collection of trees, so that might be a possibility. Better yet, for real-valued variables, if we compute the sum of all squared pairwise differences of random observations, $W_n = \sum\sum_{ij}(x_i - x_j)^2$, then assuming each observation is *iid*, we get $\frac{W_n}{2n^2} = \frac{1}{n}\sum_i (x_i - \bar{x})^2$ so we could compute $\frac{1}{2n^2}\sum\sum_{ij} d^2(T_i, T_j)$ for the tree topologies, *T_i*, estimated on *n*-sized datasets.

Bias: The standard definition of bias is E($\phi_\alpha(D_n) - \mu$) or $E(\phi_\alpha(D_n)) - \mu$. While algebraically identical, these have slightly different meanings. The first measures the expectation of the differences between the estimate and the true value while the second measures the difference between the expected estimate and the true value. In both cases, when bias equals zero, we say our estimator is unbiased. When we try to translate the idea of bias into tree topology estimates the two definitions have different interpretations. The first definition captures the idea that if we are trying to estimate μ and we have lots of different estimates from *n*-sized datasets, the estimates should be scattered with μ as their centroid. That is, the estimates should not have a tendency to vary in a particular direction. If we replace $\phi_\alpha(D_n) - \mu$ with *$d(T,T_\mu)$* where *T_μ* is the true tree (the model tree from which the Markov model of sequence evolution was sampled), then we lose the notion of directionality; E(*$d(T,T_\mu)$*) = 0 does not mean that the estimates *T* are uniformly scattered around *T_μ* at the center. We might redefine bias of tree estimates to reflect the idea that an unbiased estimator should have no preferred directions. For example, we might examine *k*th NNI neighbors of *T_μ*. For each *k*, we might want the estimates from *n*-sized datasets to have a uniform distribution over the possible tree neighbors. On the other hand, it is not clear that NNI neighbors are important so there is considerable ambiguity in transferring the notion of bias to tree topologies. The second definition tries to enforce the idea that the average of lots of estimates from *n*-sized datasets should converge to the true value. If we can reasonably define average tree from collection of trees, then this definition can be easily transferred to the tree topology case. Given *k* number of trees with *k* possibly infinite, we might define the average tree as the majority-rule consensus tree as before. Another possibility is to find the modal tree, the tree that appears most frequently, if it exists. Then we can define bias = $d(\bar{T}, T_\mu)$ where $\bar{T}$ is the average tree and T_μ is the true tree. This definition is somewhat more straightforward but it is not entirely clear that it captures the core idea of bias, which really is that estimates from replicates *n*-sized datasets shouldn't have the tendency to err towards one set of trees more than other set of trees.

Accuracy: Accuracy measures how different the estimate is from the true value, on average over replicates of *n*-sized datasets. We might define it as E[($\phi_\alpha(D_n) - \mu)^2$], the expectation of the squared error; or, something like $\sqrt{E[\phi_\alpha(D_n) - \mu)^2]}$, which is also called root-mean-squared (rms) error. We can make a straightforward substitution of $\phi_\alpha(D_n) - \mu$ with *$d(T,T_\mu)$* for tree topologies and everything makes sense as long as *$d(T,T_\mu)$* captures the idea of error in tree topologies. Note that the main difference with variance is that we are measuring deviations from the true value as opposed to the mean value.

For real-valued random variables we have the following derivation:

$$E[(x-\mu)^2] = E[(x - E(x) + E(x) - \mu)^2]$$
$$= Var(x) - 2(E(x) - \mu)E\left(x - E(x)\right) + E[(E(x) - \mu)^2]$$

$$= Var(x) + bias^2$$

Thus, the expected error in our estimates can be decomposed into the expected variability of the estimate and the bias of the estimate. The interesting thing is that for different estimators the relationship between variance and bias is not simple. One could have an estimator with twice the bias but much smaller variance such that the total squared error is smaller than the original estimator. And, the relationship between variance and bias typically changes as a function of sample size. For example, suppose we were estimating the scaled expected number of mutations over a two-leaf tree as we previously discussed. Assume that the sequences were drawn from a continuous time Markov chain model with six different parameters for the rate matrix and we are using a maximum likelihood (ML) estimator. We can construct a ML estimator for the expected number of mutations based on either a simple Jukes-Cantor (JC) rate matrix or based on the original six-parameter rate matrix identical in structure to the one from which we sampled the data. It will turn out that the ML estimator based on the JC model is a biased estimator—roughly speaking it doesn't have the parameter flexibility to fit exactly to the observed data. However, for small sample sizes, it turns out to have smaller variance and lower squared error. It is these kinds of complexities that keeps people up at night and makes people develop new kinds of estimators.

Confidence: When we obtain an estimate of a parameter, it would be nice if we also somehow knew how sure we should be of the estimate—or, maybe better, a range of estimates that we think should cover the true parameter value with high probability. There are several different standard methods for computing an ensemble of estimates whose aggregate probability is high.

The classic confidence interval is obtained by assuming some model of sampling for D_n, computing the estimate $\phi_\alpha'(D_n')$ and then finding an interval of the sampling distribution of $\phi_\alpha'(D_n')$ that captures, say 95% of the distribution. The caveat here is that when we have empirical data, we don't have a true model for D_n. We will typically estimate the model first, then assume the model is true and compute sampling distribution of D_n from this model, and then find the confidence interval. Thus, we introduce a certain kind of circularity; therefore, we do not interpret the classic confidence interval as representing the probability that the true parameter value is contained in the interval. Rather, it is the interval that will contain estimates (as random variables) with high probability, if we were to sample *n*-sized datasets repeatedly from the estimated model. Under the Bayesian framework for estimation, we can obtain the posterior distribution of the parameter given the data. Thus, the estimation procedure already contains a probability statement about the parameter values. Given the posterior distribution, we can compute an interval of the parameter value for which the posterior probability has some desired value, say 0.95. Typically, such an interval will be centered at the mean or the median of the posterior distribution. We call such an interval either a Bayesian posterior interval or a credibility interval.

Finally, an ensemble of estimates can be obtained by what is called "**bootstrap (re)-sampling"** of the observed dataset. The idea is to generate copies of the original dataset as if we were obtaining new data of size *n*, then compute new estimates. If we do this multiple times, we can get an approximation of the classic confidence interval. We can generate bootstrap samples in two ways. First, we can estimate parameters of the model as usual, then computationally simulate data generation from the model—for example, estimate parameters of sequence evolution model and then simulate sequence evolution over a tree. This is called **parametric bootstrap sampling**. The second method is called **non-parametric bootstrap resampling**. We assume that the dataset consists of *n* units of independent characters (or observations); for example, an alignment with *n* nucleotides. We pick one character at random (with probability *1/n*). We copy the character to a new dataset, then we repeat the process, copying each time until we get another dataset with *n* characters. Note that since we sample with replacement, the same character can get picked multiple times. An ensemble of estimates can be obtained by repeating the bootstrap resampling many times (say 1000's).

For tree topology estimates, we can apply any of the previous principles but the easiest is the Bayesian credibility idea or the bootstrap resampling idea. The main problem is how to summarize the ensemble of estimates obtained either by MCMC algorithm in Bayesian estimates or bootstrap estimates. The most common method is to compute the majority-rule consensus tree as a representation.

Efficiency: Efficiency is not very precisely defined. Basically, we would like to know how the variance or the accuracy of an estimator scales as a function of the sample size n. Typical estimators have variances that scale $O(1/n)$. One might think of efficiency as a measure of how well the estimator uses input information to gain inference of the model. For phylogeny estimators we do not know too much about the efficiency of various methods and researchers have carried out many simulation-based tests. For the phylogeny estimation problem, one other characteristic of the problem is the number of leaves, t. We might want to know the scaling relationship between the dataset size, n, and the problem size, t, such that we have constant error rate. That is, suppose I am happy with some fixed error rate $\mathrm{E}(d(T, T_\mu)) = \varepsilon$ for my tree estimate. It is most likely that if the problem size increases by increasing the number of leaves the error rate might go up, while if we add more data, the error rate might go down. So, I might want to know $n = O(f(t))$ such that $\mathrm{E}(d(T, T_\mu)) = \mathrm{O}(\varepsilon)$ for all t and n. For phylogeny problems there has been several results that suggest $n \sim O(\log t)$ for $\mathrm{E}(d(T, T_\mu)) = \mathrm{O}(\varepsilon)$ under certain restricted conditions. This is a very happy result because it tells us that as the problem size becomes larger in terms of number of leaves, we only need to increase the total number of characters slightly (log of the number of leaves) to maintain similar expected accuracy.

Robustness: Robustness is somewhat of a loose idea that we should be more confident of our estimates if they do not change much when the data is unusual or if the data is drawn from not exactly the model we were expecting. For the latter idea, the parameter we are estimating must be a parameter common to different models. For example, we might be interested in the tree topology as a common parameter across many different models of continuous time Markov chain on the edges of the tree. There is a strong intuitive sense that if some estimate changes little even if we change the estimation method, assumed model, or source of data, then the estimate is more likely to be true. Of course, other properties must also have good measures otherwise the "idiot" estimator that returns the same value regardless of the data will also be a robust estimator too. An interesting question is how the estimate changes as the problem itself is slightly perturbed. One type of problem change is adding data for a new leaf given that I already have an estimate for a t-leaf tree; i.e., extending the solution to $(t+1)$-leaf tree. The NP-hard part of the optimization problem for phylogenies tells us that for most datasets, the estimate will not be robust; i.e., we can't just add the leaf to existing tree estimate