1 Computational Biology

Computational biology is the application of computational tools (algorithms) to automatically extract information and/or patterns directly from biological data. Today, that biological data is most commonly derived from DNA sequences. Understanding the basics of the algorithms does not require very much knowledge of biology. Of course, the more biology we learn, the more clearly we see that underlying data can be quite messy, that the assumptions we make to design our algorithms are often not biologically realistic, or that we simply don't know as much about either the data-generating or the evaluation process as we like to think. These are important caveats to keep in mind when learning computational biology algorithms.

For our purposes, a brief biological introduction is sufficient. A DNA sequence is a sequence of ℓ characters, denoted $x_1, x_2, x_3, \ldots x_\ell$, each of which is drawn from the alphabet $x_i \in \Sigma = \{A, T, G, C\}$. In an organism, DNA sequences are "double-stranded" meaning that each sequence x is paired with a complementary sequence x', with $x'_i \in \Sigma$ but where an A in x is paired with a T in x', a T with A, a G with C and a C with G. Thus, the string $x =$ `GATTACA` would be paired with $x' =$ `CTAATGA`.

The data-generating process for naturally derived DNA sequences is evolution, which has two main components. The first is "variation with descent," in which some set of processes generate differences in the DNA sequences of an organism and its offspring. Perhaps the most commonly known of these are point mutations, in which single characters change, e.g., `GATTACA`→`CATTACA`. Sequences can also experience deletions (`GATTAC`), duplications (`GATTATTACA`), insertions (`GACATTACA`) or reversals (`GATACAT`) of both single characters or entire subsequences. The second part of evolution is "selection," in which some set of processes prevent the replication of some sequences versus others. This aspect of evolution is sometimes (erroneously) referred to as "survival of the fittest." The real story is more complicated: there are many reasons that lead to some sequences being passed on to offspring despite the fact that they confer little, no or even negative "selective advantage". The good news is that evolution works so long as, on average, in large populations and over long timescales, the genome as a whole confer a net selective advantage. (This is a fairly weak condition, which is partly what makes evolution such a powerful strategy for finding relatively good solutions to relatively hard problems, and it is what underlies the way genetic algorithms work.)

With this background established, we can talk about one of the main areas of computational biology: inferring the history of genetic diversification, given a set of character vectors.[1]

[1]The two main areas in computational biology are sequence alignment and phylogenetic tree reconstruction. Lately, there has been additional emphasis on applying graph theory ideas and techniques to understanding biological networks.

2 Phylogenetic Trees

If we could watch evolution progress forward in time, we would see it produce genuinely new species by taking a single existing species and splitting into two distinct genetic groups.[2] Even without active selection, these groups will "drift" apart genetically over time (a process called "neutral evolution") so long as they do not exchange much genetic material.[3]

If we imagine evolution running backwards in time, however, we would see separate species "merge." The character vector of the merged species is interpreted as the potential character vector of the common ancestor of the two descendent species. A *phylogenetic tree* is the set of all such mergers on a set of n species.[4] That is, if we are given a set of n character vectors $\{X_i\}$, we place these on the leaves of a tree. A phylogenetic tree is a tree that connects these sequences. Internal nodes of this tree are inferred common ancestors. For instance, Figure 1 shows an inferred phylogenetic tree on the three major domains of life: bacteria, archaea and eucarya.

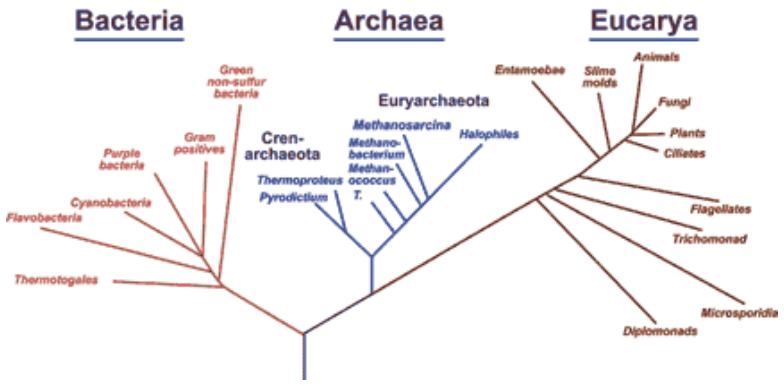


Figure 1: A phylogenetic tree on bacteria, archaea and eucarya. (From Carl Woese's website.)

[2]This process is called "speciation" and can happen quickly or slowly, depending on the mechanism, where the timescales here are somewhere between hundreds of years to hundreds of thousands of years, depending on the specific selection pressures. The typical speciation event is thought to be a very slow process.

[3]Sexual reproduction is one process by which a population of organisms can maintain a kind of "coherence" in the "gene space," and this is a mechanism for two populations to exchange genetic material.

[4]A tree structure assumes that there are no mechanisms for genetic exchange between species, which is not always the case. There are several known inter-species genetic exchange mechanisms. For instance, viruses can transport DNA between species, the "horizontal gene transfer" mechanism allows bacteria and archaea to exchange DNA between individuals, and "hybridization" is a method by which species like plants and animals can exchange DNA by producing a descendant organism with genes drawn from two different parent species, among a few less common others. Within a population, sexual reproduction serves the same function. Thus, the tree structure is simply a rough approximation of the large-scale genetic history of a set of species. A more general approach would use directed acyclic graphs (unless we allow for time travel, which would provide for cycles, among other things).

Many of the species most familiar to us (plants, animals, fungi, etc.) are in the eucarya group. This picture, reproduced from Carl Woese's lab at UIUC, shows one of the better inferred phylogenies for the base of the "tree of life." Figure 2 shows a slightly more artistic (but still data-derived) phylogeny from the Tree of Life project, which shows more structure in the eucarya.

Figure 2: An artistic "tree of life," showing the eucarya in particular. (From `http://tolweb.org/`.)

The goal of phylogenetic tree reconstruction is to build accurate trees that capture the true biological relatedness of organisms. The input data can be any set of "characters", e.g., a DNA sequence (common for many molecular biology studies) or a set of morphological features (common in paleontology, where DNA is not typically available).[5]

Given a set of n character vectors $\{x^{(i)}\}$ as inputs, the output should be the "best" tree T on these sequences, where "best" is determined by some kind of score function that encodes our scientific beliefs and methodological approach. Generally, our observed vectors are placed on the leaves of

[5]Paleontologists identify sets of body structures or patterns or measurements for each species, and code these as a morphometric vector $\vec{\theta}$. The algorithms then work precisely the same, although typically, because the model of feature variation is not known, paleontologists work with maximum parsimony or distance-based approaches instead of the probabilistic methods favored for DNA evolution.

T.[6] There are many different approaches; here, we will delve into the *maximum parsimony* and distance-minimization approaches. We won't cover *maximum likelihood* or probabilistic approaches, which are increasingly popular.[7]

2.1 The maximum parsimony principle

For simplicity, we only consider sequences drawn from a binary alphabet $\Sigma_{01} = \{0, 1\}$. We are given n such binary sequences, each of length ℓ. The maximum parsimony tree on these n sequences is the one that minimizes the number of character *mismatches* between ancestors and descendants.

To formalize this notion, we define a mismatch to be the case where the character x_i in the descendent sequence differs from the character in the same position in its ancestor sequence. The total distance between two sequences is then the number of these mismatches, which we count using the pairwise Hamming distance[8] of the ancestor and descendent strings:

$$d_{\mathrm{H}}(X, Y) = \sum_{i=1}^{\ell} \mathrm{XOR}(x_i, y_i) \ .$$

2.1.1 Big parsimony

The general problem of finding a maximum parsimony tree is then reduced to solving this equation:

$$T_{\mathrm{MP}} = \operatorname*{argmin}_{T \in \mathcal{T}} \left(\sum_{X \in T} d(X, p(X)) \right) \ .$$

That is, we want the tree T out of all trees $\mathcal{T}$ (with n leaves) that minimizes the interior sum; the interior sum ranges over all pairs of descendants X and ancestors $p(X)$ within a given tree T, and requires that each interior node be labeled with the inferred ancestral sequence. The tree that minimizes this sum is the maximum parsimony tree T_{MP}. Note that T_{MP} is not necessarily unique. (What criteria would be necessary to make it unique?) Finding the maximum parsimony tree is sometimes called the "big parsimony" problem and is known to be NP-hard in general (via reduction to vertex cover). Existing techniques largely focus on sampling or searching the entire space of $\mathcal{T}$ trees, which contains $(2n-3)!!$ candidate phylogenies,[9] for good trees.

[6] It is possible to define algorithms that allow for the input sequences to be placed on internal tree nodes, rather than only at the leaves. However, this requires having access to ancestral sequences, which is generally not the case.

[7] These methods have much in common with probabilistic modeling techniques from machine learning, and are thus beyond the scope of this class.

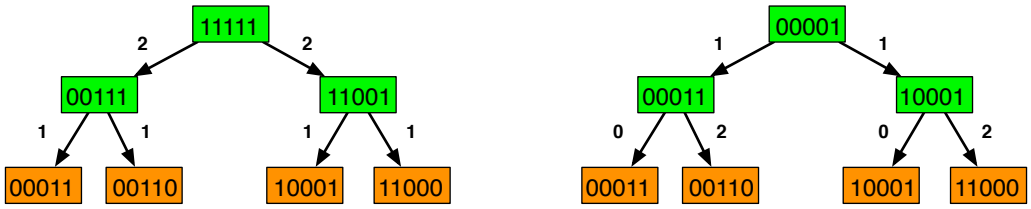
[8] In general, other measures of distance could be used, e.g., Euclidean, Mahalanobis, etc.

[9] The double factorial means this: $n!! = n(n-2)(n-4)...$; you can derive the $(2n-3)!!$ result for unrooted phylogenies, or $(2n-5)!!$ for rooted phylogenies, via induction.

2.1.2 Little parsimony

A more restricted problem, sometimes called the "little parsimony" problem, can be solved in polynomial time. Given a candidate tree T with the n sequences positioned on the leaves, we must derive the ancestral sequences that maximize the parsimony of T.

Consider a small example. Suppose we are given the strings $\{00011\}, \{11000\}, \{10001\}, \{00110\}$. Below are two candidate trees along with potential ancestral states. In each, the given sequences are shown in orange while the potential ancestral states are shown in green, and the ancestor-descendant distance shown as the edge label in the tree. The *weight* or *parsimony score* of a tree is defined as the sum of all edge distances in T. Here, the left tree has tree weight $d_{\mathrm{H}} = 8$ while the right tree has weight $d_{\mathrm{H}} = 6$. Thus, the right-hand tree is the more parsimonious explanation.



To infer the maximum parsimony labeling of the ancestral states, we use a dynamical programming algorithm (due to Fitch, 1971). Fitch's algorithm takes $O(n\ell)$ time in total. It treats each character in the sequence independently, each of which takes $O(n)$ time.[10] By running the basic algorithm on each of the ℓ positions, it derives full ancestral sequences for each internal tree node.

We will not go into the algorithm in detail here (since we haven't covered dynamic programming), but here is a sufficient description. Because we run this algorithm on each position in the sequence independent, it suffices to describe the algorithm for a single character. The algorithm operates in two phases, one that starts at the leaves and moves up to the root, and one that starts at the root and moves down to the leaves.

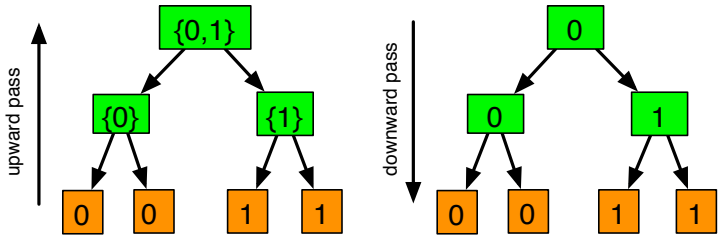
In the first phase, we start at the leaves and work upward to the root assigning each internal node

[10]The running time will take longer if we don't use an efficient data structure for storing the tree, or an efficient algorithm for exploring it. The simplest way is to run a depth-first search algorithm starting at the root. Each time the DFS is about to return from a given internal node to its parent, it does the intersection or union operation described below. DFS takes $O(E)$ time to run, but since there are $O(n)$ internal edges in the tree, it take $O(n)$. If the tree is a balanced binary tree, you could store the tree in an array (see CLRS Chapter 6) and then work backwards from the last element in the array; if it's not balanced, this won't work unless you can find some other way to arrange all the tree nodes at a given depth to occur sequentially in the array.

v a set of letters: if the sets of v's children overlap, then make v's set their intersection, otherwise, make it their union.

In the second phase, we start at the root and works downward to the leaves, assigning a character to each tree node. If v is the root, we choose a character arbitrarily from v's set and recurse on v's children. If some v's set contains the character currently assigned to its parent $p(v)$, then the algorithm assigns that same character to v; otherwise, if the parent character is not in v's set, the algorithm chooses arbitrarily from within v's set and recurses.

The following figure shows an application of Fitch's algorithm on the first character of our example figure above: the left-hand tree shows the upward set-building pass and the right-hand tree shows the downward node-labeling pass. (Is the right-hand tree in the previous figure a maximum parsimony labeling on T? How many such labelings are there?)[11]



2.2 Distance methods

Suppose that our characters are not discrete but are instead drawn from some continuous space, for instance, instead of looking at DNA sequences, suppose we look at some set of morphological (or even linguistic) features of species.[12] This allows us to represent the state of a species as a vector in some $\mathbb{R}^\ell$ space. Unlike the case of binary sequences, in this case, algorithms do exist that can derive both the minimum weight tree and the ancestral states in polynomial time.

Most of these algorithms take a greedy approach by building the tree up by repeatedly merging the pair of states with minimum distance, much like we did in Lecture 9. Because evolution should generate only a small number of changes from ancestor to descendants, and because we assume

[11] Recall that a sequence of length ℓ can be considered a vector in a ℓ-dimensional hypercube. Choosing the ancestral states and the phylogeny can then be seen as the problem of finding a tree within the hypercube, with leaves located at the given n sequences, that has minimum length.

[12] We can always apply a continuous-character algorithm to discrete-character data, but using a continuous-version of the `merge(a,b)` function below will produce continuous-character ancestors. A discrete version of `merge(a,b)` would fix this problem, allowing distance-based methods to be applied to discrete-character data.

these changes are largely independent, given two vectors x, y, these algorithms typically assume that the ancestral state splits the distance between the two descendent states, i.e., the inferred ancestral state is the mid-point of the descendants.

In general, because such an algorithm reduces the number of entities by one each time we iterate, the algorithm will terminate after $n - 1$ steps. What distinguishes different distance-based algorithms is mainly their choice of distance measure, how it chooses which pair of entities to agglomerate, how it chooses the ancestral state, and how it manages all the data.

Most such algorithms maintain a pairwise distance matrix $S_{ij} = d(i, j)$. Given that we choose to replace some pair i, j with their merged state z, we then only need to update $4(n - 1)$ entries in the distance matrix S: we need to delete each of the entries in the ith row and column, and replace each entry in the jth row and column with $d(z, k)$ for all k. Here's pseudocode for a generic distance-based algorithm:

```
Generic-GreedyAgglomerative(X) {
// X = set of vectors
   S = {}
   for all pairs x,y in X {
      S.add(d(x,y),(x,y))
   }
   while S non-empty {
      (a,b) = S.returnElement()
      for each x in X {
         S.remove(d(a,x))
         S.remove(d(b,x))
      }
      remove a and b from X
      c = merge(a,b)
      for all x in X {
         S.add(d(c,x),(c,x))
      }
      add c to X
   }
}
```

If we choose which pair to merge in a greedy fashion, much like we did with the graph clustering algorithm in Lecture 9, then we can replace S with a min-heap data structure. It takes $O(\ell)$ time to compute the distance between some pair of sequences; thus, it takes $O(n^2\ell)$ time to build the initial heap (if we use something like a Fibonacci heap; if we use a binary heap, it takes $O(n^2\ell \log n)$). If we assume that the `merge(a,b)` step takes no more than $O(\ell)$ time, then each pass through

the while loop takes $O(n\ell)$ time (do you see why?), and there are $n-1$ passes; so the greedy agglomerative algorithm runs in $O(n^2\ell)$ time overall.

The particular choice of `merge(a,b)` functions determines which agglomerative tree the algorithm constructs; if we choose the state that splits the difference between a, b, i.e., the equidistant point, then this algorithm is logically equivalent to a single-linkage hierarchical clustering algorithm, which is a popular (but not a very good[13]) technique for clustering spatial data. (The "split-the-difference" choice is not guaranteed to find the ancestral states that minimize tree weight; do you see why?)

2.2.1 UPGMA

A popular distance method is the Unweighted Pair Group Method Using Arithmetic Mean (UPGMA), named by Sokal and Michener (1958), which uses the arithmetic mean as the `merge(a,b)` function. Because of the averaging it does when merging entities, it implicitly assumes a uniformly constant molecular clock, which may not be desirable in some applications.

2.2.2 Neighbor joining

Another variation is the neighbor joining method, which yields a minimum-evolution tree (this is like a maximum parsimony tree, but on a general metric space) that does not assume a uniformly constant molecular clock. Instead of choosing a pair of entities based on their minimum distance, neighbor-joining instead uses the function

$$Q(x,y) = d(x,y) - \frac{1}{n-2}\left(\sum_{i\neq x,y} d(x,i) + \sum_{i\neq x,y} d(y,i)\right)$$

but still chooses the pair with minimum $Q(x,y)$. Intuitively, Q is the distance $d(x,y)$ minus the sum of the average distances from x (and y) to the rest of leaves. Neighbor-joining operates only on pairwise distances; once a pair x, y is chosen, the distance matrix is updated like so: let z denote the new "merged" entity, then for each neighbor i of either x or y, let

$$d(i,z) = \frac{1}{2}\left(d(x,i) + d(y,i) - d(x,y)\right) \quad .$$

3 Unrooted and Consensus Phylogenies

Recall that the usual phylogenetic tree inference setup is that we are provided n entities, each of which has a character vector (e.g., a DNA sequence or a vector of morphometric measurements) of

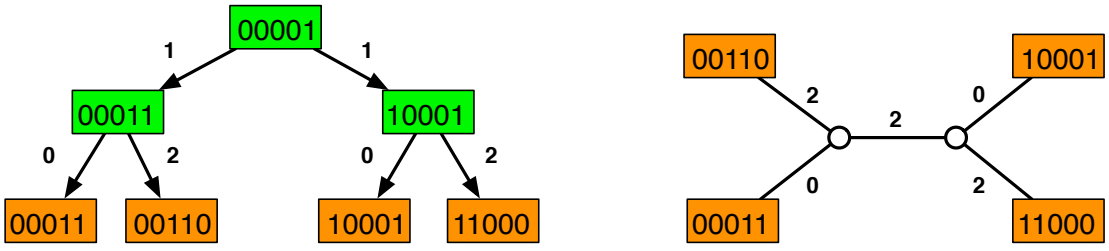
[13] There are many other variations on such hierarchical clustering algorithms, most of which you would encounter in a data mining course. They should be treated with caution however as they are typically not based on any strong statistical or mathematical principles, as far as I can tell, and can perform badly under entirely reasonable conditions.

length ℓ. Our goal is to infer a phylogenetic tree that accurately captures a plausible evolutionary history of the relatedness of these entities. If we assume only vertical transmission of characters (that is, traits only pass from a parent to its offspring) and that new species are derived from older species, then the evolutionary history has the structure of a tree. For n leaves, there are $(2n-5)!!$ rooted binary trees. The different approaches to choosing these trees are largely differences in the way the quality of candidate trees are scored.

3.1 Unrooted phylogenies

The maximum parsimony principle searches this space of trees for the one or set in which the number of character differences between parents and offspring is minimized. Distance-based measures are fairly similar, but typically operate on continuous-valued characters. Both approaches estimate the ancestral (hidden) character vectors in the phylogeny, and explicitly focus on the evolutionary paths from one state to another.

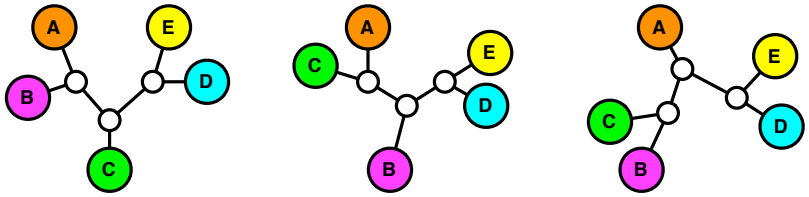
Suppose that we do not particularly care about these ancestral states or the particular evolutionary paths between them. If so, then we only need to consider *unrooted* binary trees, in which all nodes in the tree have degree exactly 3 or 1, and parent-offspring directionality is omitted. In a *rooted* tree, one node, the root, has degree 2, and parents are always closer to the root than their offspring. Consider our running example from above; the left-hand figure shows the maximum parsimony tree, with ancestral states and the evolutionary distance, while the right-hand figure shows an equivalent unrooted phylogeny:



As before, the weight of the tree is the sum of the edge distances, and we can use various score functions to choose which unrooted tree is a better representation of the evolutionary relationships.

3.2 Consensus trees

What happens when we have multiple plausible (low weight) candidate trees for a set of n species? For instance, here are three trees on 5 vertices:



But which tree is the *correct* tree? Or, more generally, which set of evolutionary relationships are common among these three trees? This problem is called the *consensus tree* problem, in which we are given a set of unrooted phylogenetic trees $\{T_i\}$ and our task is to derive from them a single "consensus tree" T_c that includes the common set of evolutionary relationships present in the set. Although the input trees are usually binary trees, the output tree is not necessarily so, i.e., the degree of internal nodes is $k \geq 3$ although leaf nodes are still required to have degree $k = 1$.

In order to build a consensus tree, we need some new formalisms that will help us see how to extract specific evolutionary statements from each tree T_i and then build the consensus tree T_c out of them.

Definition: A *split* on a tree T, denoted $A|B$ is a bipartition of the leaves of $T = (V, E)$ into exactly two non-empty subsets. Further, each edge $(u, v) \in E$ defines such a split: removing (u, v) bipartitions the leaves of T into a set reachable from u through v, and a set reachable from v through u.

Definition: Two splits $A_1|B_1$ and $A_2|B_2$ are *compatible* if at least one of the sets $\{A_1 \cap A_2\}, \{A_1 \cap B_2\}, \{B_1 \cap A_2\}, \{B_1 \cap B_2\}$ is empty.

For the first of the three trees above, its split set is $\{AB, CDE\}, \{ABC, DE\}$ plus the set of trivial partitions $\{X, \{ABCDE\} - X\}$, which separate a single leaf from the rest of the tree. It's straightforward to verify that the set of splits induced by a tree T are pairwise compatible (do you see how?). The converse is also true:

Splits-Equivalence Theorem: Let Ω be a collection of splits. Then, $\Omega \to T$ for some tree T if and only if the splits in Ω are pairwise compatible. Such a tree is unique up to isomorphism.

3.2.1 Tree re-construction

We'll skip the proof of the splits-equivalence theorem and instead simply discuss the constructive algorithm that makes it work, i.e., the algorithm by which we can take a collection $\Omega = \{\omega_1, ..., \omega_k\}$

of pairwise compatible splits and returns a tree T such that the splits of T are Ω.[14]

The idea is the following. Without loss of generality, we can represent a split as a binary vector s_i of length n where $s_i(j) = 1$ if $v_j \in A$ and $s(j) = 0$ otherwise, and, again w.l.o.g. A always denotes the smaller of the two sets, i.e., $|A| < |B|$. (Do you see why this convention is general?) We then initialize a forest F composed of n singletons, one for each leaf vertex (taxon). We'll then iterate over the set of splits s_i and add a single internal tree node to F that serves as the nearest common ancestor in F of all vertices $s_i(j) = 1$. If there are k non-trivial splits in the set, T must contain $k+1$ internal nodes (do you see why?); thus, after adding one node for each of the k splits, we must add one final internal node that connects the remaining components. What remains is to choose an ordering of the splits that guarantees that we build the tree correctly.

This can be done by noting that if we always add an internal node that connects the two smallest components in F, then we are guaranteed to build the tree from the bottom up. This is equivalent to ordering the splits in increasing size of their smaller set: on the ith pass of the algorithm, we add a node u to F and make it the common ancestor of all the subtrees that are full contained in A_i.

We can make this step easier by using a trick similar to that of Kruskal's algorithm in Lecture 8, which assigned each component a "leader." In our case, the leader of a component is the nearest common ancestor. For each subtree that will connect to the new node u, we simply look up its leader and connect it to u.[15]

Here's pseudo-code:

```
Build-Consensus-Tree(Q) {
// Q is the set of splits, e.g., k binary vectors of length n
// {A|B} is a split; let A be the smaller set
// d is an array storing the indices of nearest common ancestors

   sort Q in order of smallest subset A
   for each vertex v in V { d[i] = i }  // initialize nearest common ancestor list
   for i = 1 to n+k { F[i] = -1 }       // initialize tree structure
   for i = 1 to k {                     // for each split
      {A|B} = split with ith smallest subset A
      // add node n+i to F
```

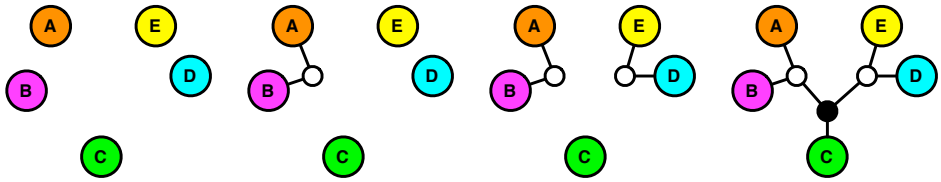
[14] Although we'll only consider unrooted trees here, the split-equivalence theorem and the corresponding algorithms can easily be generalized to rooted trees. The key insight is to recognize that in a rooted tree, each internal node tri-partitions the leaves, and one of those partitions is always "above" that node in the tree; this provides the necessary directionality to choose a root of the tree.

[15] Another split-based algorithm called "tree popping" accomplishes the same goal but using a different approach.

```
      for each v in A {
         F[d[v]] = n+i        // point  NCA (d) at node n+i
         d[v]    = n+i        // update NCA (d) of v
   }}
   for each v {
      F[d[v]] = n+k           // point NCA (d) at node n+k
      d[v]    = n+i           // update NCA (d) of v
}}
```

The running time of `Build-Consensus-Tree()` depends partly on the time to sort the splits; since we're representing them as n-bit binary vectors, this can be done in $O(nk \log k)$ (the additional factor of n is the comparison cost for a n-bit binary number). The time to sort the splits dominates the time to initialize the data structures, which takes $O(n + k)$ time. Finally, the main loop takes $O(nk)$ because we make k passes and each pass potentially requires reading the entire length of an n-bit vector. Thus, the running time is dominated by the time to sort the splits. (This algorithm could be more efficient in the way it updates the tree structure, but asymptotically, the duplicated work doesn't matter.)

Let's apply this algorithm to the splits of the first tree in the figure above; recall that the non-trivial splits were $\{AB, CDE\}, \{ABC, DE\}$, which we can rewrite as $\{11000\}, \{00011\}$ following our convention. Here's the reconstruction process, the black internal node is the node added in the last step of the algorithm:



$s_i = \{11000\}$ $s_i = \{00011\}$

3.2.2 Building the consensus tree

Thus, we can convert any tree T into a set of splits (we haven't described an algorithm for doing so; do you see how to do it? how long does it take?) and any set of splits Ω that satisfies the splits-equivalence theorem can be converted back into a tree.[16] (Note that our function `Build-Consensus-Tree()` does not verify that its input satisfies the requirements—what will happen if we pass the function a bad input?) This suggests an algorithm for deriving a consensus tree:

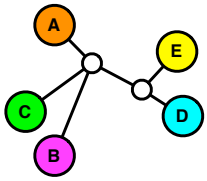
[16]The splits-equivalence theorem is sometimes also called the "four gametes condition."

convert each tree T_i into its set of splits and combine these into a common pool; then, extract the subset of splits with high frequency and construct a new tree T_c composed only of those splits. What remains however to decide which splits to extract and to guarantee that the set we extract satisfy the splits-equivalence theorem. There are, of course, several approaches, depending on what kind of consensus you want.

The *strict consensus* approach extracts only those splits that occur in every tree T_i while the *majority consensus* approach extracts all splits whose frequency $f_{A|B} \geq 1/2$.[17] Both of these approaches fundamentally construct a histogram on the set of splits, and thus takes time $O(nL \log(nL))$ for L trees on n leaves. If we consider our three trees above, we have a normalized split histogram that looks like this (again, following our labeling convention):

$$\frac{1}{3} \times \{11000\},\ \frac{1}{3} \times \{10100\},\ \frac{1}{3} \times \{01100\},\ \frac{3}{3} \times \{00011\}$$

Thus, the majority consensus tree and the strict consensus tree in this case are the same: only one of the splits appears in at least half of the input trees and it also appears in all of them. Thus, the consensus tree omits any structure on the nodes A, B, C because the input trees disagree on their proper evolutionary relationship.



4 For Next Time

1. Optimization algorithms
2. Reminder: your solutions to PS2 due Monday (2/25), via email by 11:59pm

[17] Naturally, there are additional approaches, e.g., the "semi-strict" approach, which retains all splits from the strict consensus along with any splits that are not incompatible with the strict consensus and each other.