Growing Phylogenetic Trees in R

Erik S. Wright

August 31, 2026

Contents

1 Introduction **2**

2 Performance Considerations **2**

3 Preparing the Input Data **3**

4 Choosing a Method and Model of Evolution **5**
4.1 Minimum Evolution (models of evolution) 6
4.2 Maximum Likelihood (models of evolution) 6
4.3 Maximum Parsimony (models of evolution) 6
4.4 Treatment of gaps in models of evolution 6
4.5 Missing models of evolution 7

5 Building Minimum Evolution Phylogenetic Trees **7**

6 Building Maximum Likelihood Phylogenetic Trees **8**
6.1 Plotting Branch Support Values 10

7 Building Maximum Parsimony Phylogenetic Trees **12**
7.1 Ancestral State Reconstruction 14

8 Bootstrap and Jackknife Branch Supports **16**

9 More Examples of Manipulating Dendrograms **19**

10 Inspecting the Inputs and Outputs **21**

11 Exporting the Tree **25**

12 Generating a Summary Tree **25**
12.1 Concordance Vector Analysis 30

13 Conclusion **34**

14 Session Information **34**

1 Introduction

This document describes how to grow phylogenetic trees using the `Treeline` [19] and `Zipline` functions in the DECIPHER package. `Treeline` takes as input a set of aligned nucleotide or amino acid sequences and returns a phylogenetic tree (i.e., *dendrogram* object) as output. `Zipline` takes a list of trees and produces a single summary tree. The overarching idea is to run `Treeline` on a set of multiple sequence alignments corresponding to genes from the same organisms and then give the output gene trees to `Zipline` for summarization into a species tree.

Why are the functions called `Treeline` and `Zipline`? The goal of `Treeline` is to find the best tree according to an optimality criterion. There are often many trees near the optimum. Therefore, `Treeline` searches along the metaphorical treeline of high scoring trees to find the best one. `Zipline` is named for its goal of zippy (fast) and accurate summary trees by using linear regression to connect trees on different treelines.

Why use `Treeline` and `Zipline` versus other programs? Both functions are designed to return excellent phylogenetic trees with minimal user intervention. Many tree building programs have a large set of complex options for niche applications. In contrast, `Treeline` and `Zipline` simply build great trees by default. `Treeline`'s unified optimization strategy also makes it easy to compare trees optimized under different optimality criteria. This vignette is intended to get you started and introduce additional options/functions that might be useful.

This vignette focuses on optimizing balanced minimum evolution (ME), maximum likelihood (ML), and maximum parsimony (MP) phylogenetic trees starting from sequences, followed by constructing a species tree from a large set of trees. Treeline uses a strategy of multi-start optimization followed by hill-climbing and grafting. Since Treeline is a stochastic optimizer, it optimizes many trees to prevent chance from influencing the final result. With some luck it'll find the highest scoring tree!

2 Performance Considerations

Finding an optimal tree is no easy feat. `Treeline` systematically optimizes many candidate trees before returning the best one. This takes time, but there are things you can do to make it go faster.

- Only use the sequences you need: `Treeline`'s optimization runtime scales approximately quadratically with the number of sequences. Hence, limiting the number of sequences is a worthwhile consideration. In particular, always eliminate redundant sequences, as shown in the example below.

- Compile with OpenMP support: Significant speed-ups can be achieved with multi-threading using OpenMP, particularly for ML and MP *method*s. See the "Getting Started DECIPHERing" vignette for how to enable OpenMP on your computer. Then you can set the argument `processors=NULL` and `Treeline` will use all available processors.

- Compile for SIMD support: `Treeline` is configured to make use of SIMD operations, which are available on most processors. The easiest way to enable SIMD is to add a line with "CFLAGS += -O3 -march=native" to your ∼/.R/Makevars text file. Then, after recompiling, there should be a modest speed-up on systems with SIMD support. Note that enabling SIMD makes the compiled code non-portable, so the code always needs to be compiled for the specific hardware being used.

- Set a timeout: The *maxTime* argument specifies the (approximate) maximum number of hours you are willing to let `Treeline` run. If you are concerned about the code running for too long then simply set this argument.

- Limit iterations: `Treeline` will converge after *minIterations* when the score is expected to change less than *tolerance* per iteration, unless *maxIterations* is met before convergence. A reasonable way to converge early is to set *minIterations* and *maxIterations* to lower values. There is evidence supporting the notion that exhaustive searching is unlikely to result in a significantly more correct tree [9, 16], even as the score continues to improve.

- For ML, choose a model: Automatic model selection is a useful feature, but frequently this time-consuming step can be skipped. For many nucleotide sequences the `"GTR+G4"` model will be automatically selected. Typical

amino acid sequences will pick the "LG+G4" or "WAG+G4" models, unless the sequences are from a particular origin (e.g., mitochondria). Pre-selecting a subset of the available MODELS and supplying this as the *model* argument can save time.

Treeline is a stochastic optimizer, so it will continue searching the space of possible trees until convergence. It is possible to find the best tree on the first iteration, but most of the time additional iterations will yield a better scoring tree. If you are feeling unlucky, you can simply increase the number of iterations (i.e., *minIterations* and *maxIterations*) to ensure a high scoring tree is found. There is a decreasing marginal return to more iterations, and it's probably not worth searching (almost) endlessly for a slightly better tree. Treeline's default settings are designed to balance runtime versus the reward of better scoring trees.

To largely remove luck from the equation, it is best to use an input alignment with many more sites than sequences. A good rule of thumb is that the input alignment should have a *width* at least four times its *length* for nucleotides and two times for amino acids. Otherwise the optimization space may have a rugged topology with many similarly scoring peaks, and it might be impossible to find the treeline of highest scoring trees. More importantly than the difficulty of optimization is the lack of robustness, which indicates weak confidence in the resulting tree. A good way to verify robustness is to use repeated resampling (see Section 7.1).

Previous work has revealed a few general principles that lead to more accurate trees:

- High-quality alignments are essential for good trees. It is intuitive that trimming ambiguously aligned regions would lead to better trees, but studies have shown stringent trimming has a negative effect and gentle trimming has a minimal positive effect [15, 17]. This is because it is very difficult to remove false positive homologies from an alignment without removing far more true positive homologies. If desired, poorly aligned sites and problematic gap regions can be mitigated with MaskAlignment and StaggerAlignment.
- Use of orthologous genes is necessary if the goal is to infer the species tree rather than the gene tree. Recall orthologs split at speciation events, whereas paralogs may have split further in the past.
- Dense taxon sampling generally results in better trees by breaking up longer branches and helping to avoid saturation effects that spuriously attract long branches. Adding diverse sequences to the alignment is one of the most straightforward ways to dependably improve trees [20].
- Empirical results indicate the minimum evolution objective is superior to maximum likelihood and maximum parsimony objectives [4, 13]. This effect may be lessened in the regime of very long input sequences, but minimum evolution still tends to be at least as good.
- Specifying a model of evolution is important for datasets that have undergone many mutations, because models account for repeated substitutions at the same site. However, the choice of evolutionary model generally has less of an effect than the choice of optimization objective (i.e., minimum evolution versus alternatives).

3 Preparing the Input Data

Treeline takes as input a multiple sequence alignment and/or a distance matrix. All distance-based methods (including ME) only require specification of *myDistMatrix* but will generate a distance matrix using DistanceMatrix if *myXStringSet* is provided instead. The character-based methods (i.e., ML and MP) require a multiple sequence alignment and will generate a distance matrix to construct the first candidate tree unless one is provided as a starting point.

Multiple sequence alignments can be constructed from a set of (unaligned) sequences using AlignSeqs or related functions. Treeline will optimize trees for amino acid (i.e., AAStringSet) or nucleotide (i.e., DNAStringSet or RNAStringSet) sequences. For coding sequences, it is intuitive to assume that nucleotide data would better resolve close taxa, whereas amino acid data would be preferable to determine the branching order of deep taxa. However, studies have shown that nucleotide data is adequate for determining distant relationships [7]. A good bet is to use nucleotide sequences with the "ME" *method*, possibly specifying a model (e.g., "TN93+F") that corrects for multiple substitutions per site.

Here, we are going to use a set of sequences that is included with DECIPHER. These sequences are from the internal transcribed spacer (ITS) between the 16S and 23S ribosomal RNA genes in several *Streptomyces* species. To avoid letting the result come down to good old-fashioned luck, it is always best to compare multiple trees optimized for different objectives (ME, ML, and MP) and alternative models of evolution. `Treeline` is designed to facilitate this type of comparison, ideally across multiple loci.

```
> library(DECIPHER)
> # specify the path to your sequence file:
> fas <- "<<path to FASTA file>>"
> # OR find the example sequence file used in this tutorial:
> fas <- system.file("extdata", "Streptomyces_ITS_aligned.fas", package="DECIPHER")
> seqs <- readDNAStringSet(fas) # use readAAStringSet for amino acid sequences
> seqs # the aligned sequences
DNAStringSet object of length 88:
     width sequence                                          names
 [1]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC supercont3.1 of S...
 [2]   627 NNNNCACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC supercont3.1 of S...
 [3]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC supercont1.1 of S...
 [4]   627 CGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC supercont1.1 of S...
 [5]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC supercont1.1 of S...
 ...   ... ...
[84]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC gi|297189896|ref|...
[85]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTGTCCGAATGGGGAAACC gi|224581106|ref|...
[86]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTGTCCGAATGGGGAAACC gi|224581106|ref|...
[87]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTGTCCGAATGGGGAAACC gi|224581106|ref|...
[88]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC gi|224581108|ref|...
```

Many of these sequences are redundant or from the same genome. We can de-replicate the sequences to accelerate tree building and simplify analyses:

```
> seqs <- unique(seqs) # remove duplicated sequences
> ns <- gsub("^.*Streptomyces( subsp\\. | sp\\. | | sp_)([^ ]+).*$",
         "\\2",
         names(seqs))
> names(seqs) <- ns # name by species (or any other preferred names)
> seqs <- seqs[!duplicated(ns)] # remove redundant sequences from the same species
> seqs
DNAStringSet object of length 19:
     width sequence                                          names
 [1]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC albus
 [2]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC clavuligerus
 [3]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTGTCCGAATGGGGAAACC ghanaensis
 [4]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC griseoflavus
 [5]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTGTCCGAATGGGGAAACC lividans
 ...   ... ...
[15]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTGTCCGAATGGGGAAACC cattleya
[16]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC bingchenggensis
[17]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC avermitilis
[18]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTGTCCGAATGGGGAAACC C
[19]   627 TGTACACACCGCCCGTCA-CGTC...GGGGTGTCCGAATGGGGAAACC Tu6071
```

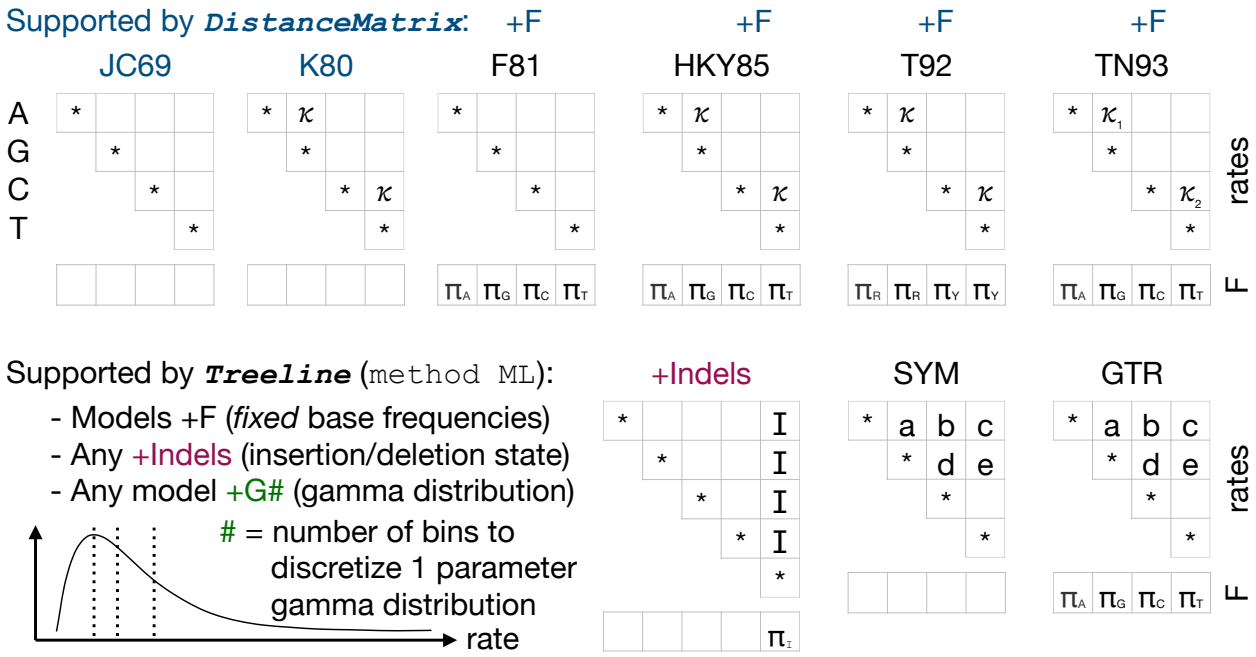


Figure 1: Free rates and frequencies in nucleotide models.

4 Choosing a Method and Model of Evolution

Before choosing a `model` of evolution, it is necessary to choose a *method* for optimizing the tree. The default *method* is `"ME"` because it is fast and performs best on empirical datasets [4, 13]. The ME *method* accepts *myDistMatrix* as input, or *myXStringSet* can be given with or without a *model* to use with `DistanceMatrix` for building a distance matrix. For maximum likelihood, set *method* to `"ML"`, which requires one or more *model*s of sequence evolution. For maximum parsimony, set *method* to `"MP"` and (ideally) specify a *costMatrix*.

`Treeline` supports many `MODELS` of evolution. In many cases, these `MODELS` can be extended by appending the model with `"+F"`, `"+G#"`, or `"+Indels"`. Here is the list of built-in `MODELS`:

```
> MODELS
$Nucleotide
[1] "JC69"   "K80"    "F81"    "HKY85" "T92"    "TN93"   "SYM"    "GTR"

$Protein
 [1] "AB"             "BLOSUM62"       "cpREV"          "cpREV64"
 [5] "Dayhoff"        "DCMut-Dayhoff" "DCMut-JTT"      "DEN"
 [9] "FLAVI"          "FLU"            "gcpREV"         "HIVb"
[13] "HIVw"           "JTT"            "LG"             "MtArt"
[17] "mtDeu"          "mtInv"          "mtMam"          "mtMet"
[21] "mtOrt"          "mtREV"          "mtVer"          "MtZoa"
[25] "PMB"            "Q.bird"         "Q.insect"       "Q.LG"
[29] "Q.mammal"       "Q.pfam"         "Q.plant"        "Q.yeast"
[33] "rtREV"          "stmtREV"        "VT"             "WAG"
[37] "WAGstar"
```

The nucleotide models each have different numbers of free parameters (Fig. 1). The `MODELS` with few free

parameters are supported by `DistanceMatrix` and, therefore, *method* `"ME"`. This is because distance for few-parameter models can be analytically estimated from the sequences with relatively little error. High-parameter models, such as `"GTR"`, must be optimized and are only supported by `Treeline` *method* `"ML"`. All base built-in amino acid `MODELS` have no free parameters and are supported by `DistanceMatrix` and `Treeline`. See `?MODELS` for more information.

4.1 Minimum Evolution (models of evolution)

Empirical benchmarks suggest ME with p-distance (i.e., length-normalized Hamming distance) results in impressively accurate trees, given p-distance does not account for repeated substitutions at the same site. Therefore, this is the default configuration when *myXStringSet* is supplied without `myDistMatrix`, which returns branch lengths in units of differences per site. If you would prefer to have branch lengths in units of substitutions per site, it is possible correct for multiple substitutions (e.g., *A* to *G* to *C* then back to *A*) by setting *model* to any of the `MODELS` of evolution supported by `DistanceMatrix` (e.g., `"JC"` or `"F81+F"` for nucleotides, and `"WAG"` or `"WAG+F"` for amino acids). See Figure 1 for a list of models supported by `DistanceMatrix`. When *method* is `"ME"`, maximum control is gained by supplying *myDistMatrix*, which can be calculated with `DistanceMatrix` beforehand.

For example, a standard *model* to select for nucleotide sequences would be `"TN93+F"` and for amino acid sequences would be `"WAG"`. These models return trees with branch lengths in units of substitutions per site.

4.2 Maximum Likelihood (models of evolution)

For ML trees, `Treeline` will automatically select an appropriate *model* according to Akaike information criterion (by default). It is possible to choose specific models (e.g., `model="GTR+G4"`) to limit the possible selections and test your luck with fewer options. There is evidence that the choice of nucleotide model does not substantially alter tree accuracy [1, 5, 12], and picking a medium complexity model every time is a reasonable decision. All *model*s can be used with empirical letter frequencies (i.e., by appending with `+F`) and/or gamma rate variation across sites (e.g., `+G4`). Note `Treeline` supports two discretizations of the gamma distribution: the default of equal binning, or the Laguerre quadrature if *quadrature* is set to `TRUE`. The former will give likelihoods comparable with other programs, but the latter is more accurate at representing the gamma distribution with limited bins.

For example, a standard *model* to select for nucleotide sequences would be `"GTR+G4+F"` and for amino acid sequences would be `"WAG+G4"`, with *quadrature* set to `TRUE` in both cases. These models return trees with branch lengths in units of substitutions per site.

4.3 Maximum Parsimony (models of evolution)

For MP trees, the best results are typically obtained by providing a *costMatrix* rather than relying on the default of binary costs. The choice of *costMatrix* is up to you, and several rational options are provided in the examples section of the `Treeline` manual page (see `?Treeline`). A systematic approach to deriving a substitution matrix is also provided as an example below.

4.4 Treatment of gaps in models of evolution

The standard models of evolution described above all ignore gap ("-" and ".") characters representing insertions or deletions (indels). But you're in luck — `Treeline` has the ability to incorporate gaps into all *method*s. For ME trees, `DistanceMatrix` allows gaps to be penalized in p-distance or added to any distance corrected for multiple substitutions per site. You can either provide a model with `"+Indels"` in `Treeline`, or specify *myDistMatrix* after setting *penalizeGapLetterMatches* to `TRUE` or `NA` (see `?DistanceMatrix`). For ML trees, gaps can be added into any model of evolution as an additional state by specifying a model `"+Indels"`, which adds two free parameters (Fig. 1). Incorporating gaps results in branch lengths in units of *changes* per site, since both substitutions and indels contribute to distance. For MP trees, gaps can be added as another state in the *costMatrix*. As luck would have it,

incorporating gaps tends to result in *slightly* better trees on empirical datasets, although the average improvement is typically very small.

4.5 Missing models of evolution

There exists a plethora of published models representing sequence evolution, not all of which are supported. Two notably absent `MODELS` are invariant and codon models. Models with a fraction of invariant sites, often represented as `+I`, are biologically unrealistic and effectively captured by gamma rate variation across sites (e.g., `+G4`). Including both `+I` and `+G` creates unnecessary over-parameterization. Similarly, empirical codon models are not offered because they contain too many (2080) free parameters. It is hard to believe a single codon substitution matrix can adequately capture variation in codon usage across organisms, when it is known generic amino acid matrices (210 parameters) inadequately represent many proteins. Nucleotide models have the advantage that their relatively low number of parameters can be estimated from the data, and there is evidence nucleotide models can even be used for distant relationships where amino acid models were traditionally thought to have an advantage [7].

5 Building Minimum Evolution Phylogenetic Trees

Now, it's time to try our luck at finding the most likely tree. We will use the default settings, which return a minimum evolution tree based on a p-distance matrix. Simply specify a *model* to correct for multiple substitutions (e.g., `"TN93+F"` or `"WAG"`).

Since `Treeline` is a stochastic optimizer, it is critical to always set the random number seed for reproducibility. This will result in the same sequence of random numbers every time and, therefore, reproducibility. You can pick any lucky number, and if you ever wonder how much you pushed your luck, you can try running again from a different random number seed to see how much the result came down to luck of the draw. Note that setting a time limit, as done below with *maxTime*, negates the purpose of setting a seed – never set a time limit if reproducibility is required or you'll have no such luck.

```
> set.seed(123) # set the random number seed
> treeME <- Treeline(seqs, verbose=FALSE, processors=1)
> set.seed(NULL) # reset the seed
```

`Treeline` returns an object of class *dendrogram* that stores the tree in a nested list structure. We can take an initial look at the tree and its attributes.

```
> treeME
'dendrogram' with 2 branches and 19 members total, at height 0.1533688
> attributes(treeME)
$members
[1] 19

$height
[1] 0.1533688

$class
[1] "dendrogram"

$method
[1] "ME"

$score
[1] 1.152586
```

```
$midpoint
[1] 10.91406
> str(treeME, max.level=4)
--[dendrogram w/ 2 branches and 19 members at h = 0.153]
  |--[dendrogram w/ 2 branches and 18 members at h = 0.11]
  |  |--leaf "cattleya" (h= 0.0376  )
  |  `--[dendrogram w/ 2 branches and 17 members at h = 0.0931]
  |     |--[dendrogram w/ 2 branches and 7 members at h = 0.087]
  |     |  |--[dendrogram w/ 2 branches and 5 members at h = 0.0787] ..
  |     |  `--[dendrogram w/ 2 branches and 2 members at h = 0.0686] ..
  |     `--[dendrogram w/ 2 branches and 10 members at h = 0.0849]
  |        |--[dendrogram w/ 2 branches and 2 members at h = 0.0779] ..
  |        `--[dendrogram w/ 2 branches and 8 members at h = 0.0764] ..
  `--leaf "AA4"
```

6 Building Maximum Likelihood Phylogenetic Trees

For the next example, we will grow a maximum likelihood phylogenetic tree, which is the most computationally demanding optimization objective that is supported. We will set a stringent time limit (0.01 hours) to make this example faster, although longer time limits (e.g., 24 hours) are advised because setting very short time limits leaves the result partly up to luck. It is worth reiterating that setting a time limit negates the purpose of setting a seed, so omitting a time limit is required for reproducibility.

```
> set.seed(123) # set the random number seed
> tree <- Treeline(seqs,
          method="ML",
          model="GTR+G4",
          maxTime=0.01,
          verbose=FALSE,
          processors=1)
> set.seed(NULL) # reset the seed
> plot(tree)
```

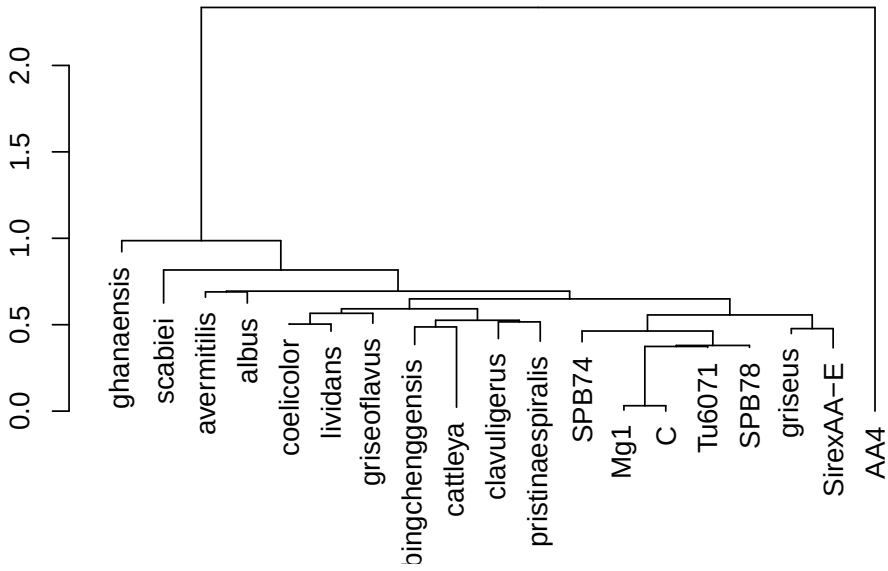


Figure 2: ML tree showing the relationships between *Streptomyces* species.

6.1 Plotting Branch Support Values

Maybe it was just beginner's luck, but we already have a good looking tree! `Treeline` automatically returns a variety of information about the tree that can be accessed with the `attributes` and `attr` functions:

```
> attr(tree, "members") # number of leaves below this (root) node
[1] 19
> attr(tree, "height") # height of the node (in this case, the midpoint root)
[1] 2.335526
> attr(tree, "score") # best score (in this case, the -LnL)
[1] 4362.251
> attr(tree, "model") # either the specified or automatically select transition model
[1] "GTR+G4"
> attr(tree, "parameters") # the free model parameters (or NA if unoptimized)
    FreqA     FreqC     FreqG     FreqT     FreqI       A/G       C/T       A/C
0.1765648 0.2431969 0.3456666        NA        NA 3.2658511 2.9258023 0.7018299
      A/T       C/G    Indels     alpha
1.0879400 0.5788045        NA 0.1910076
> attr(tree, "midpoint") # center of the edge (for plotting)
[1] 9.950195
```

The tree is rooted at its midpoint by default. For maximum likelihood trees, all internal nodes include aBayes branch support values [2]. These are given as probabilities that can be used in plotting on top of each edge. We can also italicize the leaf labels (species names) and add a scale bar.

```
> plot(dendrapply(tree,
        function(x) {
                s <- attr(x, "probability") # choose "probability" (aBayes)
                if (!is.null(s) && !is.na(s)) {
                        s <- formatC(as.numeric(s), digits=2, format="f")
                        attr(x, "edgetext") <- paste(s, "\n")
                }
                attr(x, "edgePar") <- list(p.col=NA, p.border=NA, t.col="#CC55AA", t.ce
                if (is.leaf(x))
                        attr(x, "nodePar") <- list(lab.font=3, pch=NA)
                x
        }),
        horiz=TRUE,
        yaxt='n')
> # add a scale bar (placed manually)
> arrows(2, 0, 2.4, 0, code=3, angle=90, len=0.05, xpd=TRUE)
> text(2.2, 0, "0.4 subs./site", pos=3, xpd=TRUE)
```

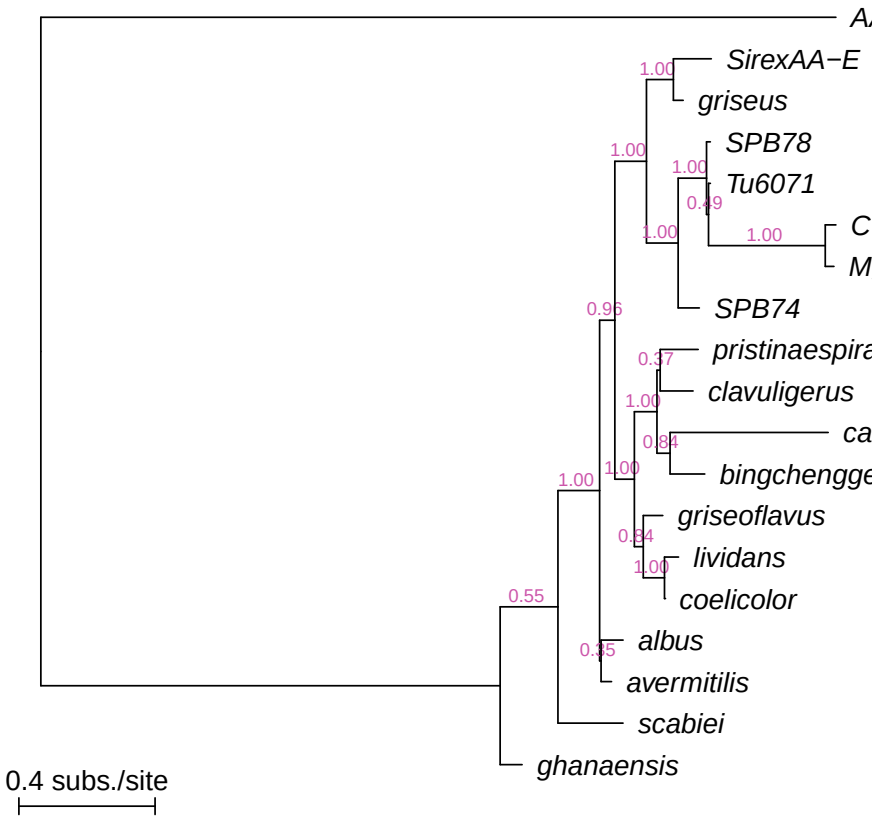


Figure 3: ML tree with aBayes probabilities at each internal node.

7 Building Maximum Parsimony Phylogenetic Trees

While ME and ML trees are based on models of evolution, MP relies on a cost matrix giving the penalty for switching characters along a branch. The default *costMatrix* is binary, which is biologically implausible and may invite bad luck. Hence, we will construct a binary tree and use the result to infer a more appropriate *costMatrix*.

```
> set.seed(123) # set the random number seed
> tree_UniformCosts <- Treeline(seqs,
         method="MP",
         reconstruct=TRUE,
         verbose=FALSE,
         processors=1)
> set.seed(NULL) # reset the seed
```

The literature is surprisingly devoid of methods for constructing optimal cost matrices, but that won't stop us from giving it a try. Since we set *reconstruct* to `TRUE`, `Treeline` output the state transition matrix as an attribute of the tree. We will use this to make our own luck by deriving a more biologically plausible *costMatrix*. It is apparent that transitions are more frequent than transversions and, therefore, are presumably less costly.

```
> mat <- attr(tree_UniformCosts, "transitions")
> mat # count of state transitions
    A  C   G   T
A   0 49 107  55
C  26  0  69 151
G 107 63   0  80
T  43 81  51   0
> mat <- mat + t(mat) # make symmetric
> mat <- mat/(sum(mat)/2) # normalize
> mat <- -log2(mat) # convert to bits
> diag(mat) <- 0 # reset diagonal
> mat # a derived cost matrix
         A        C        G        T
A 0.000000 3.555816 2.043168 3.169925
C 3.555816 0.000000 2.740241 1.926654
G 2.043168 2.740241 0.000000 2.751212
T 3.169925 1.926654 2.751212 0.000000
```

Now we can compare the two trees to see whether specifying a non-uniform cost matrix made a difference. We will highlight different partitions between the trees with dashed edges. Ideally the two tree topologies would be identical, implying the tree is robust to the specification of the cost matrix. The fact that this isn't the case suggests the cost matrix has a substantial influence over the tree, as might be expected. Note the scale of the two trees is different, because branch lengths are in units of average cost (per site) according to each *costMatrix*.

```
> set.seed(123) # set the random number seed
> tree_NonUniformCosts <- Treeline(seqs,
        method="MP",
        costMatrix=mat,
        reconstruct=TRUE,
        verbose=FALSE,
        processors=1)
> set.seed(NULL) # reset the seed
> splits <- function(x) {
        y <- sapply(x, function(x) paste(sort(unlist(x)), collapse=" "))
        if (!is.leaf(x))
                y <- c(y, splits(x[[1]]), splits(x[[2]]))
        y
 }
> splits_UniformCosts <- splits(tree_UniformCosts)
> splits_NonUniformCosts <- splits(tree_NonUniformCosts)
> dashEdges <- function(x, splits) {
        y <- paste(sort(unlist(x)), collapse=" ")
        if (!y %in% splits)
                attr(x, "edgePar") <- list(lty=2)
        x
 }
> layout(matrix(1:2, nrow=1))
> plot(dendrapply(tree_UniformCosts, dashEdges, splits_NonUniformCosts),
        main="MP uniform costs")
> plot(dendrapply(tree_NonUniformCosts, dashEdges, splits_UniformCosts),
        main="MP non-uniform costs")
```

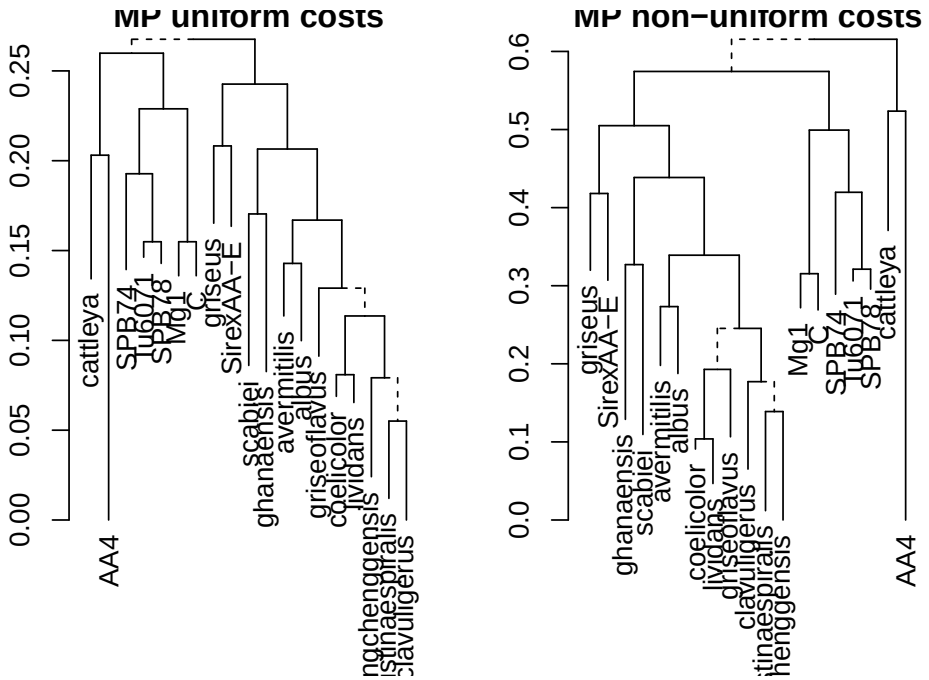


Figure 4: Comparison of MP trees built with different cost matrices.

7.1 Ancestral State Reconstruction

We're in luck —when *reconstruct* is `TRUE`, `Treeline` infers ancestors for each internal node on the tree [6]. These character states can be used by the function `MapCharacters` to determine state transitions along each edge of the tree. This information enables us to plot the total number of substitutions occurring along each edge. The state transitions can be accessed along each edge by querying a new "change" attribute.

```
> new_tree <- MapCharacters(tree_NonUniformCosts, labelEdges=TRUE)
> plot(new_tree, edgePar=list(p.col=NA, p.border=NA, t.col="#55CC99", t.cex=0.7))
> attr(new_tree[[1]], "change") # state changes on first branch left of (virtual) root
 [1] "A168T" "A177G" "A208T" "C269T" "A274G" "A275C" "A308G" "C333G" "A371T"
[10] "A375G" "C386G" "C395T" "A403G" "G405T" "A406G" "C417T" "G432T" "A453G"
[19] "G455T" "G598T"
```

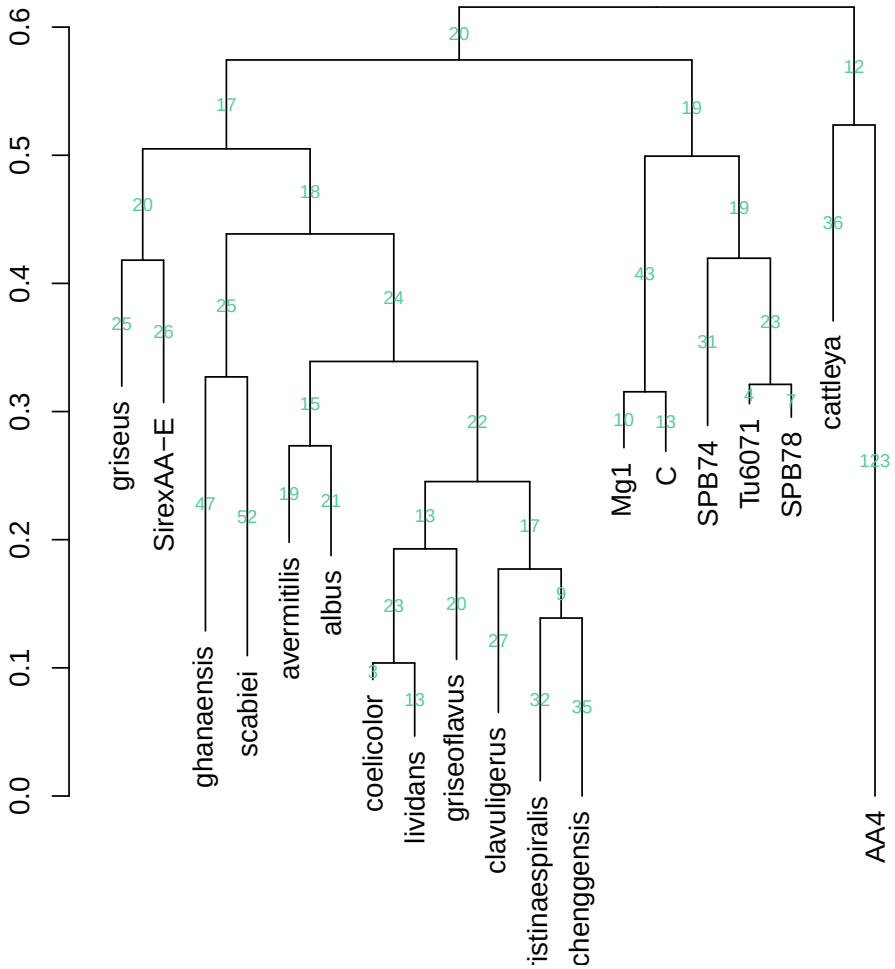


Figure 5: Edges labeled with the number of state transitions.

8 Bootstrap and Jackknife Branch Supports

Each branch in a phylogenetic tree partitions the leaves into two sets. We can use non-parametric subsampling to assign a support value (confidence) to that partitioning. High values imply the sequences reliably support a given partition, otherwise the existence of a branch is called into question. The best way to quantify branch support is by using multiple loci, i.e., how often summary trees constructed from different sets of genes yield the same partition (see Section 11). However, we can also subsample sites (columns) of an alignment with (bootstrap) or without (jackknife) replacement to quantify support.

Many different single-branch methods have been developed for estimating local branch supports. These include the aBayes probabilities [3] that are automatically included on maximum likelihood trees output by `Treeline`. Single-branch methods fail to account for the dependencies among surrounding branches that undermine the support of a branch. For this reason, non-parametric subsampling is preferable to obtain more precise estimates of branch support. The main problem with subsampling is that it is slow, since the tree needs to be rebuilt in every iteration. Fast approximations have been developed, but these can be poorly calibrated to the actual bootstrap value at each branch.

Site-level subsampling (bootstrap/jackknife) has many different flavors, which is why it is not directly incorporated into the `Treeline` function as it is for multi-locus bootstrapping in the `Zipline` function. The code below can be tailored to different applications, such as bootstrapping with non-uniform site sampling or different models of evolution in each replicate. Keeping bootstrapping outside the `Treeline` function also facilitates the splitting of separate bootstrap replicates across multiple computers to accelerate the computation of support values. The basic idea is to record how often a branch in the original tree was supported and then use that information to annotate the tree.

We've hit the jackpot – the code below implements standard bootstrapping and jackknife subsampling, which are both applicable to trees built under any optimization objective or model of evolution. The value of `subsamp` controls whether the standard bootstrap (`1`) or jackknife (`< 1`) is performed. Recent empirical work suggests that jackknife subsampling of 40% of sites results in the best calibrated and predictive support values on average, so this is the default method used below. This calibration allows for support values to be directly interpreted as percent repeatability of a branch in an equivalent data matrix.

```
> reps <- 100 # number of replicates (typically >= 100)
> subsamp <- 0.4 # proportion of sites sampled (< 1 is jackknife, 1 is bootstrap)
> # make a tree from the full alignment
> tree1 <- Treeline(seqs, verbose=FALSE, processors=1)
> partitions <- function(x) { # record splits
          if (is.leaf(x))
                  return(NULL)
          x0 <- sort(unlist(x))
          if (length(x0) > N/2 ||
                  (length(x0) == N/2 && x0[1] != 1))
                  x0 <- seq_len(N)[-x0]
          if (length(x0) > 1) {
                  x0 <- paste(x0, collapse=" ")
          } else {
                  x0 <- NULL
          }
          x1 <- partitions(x[[1]])
          x2 <- partitions(x[[2]])
          return(list(x0, x1, x2))
 }
> # initialize values
> L <- width(seqs)[1]
> N <- length(seqs)
> counts <- unique(unlist(partitions(tree1)))
```

```
> counts <- setNames(integer(length(counts)), counts)
> pBar <- txtProgressBar()
> for (i in seq_len(reps)) {
        # sample positions
        if (subsamp == 1) { # bootstrap
                r <- sample(L, replace=TRUE) # with replacement
        } else if (subsamp > 0 && subsamp < 1) { # jackkife
                r <- sample(L, ceiling(subsamp*L)) # without replacement
        } else {
                stop("Invalid value of subsamp.")
        }

        # generate bootstrapped alignment
        at <- IRanges(r, width=1)
        seqs2 <- extractAt(seqs, at)
        seqs2 <- lapply(seqs2, unlist)
        seqs2 <- as(seqs2, class(seqs))

        # recompute tree and record partitions
        temp <- Treeline(seqs2, verbose=FALSE)
        m <- match(unlist(partitions(temp)), names(counts))
        m <- m[!is.na(m)]
        counts[m] <- counts[m] + 1

        setTxtProgressBar(pBar, i/reps)
 }
================================================================================
> close(pBar)
```

Now we can label edges by the percentage of times each partition appeared among the bootstrap replicates. Since we used `blocks = 2`, we can have confidence the bootstrap values accurately represent the percent repeatability justified by the data.

```
> counts <- round(100/reps*counts) # convert to percentage
> labelEdges <- function(x) {
        if (!is.leaf(x)) {
                x0 <- sort(unlist(x))
                if (length(x0) > N/2 ||
                        (length(x0) == N/2 && x0[1] != 1))
                        x0 <- seq_len(N)[-x0]
                if (length(x0) > 1) {
                        x0 <- paste(x0, collapse=" ")
                        attr(x, "edgetext") <- counts[x0]
                }
        }
        return(x)
 }
> tree2 <- dendrapply(tree1, labelEdges)
> attr(tree2, "edgetext") <- NULL # remove text from (virtual) root branch
> plot(tree2, edgePar=list(t.cex=0.5), nodePar=list(lab.cex=0.7, pch=NA))
```

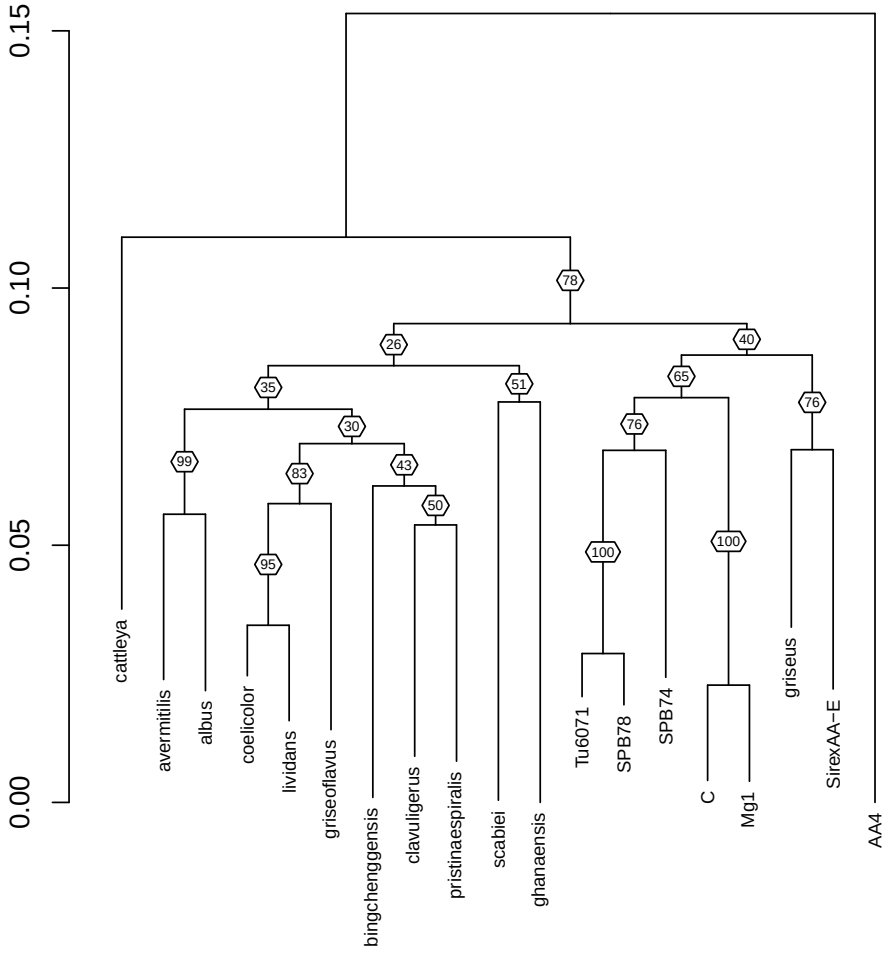


Figure 6: Tree with bootstrap support probabilities at each internal node.

9 More Examples of Manipulating Dendrograms

It is sometimes useful to alter *dendrogram* objects output by `Treeline`. There are three main ways for working with *dendrogram*s: apply a function to each leaf with `rapply`, apply a function to every node with `dendrapply`, or apply your own function recursively. The next examples will illustrate each of these approaches with increasing complexity.

In the first example, we will use `rapply` to query and set attributes of each leaf.

```
> rapply(tree, attr, which="label") # label of each leaf (left to right)
 [1] "ghanaensis"        "scabiei"           "avermitilis"
 [4] "albus"             "coelicolor"        "lividans"
 [7] "griseoflavus"      "bingchenggensis"   "cattleya"
[10] "clavuligerus"      "pristinaespiralis" "SPB74"
[13] "Mg1"               "C"                 "Tu6071"
[16] "SPB78"             "griseus"           "SirexAA-E"
[19] "AA4"
> labels(tree) # alternative
 [1] "ghanaensis"        "scabiei"           "avermitilis"
 [4] "albus"             "coelicolor"        "lividans"
 [7] "griseoflavus"      "bingchenggensis"   "cattleya"
[10] "clavuligerus"      "pristinaespiralis" "SPB74"
[13] "Mg1"               "C"                 "Tu6071"
[16] "SPB78"             "griseus"           "SirexAA-E"
[19] "AA4"
> rapply(tree, attr, which="height") # height of each leaf (left to right)
 [1] 9.218325e-01 6.252940e-01 6.590604e-01 6.251439e-01 5.035518e-01
 [6] 4.622402e-01 5.085003e-01 3.851613e-01 2.275979e-02 4.196046e-01
[11] 4.045839e-01 4.011957e-01 5.799664e-03 0.000000e+00 3.687537e-01
[16] 3.688375e-01 4.486063e-01 3.659990e-01 4.440892e-16
> italicize <- function(x) {
         if(is.leaf(x))
                 attr(x, "label") <- as.expression(substitute(italic(leaf),
                         list(leaf=attr(x, "label"))))
         x
 }
> rapply(tree, italicize, how="replace") # italicize leaf labels
'dendrogram' with 2 branches and 19 members total, at height 2.335526
```

In the second example, we will use `dendrapply` to identify exclusive groups wherein the members of each group are more similar to each other than they are to those outside the group [18].

```
> d <- DistanceMatrix(seqs, correction="F81+F", verbose=FALSE, processors=1)
> exclusive <- function(x) {
          if (!is.leaf(x)) { # leaves are trivially exclusive
                  leaves <- unlist(x)
                  max_dist <- max(d[leaves, leaves]) # max within group
                  if (all(max_dist < d[-leaves, leaves]))
                          attr(x, "edgePar") <- list(col="orange")
          }
          x
  }
> plot(dendrapply(tree, exclusive))
```

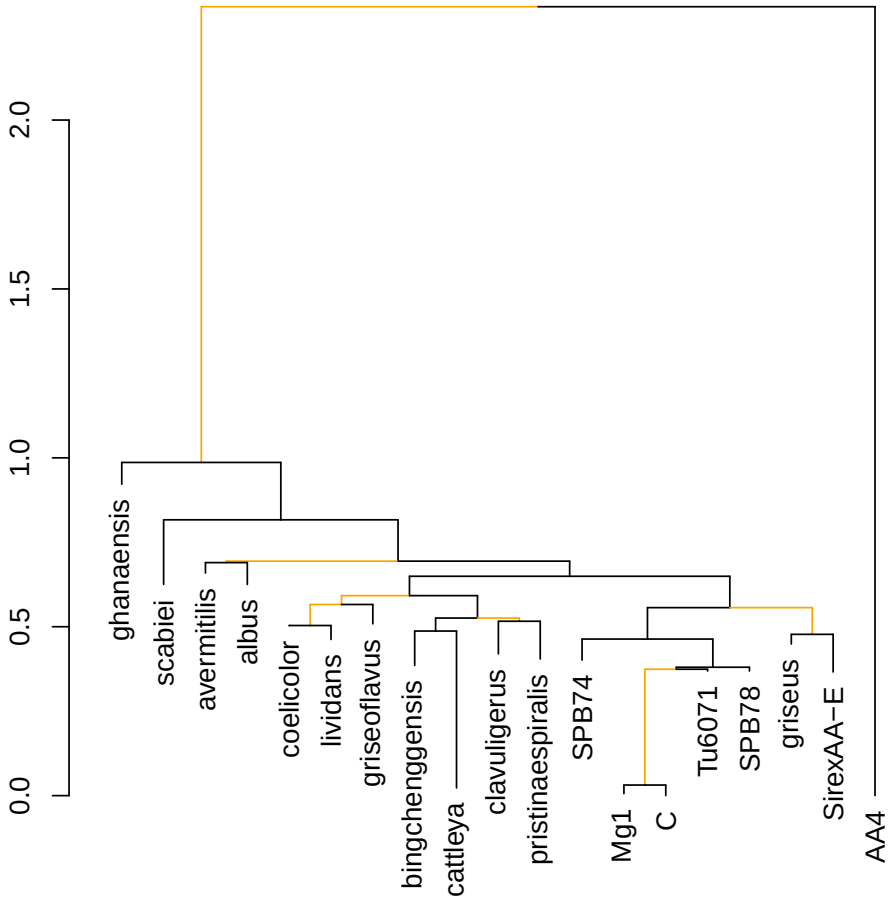


Figure 7: Tree with colored branches above exclusive groups.

In the third example, we will extract the branching order of five species of interest using a recursive function. This might be useful if we wanted to count how many times different topologies occurred among a set of trees. Recursion is the most flexible approach and can be applied with more sophisticated functions to accomplish goals beyond what is possible with `dendrapply`.

```
> Spp <- c("coelicolor", "lividans", "AA4", "Mg1", "scabiei") # species to retain
> extractClade <- function(x) {
         if (is.leaf(x)) {
                 if (sum(Spp %in% labels(x)) > 0L) {
                         labels(x)
                 } else {
                         NULL
                 }
         } else {
                 x <- lapply(x, extractClade)
                 x <- x[lengths(x) > 0]
                 if (length(x) == 1)
                         x <- x[[1]]
                 x
         }
 }
> extractClade(tree)
[[1]]
[[1]][[1]]
[1] "scabiei"

[[1]][[2]]
[[1]][[2]][[1]]
[[1]][[2]][[1]][[1]]
[1] "coelicolor"

[[1]][[2]][[1]][[2]]
[1] "lividans"


[[1]][[2]][[2]]
[1] "Mg1"



[[2]]
[1] "AA4"
```

10 Inspecting the Inputs and Outputs

If you are feeling down on your luck, you might want to double-check the inputs and outputs for any issues. First, we can check for any input sequences with unexpectedly few or many characters by comparing character frequencies across all input sequences. Next, we can look for input sequences that significantly deviate from the expected background frequencies using Pearson's chi-squared test. We can also check for sequences with extreme distances

that might be incorrectly aligned. Outliers in any of these checks may point to spurious sequences that should be double-checked for correctness or completion.

```
> freqs <- alphabetFrequency(seqs, baseOnly=TRUE)
> head(freqs)
       A   C   G   T other
[1,] 110 139 206 136    36
[2,] 101 137 207 123    59
[3,] 107 166 222 108    24
[4,] 112 151 207 124    33
[5,] 116 138 196 114    63
[6,] 112 139 195 115    66
> # summarize the number of non-base characters (gaps/ambiguities)
> summary(freqs) # "other" represents non-base characters
       A               C               G               T
 Min.   :101.0   Min.   :132.0   Min.   :192.0   Min.   :108.0
 1st Qu.:110.0   1st Qu.:137.5   1st Qu.:197.0   1st Qu.:114.5
 Median :114.0   Median :139.0   Median :206.0   Median :123.0
 Mean   :113.0   Mean   :141.8   Mean   :204.6   Mean   :121.6
 3rd Qu.:116.5   3rd Qu.:144.0   3rd Qu.:210.5   3rd Qu.:126.5
 Max.   :120.0   Max.   :166.0   Max.   :222.0   Max.   :136.0
     other
 Min.   :24.00
 1st Qu.:32.00
 Median :45.00
 Mean   :45.95
 3rd Qu.:60.00
 Max.   :69.00
> # index of sequence with the most non-base characters
> which.max(freqs[, "other"])
[1] 15
> freqs <- freqs[, DNA_BASES]
> background <- colMeans(freqs)
> background
       A        C        G        T
113.0000 141.7895 204.6316 121.6316
> # look for sequences deviating from background frequencies
> chi2 <- colSums((t(freqs) - background)^2/background)
> pval <- pchisq(chi2, length(background) - 1, lower.tail=FALSE)
> w <- which(pval < 0.05)
> seqs[w] # outlier sequences
DNAStringSet object of length 0
> freqs[w,] # frequencies of outliers
     A C G T
> # get sequence index of any very distant outlier sequences
> D <- DistanceMatrix(seqs, verbose=FALSE, processors=1)
> t <- table(which(D > 0.9, arr.ind=TRUE)) # choose a cutoff
> head(sort(t, decreasing=TRUE)) # index of top outliers, if any
integer(0)
```

It is also possible to check whether the output tree reasonably represents the distances between sequences. For ME trees, the tree should explain greater than `0.9` of the variance in the distance matrix used to construct the tree. We can use Pearson's correlation for trees with branch lengths in different units than the distance matrix (i.e., ML or MP). Lower correlations may result from alignments with sites having different genealogies, such as concatenated alignments or non-orthologous sequences.

```
> P <- Cophenetic(treeME) # patristic distances
> D <- as.dist(D) # conver to 'dist' object
> plot(D, P, xlab="Pairwise distance", ylab="Patristic distance", log="xy")
> abline(a=0, b=1)
> # for ME trees we want explained variance > 0.9
> V <- 1 - sum((P - D)^2)/sum((D - mean(D))^2)
> V # check the input data if V << 1
[1] 0.9441992
> cor(P, D) # Pearson's correlation coefficient should be >> 0
[1] 0.9725586
> cor(log(P), log(D)) # taking the logarithm lessens outliers
[1] 0.973351
```

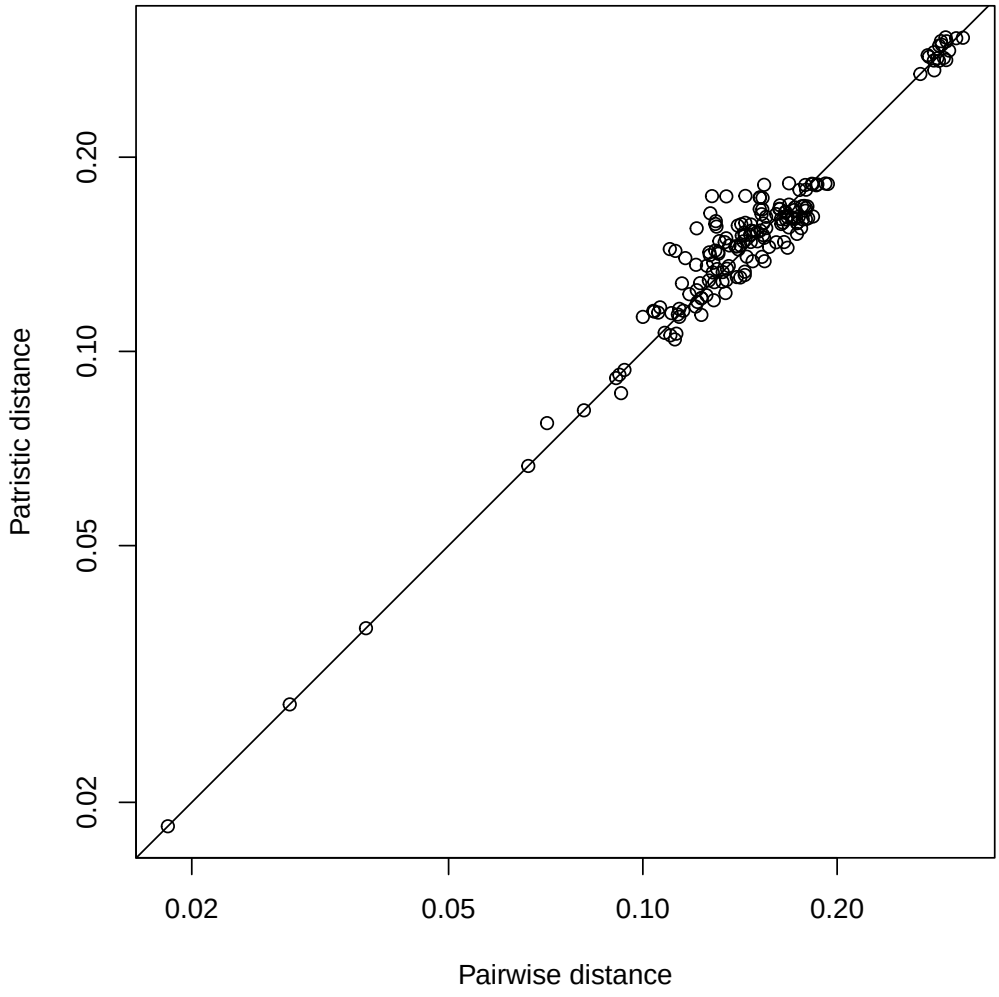


Figure 8: Confirming correlation between input distances and output patristic distances.

11 Exporting the Tree

We've had a run of good luck with this tree, so we'd better save it before our luck runs out! The functions `ReadDendrogram` and `WriteDendrogram` will import and export trees in Newick file format. If we leave the *file* argument blank then it will print the output to the console for our viewing:

```
> WriteDendrogram(tree, file="")
(('ghanaensis':0.06452862856,('scabiei':0.1912792961,(('avermitilis':0.03078935817,'albu
```

To keep up our lucky streak, we should probably include any model parameters in the output along with the tree. Luckily, Newick format supports square brackets (i.e., "[]") for comments, which we can append to the end of the file:

```
> params <- attr(tree, "parameters")
> cat("[", paste(names(params), params, sep="=", collapse=","), "]",
          sep="", append=TRUE, file="")
[FreqA=0.17656479102302,FreqC=0.243196872506926,FreqG=0.345666582799286,FreqT=NA,FreqI=NA
```

12 Generating a Summary Tree

So far, we've seen how variability can arise from different methodological decisions, but the choice of gene (or protein) also has a substantial influence on the tree. In particular, genes within a genome can have discordant genealogies due to biological processes such as incomplete lineage sorting, gene duplication/loss, and horizontal gene transfer [14]. Accounting for the inconsistencies among gene trees is important for inferring correct species phylogenies.

Lucky for us, DECIPHER has a function, named `Zipline`, for summarizing trees. `Zipline` takes a list of *dendrogram*s as input and returns a single summary tree as output. The `Zipline` algorithm leverages linear regression to generate a distance matrix representing relationships among the complete set of leaf labels. Similarly to regression, the `Zipline` function seamlessly handles missing data so that the input trees don't all need the same set of leaf labels.

Since we've lucked out so far, let's try building a species tree from a set of orthologous gene families previously collated from plant genomes [8]. These gene trees were built under the maximum likelihood "WAG+G4" model with 100 bootstrap replicates. To begin, we can compute some summary statistics about the plant dataset, which is provided along with the DECIPHER package.

```
> newick <- system.file("extdata", "Plant_gene_trees.newick.xz", package="DECIPHER")
> trees <- ReadDendrogram(newick)
> length(trees)
[1] 1103
> head(trees)
[[1]]
'dendrogram' with 3 branches and 52 members total, at height 0.3914406

[[2]]
'dendrogram' with 3 branches and 50 members total, at height 0.8568729

[[3]]
'dendrogram' with 3 branches and 68 members total, at height 0.5095524

[[4]]
'dendrogram' with 3 branches and 59 members total, at height 0.4560235

[[5]]
```

```
'dendrogram' with 3 branches and 50 members total, at height 0.1568177

[[6]]
'dendrogram' with 3 branches and 56 members total, at height 0.3387597
> tabulate(sapply(trees, attr, which="members"))
 [1]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
[26]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 95
[51] 87 74 75 88 72 55 81 66 58 57 63 56 37 32 23 21 22 15 10  9  1  5  0  0  1
> summary(sapply(trees, attr, which="height"))
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.1016  0.3971  0.4971  0.5173  0.6141  1.2471
> table(unlist(lapply(trees, labels)))
                Acacia koa   Acrocarpus fraxinifolius
                      1028                        728
             Afzelia bella        Albizia julibrissin
                       886                        969
          Alnus serrulata Amaranthus hypochondriacus
                       897                        637
       Anthonotha fragrans            Apios americana
                      1040                       1072
        Aquilegia coerulea       Arabidopsis thaliana
                       954                        733
          Arachis ipaensis    Astragalus membranaceus
                       907                        978
        Bauhinia tomentosa     Bituminaria bituminosa
                      1030                        911
             Cajanus cajan            Cannabis sativa
                      1013                       1028
             Carica papaya        Castanea mollissima
                       782                        929
         Cercis canadensis   Chamaecrista fasciculata
                       680                        337
           Cicer arietinum            Citrus sinensis
                      1003                        930
         Cladrastis lutea      Codariocalyx motorius
                       620                        923
     Copaifera officinalis            Cucumis sativus
                       956                        901
    Desmanthus illinoensis  Elaeocarpus photiniifolia
                       255                         98
         Entada abyssinica        Eucalyptus grandis
                       997                        972
           Fragaria vesca      Gleditsia triacanthos
                       806                        721
               Glycine max       Glycyrrhiza lepidota
                      1092                        957
       Gossypium raimondii        Gymnocladus dioicus
                      1069                        318
          Inga spectabilis              Juglans regia
                      1061                       1017
```

```
          Lactuca sativa           Lathyrus sativus
                     865                        904
          Lens culinaris        Linum usitatissimum
                     318                        783
         Lotus japonicus      Lupinus angustifolius
                     919                        721
      Lupinus polyphyllus          Manihot esculenta
                     354                       1056
      Medicago truncatula       Microlobius foetidus
                    1071                        999
         Mimulus guttatus            Morus notabilis
                     896                        912
         Nelumbo nucifera         Paeonia lactiflora
                    1039                        397
           Panax ginseng         Papaver somniferum
                     388                        864
       Phaseolus vulgaris              Pisum sativum
                    1080                        983
          Polygala lutea        Populus trichocarpa
                     678                       1081
           Primula veris        Prioria balsamifera
                     629                       1068
           Prosopis alba             Prunus persica
                     382                       1078
         Punica granatum         Quillaja saponaria
                     618                        802
          Salix purpurea            Senna hebecarpa
                    1071                        254
       Solanum tuberosum    Styphnolobium japonicum
                     820                        989
          Theobroma cacao         Trifolium pratense
                    1070                        672
   Tripterygium wilfordii                Vicia faba
                     746                        664
           Vigna radiata             Vitis vinifera
                     862                        949
  Xanthocercis zambesiaca             Zenia insignis
                    1049                        718
```

Note how `ReadDendrogram` imported 1,003 *dendrogram*s from the same Newick file as a *list* object. These trees have between 50 and 75 leaves and vary about an order of magnitude in their height. Most of the species are found in the majority of trees. The bootstrap values of the trees are imported as "edgetext" attributes and range between `0` and `100`. We can add a new "bootstrap" attribute to normalize the values between `0` and `1`.

```
> trees <- lapply(trees,
         dendrapply,
         function(x) {
                 b <- as.numeric(attr(x, "edgetext"))
                 if (length(b) == 0L) {
                         attr(x, "bootstrap") <- 0 # leaf
                 } else {
                         attr(x, "bootstrap") <- b/100
```

```
                }
                x
        })
```

Bootstrap values are positively correlated with branch length, and normalizing them allows us to easily construct a hybrid distance measure to use in summarizing the gene trees. `Zipline` supports *distance*s of `"length"` (path length between leaves), `"edges"` (also known as "internode" distance), and custom attributes or their combinations. Here we will try our chances at using the sum of edge length multiplied by our normalized bootstrap score as the measure of leaf-to-leaf distance.

`Zipline` constructs a distance matrix with linear regression and then calls `Treeline` to build a tree. Here, we asked `Zipline` to perform a very small number *bootstrap* replicates with the neighbor joining (`"NJ"`) method to speed up the example. Bootstrap values (fractions) are supplied as "edgetext" attributes on the output *dendrogram*. A few branches of the species tree have relatively low support (Fig. 9), suggesting more data are needed to resolve these splits on the species tree. Hopefully we'll have better luck in the future with more trees!

Different ways of parameterizing *distance* may return alternative trees, particularly in the absence of sufficient data. This raises the question: which measure of *distance* is best? Averaging output bootstrap values offers a means of comparing trees for their overall level of support. Luckily, `Zipline` is fast enough to compare different *distance* measures by generating alternative species trees and calculating their average support. Ideally all measures of *distance* would yield the same species tree, implying strong support.

```
> support <- 0
> count <- 0
> dendrapply(species_tree,
         function(x) {
                 s <- attr(x, "edgetext")
                 if (!is.null(s)) {
                         support <<- support + as.numeric(s)
                         count <<- count + 1
                 }
                 x
         })
'dendrogram' with 2 branches and 76 members total, at height 0.1512839
> support/count # average bootstrap support
[1] 0.9324324
```

```
> species_tree <- Zipline(trees,
          bootstraps=10, # typically 100 or more
          distance=c("length", "bootstrap"),
          concordanceVectors=TRUE)
Calculating bootstrap replicates:
================================================================================

Time difference of 99.26 secs

Computing concordance vectors:
================================================================================

Time difference of 14.99 secs
> plot(species_tree)
```

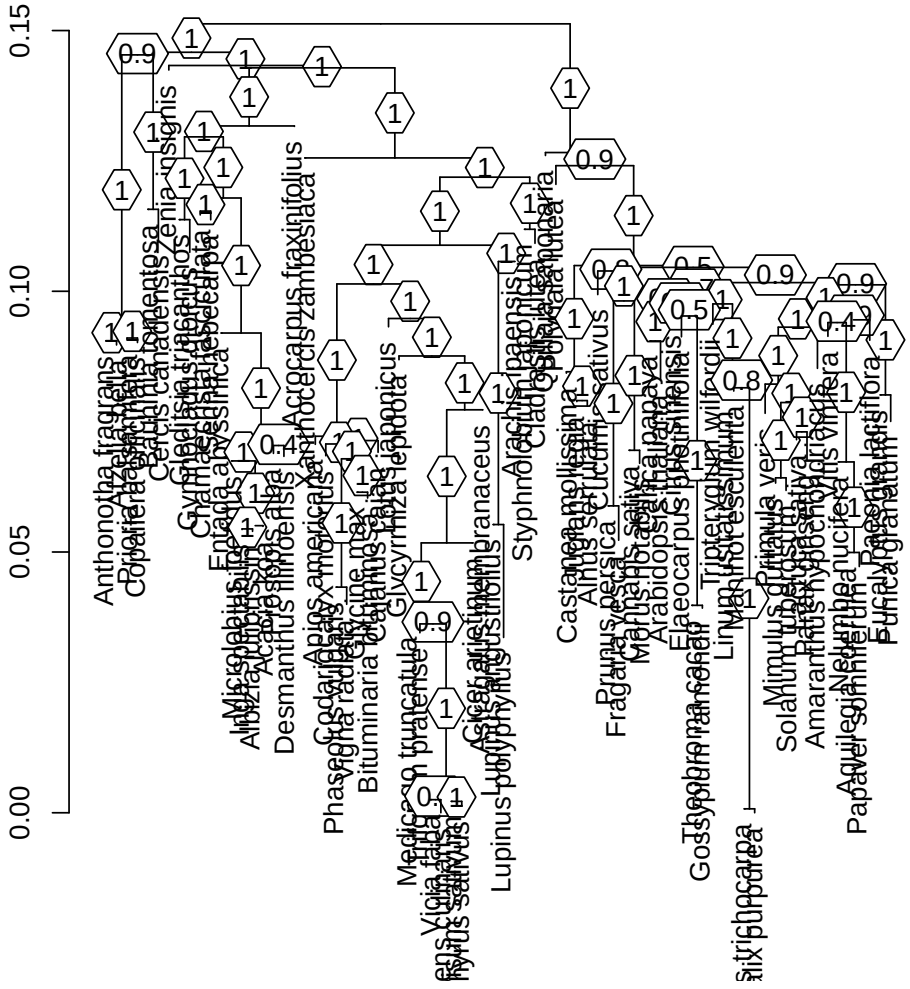


Figure 9: Summary tree of plant species with bootstrap confidences on each internal edge.

12.1 Concordance Vector Analysis

Multi-locus bootstrapping tells us about whether there is sufficient data for branches of the tree to be repeatable, but it doesn't give us insight into the signal strength underlying each branch. By a stroke of good luck, `Zipline` can calculate concordance vectors that show how many input trees supported each branch (Fig. 10). Concordance vectors have three named values provided with respect to both quartets and splits (bipartitions) [10] and another value for the total number of decisive trees. The first three values provide the number of trees supporting the observed split (`"CFsplit"`) versus two alternative splits (`"DF1split"` and `"DF2split"`). The second three values give analogous values for quartets, where the arrangement of each surrounding clade is considered. The final value provides the total number of decisive input trees, i.e., those containing at least one representative from each clade.

If we consider a multispecies coalescent process wherein the only source of gene tree incongruence is incomplete lineage sorting, we would expect the concordant topology to be at least as frequent as the two alternative discordant topologies that should be equally frequent (i.e., $CF \geq DF1 = DF2$). Violations of this 'minor symmetry' property imply that other evolutionary processes might be at work [11]. We would also expect longer branches to be less affected by incomplete lineage sorting, meaning the concordant topology (CF) should be more dominant. The number of paraphyletic (uninformative) trees is equal to the total number of decisive trees minus the counts of CF, $DF1$, and $DF2$.

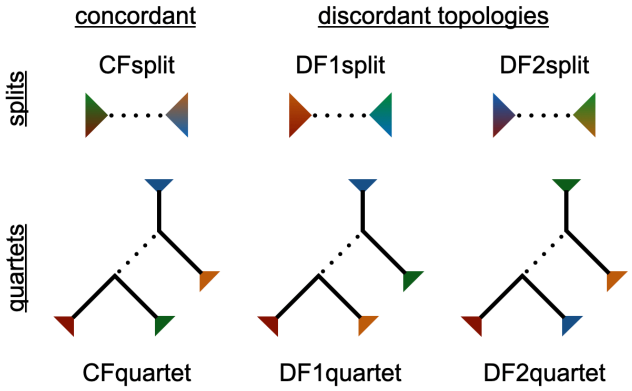
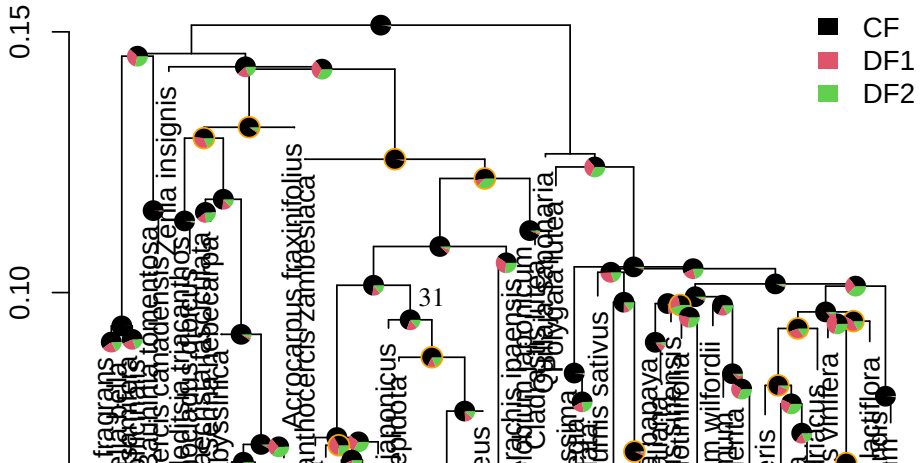


Figure 10: The values of concordance vectors are defined with respect to either the split or quartet induced by an internal branch (dotted edge). CF represents the summary tree (concordant) topology and is expected to be at least as abundant as the two discordant topologies ($DF1$ and $DF2$), which should have equal frequencies when incomplete lineage sorting is the only process acting on gene trees.

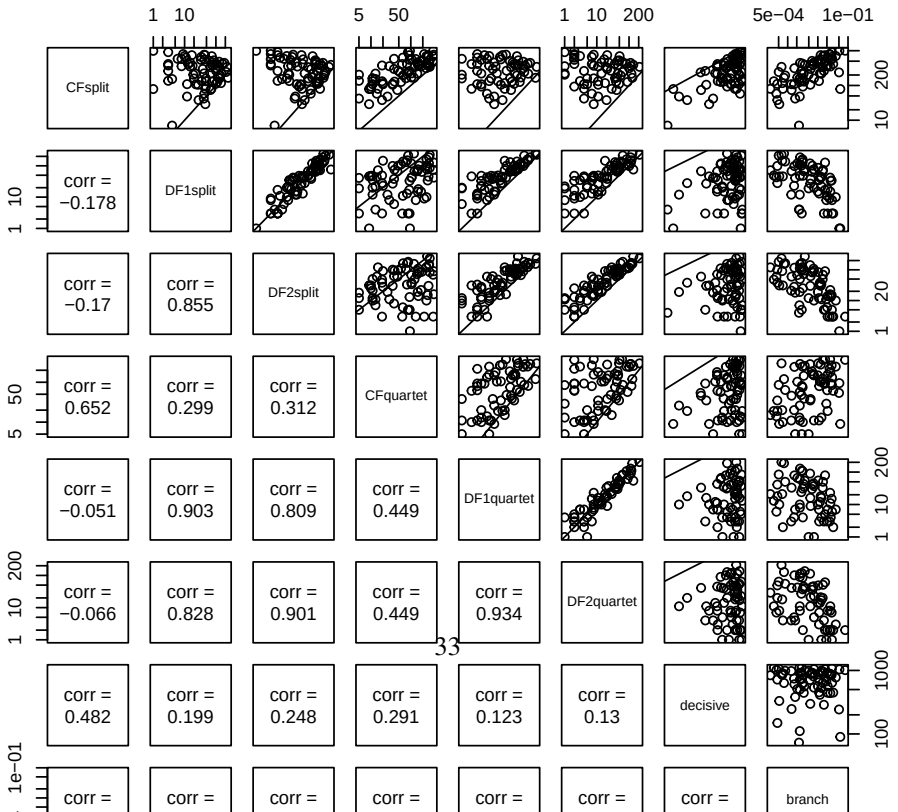
We can display the concordance vectors using pie charts at the base of each internal branch (Fig. 11). We will ignore the count of polyphyletic topologies, because these are comparatively uninformative. Plotting concordance vectors provides insights into the signal strength underlying each branch, and these insights are complementary to the statistical significance provided by multi-locus bootstrap support values. Also, it's straightforward to test for violations of minor symmetry using `pbinom`. We will draw an orange circle around pie charts with p-values less than $\alpha = 0.01$.

```
> drawPie <- function(x, y, p, r=1, alpha=0.01) {
        if (length(p) >= 3 && sum(p[1:3]) > 0) {
                dim <- par("usr")
                r_y <- r/((dim[2] - dim[1])/(dim[4] - dim[3]))
                pval <- pbinom(min(p[2:3]),
                        as.integer(sum(p[2:3])),
                        prob=0.5,
                        lower=TRUE)
                p <- p[1:3] # exclude DFp
                p <- c(1, cumsum(1000*p/sum(p)) + 1)
                angles <- seq(0, 2*pi, length.out=1000)
                for (i in seq_len(length(p) - 1))
                        polygon(c(x, x + r*cos(angles[p[i]:p[i + 1]]), x),
                                c(y, y + r_y*sin(angles[p[i]:p[i + 1]]), y),
                                col=i, border=NA)
                if (pval < alpha)
                        polygon(x + r*cos(angles),
                                y + r_y*sin(angles),
                                col=NA, border="orange")
        }
 }
> plot(dendrapply(species_tree,
        function(x) {
                attr(x, "edgetext") <- NULL
                x
        }))
> labs <- labels(species_tree)
> labs <- setNames(seq_along(labs), labs)
> dendrapply(species_tree,
        function(x) {
                drawPie(min(labs[labels(x)]) + attr(x, "midpoint"),
                        attr(x, "height"),
                        attr(x, "vector"))
                x
        })
'dendrogram' with 2 branches and 76 members total, at height 0.1512839
> legend("topright",
        c("CF", "DF1", "DF2"),
        fill=1:3,
        border=NA,
        ncol=1,
        bty="n")
```



With any luck we can further investigate the concordance vectors by extracting them from each internal branch of the summary tree and plotting observed quantities against each other (Fig. 12). In doing so, it's apparent that the two minor frequencies ($DF1$ and $DF2$) are correlated. The concordant frequency (CF) is related to the branch length, although this trend is most apparent for splits because quartets have lower counts. The number of decisive trees is high for most internal branches.

```
> vectors <- matrix(NA_integer_,
        nrow=attr(species_tree, "members") - 3, # internal branches
        ncol=8)
> count <- 0
> recordValues <- function(x, h) {
                if (!is.null(attr(x, "vector"))) {
                        count <<- count + 1
                        vectors[count,] <<- c(attr(x, "vector"),
                                h - attr(x, "height"))
                }
                if (length(x) > 1)
                        lapply(x, recordValues, h=attr(x, "height"))
                NULL
        }
> recordValues(species_tree, h=attr(species_tree, "height"))
NULL
> colnames(vectors) <- c(names(attr(species_tree, "vector")), "branch\nlength")
> suppressWarnings(pairs(vectors,
        log="xy", # omits zeros
        upper.panel=function(x, y, ...) {
                points(x, y, ...)
                abline(a=0, b=1)
        },
        lower.panel=function(x, y, ...) {
                r <- cor(x, y, "pairwise.complete.obs")
                usr <- par("usr")
                text(10^mean(usr[1:2]),
                        10^mean(usr[3:4]),
                        paste("corr =", round(r, 3), sep="\n"))
        }))
```



13 Conclusion

Using `Treeline` in combination with `Zipline` facilitates the exploration of alternative methods, models, and distance measures. Generating trees with `Treeline` and `Zipline` can be a lot of fun, especially when they include familiar species. May you have the best of luck with growing your own trees!

14 Session Information

All of the output in this vignette was produced under the following conditions:

- R version 4.6.1 (2026-06-24), `x86_64-pc-linux-gnu`
- Running under: `Ubuntu 26.04 LTS`
- Matrix products: default
- BLAS: `/usr/lib/x86_64-linux-gnu/openblas-pthread/libblas.so.3`
- LAPACK: `/usr/lib/x86_64-linux-gnu/openblas-pthread/libopenblasp-r0.3.32.so` ; LAPACK version3.12.0
- Base packages: base, datasets, graphics, grDevices, methods, stats, stats4, utils
- Other packages: BiocGenerics 0.59.12, Biostrings 2.81.7, DECIPHER 3.9.4, generics 0.1.4, IRanges 2.47.5, S4Vectors 0.51.9, Seqinfo 1.3.2, XVector 0.53.0
- Loaded via a namespace (and not attached): buildtools 1.0.0, cli 3.6.6, compiler 4.6.1, crayon 1.5.3, DBI 1.3.0, evaluate 1.0.5, KernSmooth 2.23-26, knitr 1.51, maketools 1.3.2, rlang 1.3.0, sys 3.4.3, tools 4.6.1, xfun 0.60

References

[1] Abadi, S., Azouri, D., Pupko, T., & Mayrose, I. Model selection may not be a mandatory step for phylogeny reconstruction. *Nat. Comm.*, 10(1).

[2] Anisimova, M., Gil, M., Dufayard, J., Dessimoz, C., & Gascuel, O. Survey of branch support methods demonstrates accuracy, power, and robustness of fast likelihood-based approximation schemes. *Syst. Biol.*, 60(5), 685-699.

[3] Ecker, N., Huchon, D., Mansour, Y., Mayrose, I., & Pupko, T. A machine-learning-based alternative to phylogenetic bootstrap. *Bioinformatics*, 40, i208-i217.

[4] Gonnet, G. Surprising results on phylogenetic tree building methods based on molecular sequences. *BMC Bioinformatics*, 13(148).

[5] Hoff, M., Orf, S., Riehm, B., Darriba, D., & Stamatakis, A. Does the choice of nucleotide substitution models matter topologically? *BMC Bioinformatics*, 17(143).

[6] Joy, J., Liang, R., McCloskey, R., Nguyen, T., & Poon, A. Ancestral Reconstruction. *PLoS Comp. Biol.*, 12(7), e1004763.

[7] Kapli, P., Kotari, I., Telford, M., Goldman, N., & Yang, Z. DNA Sequences Are as Useful as Protein Sequences for Inferring Deep Phylogenies. *Syst. Biol.*, 72(5), 1119-1135.

[8] Koenen, E., Ojeda, D., Steeves, R., Migliore, J., Bakker, F., Wieringa, J., Kinder, C., Hardy, O., Pennington, R., Bruneau, A., & Hughes, C. Large-scale genomic sequence data resolve the deepest divergences in the legume phylogeny and support a near-simultaneous evolutionary origin of all six subfamilies. *New Phytol.*, 225(3), 1355-1369.

[9] Kumar, S., Tao, Q., Lamarca, A., & Tamura, K. Computational Reproducibility of Molecular Phylogenies. *Mol. Biol. Evol.*, 40(7).

[10] Lanfear, R., Hahn, M. & Tamura, K. The Meaning and Measure of Concordance Factors in Phylogenomics. *Mol. Biol. Evol.*, 41(11), msae214.

[11] Pollard, D., Iyer, V., Moses, A., & Eisen, M. Widespread discordance of gene trees with species tree in Drosophila: evidence for incomplete lineage sorting. *PLoS Genet*, 2(10), e173.

[12] Ripplinger, J. & Sullivan, J. Does Choice in Model Selection Affect Maximum Likelihood Analysis? *Syst. Biol.*, 57(1), 76-85.

[13] Ripplinger, S., Sigorskikh, A., Efremov, A., Penzar, D., & Karyagina, A. PhyloBench: A Benchmark for Evaluating Phylogenetic Programs. *Mol. Biol. Evol.*, 41(6).

[14] Steenwyk, J., Li, Y., Zhou, X., Shen, X., & Rokas, A. Incongruence in the phylogenomics era. *Nat. Rev. Gen.*, 24(12), 834-850.

[15] Tan, G., Muffato, M., Ledergerber, C., Herrero, J., Goldman, N., Gil, M., & Dessimoz, C. Current methods for automated filtering of multiple sequence alignments frequently worsen single-gene phylogenetic inference. *Syst. Biol.*, 64(5), 778-791.

[16] Togkousidis, A., Stamatakis, A., & Gascuel, O. Accelerating Maximum Likelihood Phylogenetic Inference via Early Stopping to Evade (Over-)optimization. *Syst. Biol.*, 74(6), 1020-1028.

[17] Wheeler, A., Chatur, C., Goodman, P., Edgar, R., Huttley, G., & Masel, J. Improved gene tree inference from removing alignment errors both from focal genes and when training substitution models. *Mol. Biol. Evol.*, 43(8), 1-13.

[18] Wright, E. & Baum, D. Exclusivity offers a sound yet practical species criterion for bacteria despite abundant gene flow. *BMC Genomics*, 19(724).

[19] Wright, E. Treeline Provides a Unified Strategy for Optimising Phylogenetic Trees Under Alternative Criteria *Molecular Ecology Resources*, 26(5), e70163, 2026.

[20] Zwickl, D. & Hillis, D. Increased taxon sampling greatly reduces phylogenetic error. *Syst. Biol.*, 51(4), 588-598.