Growing phylogenetic trees with Treeline

Erik S. Wright

November 8, 2024

Contents

1 Introduction

This document describes how to grow phylogenetic trees using the `Treeline` function 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. This vignette focuses on optimizing, balanced minimum evolution (ME), maximum likelihood (ML), and maximum parsimony (MP) phylogenetic trees starting from sequences.

Why is the function called `Treeline`? 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` seeks to find a tree as close as possible to the Treeline, analogous to how trees cannot grow above the treeline on a mountain.

Why use `Treeline` versus other programs? The `Treeline` function is designed to return an excellent phylogenetic tree with minimal user intervention. Many tree building programs have a large set of complex options for niche applications. In contrast, `Treeline` simply builds a great tree by default. This vignette is intended to get you started and introduce additional options/functions that might be useful.

Treeline uses multi-start optimization followed by hill-climbing to find the highest trees on the optimality landscape. Since Treeline is a stochastic optimizer, it optimizes many trees to prevent chance from influencing the final result. With any luck it'll find the Treeline!

2 Performance Considerations

Finding an optimal tree is no easy feat. `Treeline` systematically optimizes hundreds of 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 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, and remove any sequences that are not necessary.
- 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 specify this argument.
- 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 only need to 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 some 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 may be an automatic 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 on the hardware being used.
- For ML, choose a model: Automatic model selection is a useful feature, but frequently this time-consuming step can be skipped. For most modestly large sets of nucleotide sequences, the `"GTR+G4"` model will be automatically selected. Typical amino acid sequences will tend to 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 considerable time.

3 Growing a Phylogenetic Tree

`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.

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. 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.

```
> 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 seq                                               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:

```
> 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 seq                                               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
```

Now, it's time to try our luck at finding the most likely tree. Here, 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.

Here, 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. For maximum parsimony, set *method* to `"MP"` and (optionally) specify a *costMatrix*. For maximum likelihood, set *method* to `"ML"`, which requires a model of sequence evolution. Note that `Treeline` automatically selects the best *model* according to Akaike information criterion (by default). It is possible to choose specific model(s) (e.g., `model="GTR+G4"`) to limit the possible selections and test your luck with fewer models.

Also, since `Treeline` is a stochastic optimizer, it is critical to always set the random number seed for 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 desired or you'll have no such luck.

```
> set.seed(123) # set the random number seed
> tree <- Treeline(seqs,
         method="ML",
         model="GTR+G4",
         reconstruct=TRUE,
         maxTime=0.01)
Fitting initial tree to model:
GTR+G4 -ln(L) = 4368, AICc = 8832, BIC = 9017

Optimizing up to 1000 candidate trees:

Tree #1. -ln(L) = 4366.470 (-0.043%), 1 Climb
Tree #2. -ln(L) = 4366.470 (0.000%), 8 Climbs, 0 Grafts of 1
Tree #3. -ln(L) = 4366.470 (0.000%), 4 Climbs, 0 Grafts of 3
Tree #4. -ln(L) = 4365.567 (-0.021%), 5 Climbs
Tree #5. -ln(L) = 4365.567 (0.000%), 8 Climbs, 0 Grafts of 2
Tree #6. -ln(L) = 4363.676 (-0.043%), 3 Climbs
Tree #7. -ln(L) = 4363.676 (0.000%), 7 Climbs, 0 Grafts of 2
Tree #8. -ln(L) = 4363.676 (0.000%), 2 Climbs, 0 Grafts of 1
Tree #9. -ln(L) = 4363.676 (0.000%), 6 Climbs, 0 Grafts of 2
Tree #10. -ln(L) = 4362.286 (-0.032%), 9 Climbs
Tree #11. -ln(L) = 4362.286 (0.000%), 11 Climbs, 0 Grafts of 1
Tree #12. -ln(L) = 4362.251 (-0.001%), 4 Climbs
Tree #13. -ln(L) = 4362.251 (0.000%), 7 Climbs, 0 Grafts of 2
Tree #14. -ln(L) = 4362.251 (0.000%), 7 Climbs, 0 Grafts of 1
Tree #15. -ln(L) = 4362.251 (0.000%), 5 Climbs, 0 Grafts of 1
Tree #16. -ln(L) = 4362.251 (0.000%), 7 Climbs, 0 Grafts of 1
Tree #17. -ln(L) = 4362.251 (0.000%), 16 Climbs, 0 Grafts of 4
Tree #18. -ln(L) = 4362.251 (0.000%), 7 Climbs, 0 Grafts of 2
Tree #19. -ln(L) = 4362.251 (0.000%), 8 Climbs, 0 Grafts of 0
Tree #20. -ln(L) = 4362.251 (0.000%), 4 Climbs, 0 Grafts of 2
Tree #21. -ln(L) = 4362.251 (0.000%), 3 Climbs, 0 Grafts of 0
Tree #22. -ln(L) = 4362.251 (0.000%), 4 Climbs, 0 Grafts of 0
Tree #23. -ln(L) = 4362.251 (0.000%), 6 Climbs, 0 Grafts of 1
Tree #24. -ln(L) = 4362.251 (0.000%), 9 Climbs, 0 Grafts of 0
Tree #25. -ln(L) = 4362.251 (0.000%), 8 Climbs, 0 Grafts of 0
Tree #26. -ln(L) = 4362.251 (0.000%), 6 Climbs, 0 Grafts of 0
Tree #27. -ln(L) = 4362.251 (0.000%), 8 Climbs, 0 Grafts of 1
Tree #28. -ln(L) = 4362.251 (0.000%), 6 Climbs, 0 Grafts of 1
Tree #29. -ln(L) = 4362.251 (0.000%), 6 Climbs, 0 Grafts of 0
Tree #30. -ln(L) = 4362.251 (0.000%), 14 Climbs, 0 Grafts of 0
Tree #31. -ln(L) = 4362.251 (0.000%), 5 Climbs, 0 Grafts of 1
Tree #32. -ln(L) = 4362.251 (0.000%), 10 Climbs, 0 Grafts of 1
Tree #33. -ln(L) = 4362.251 (0.000%), 5 Climbs, 0 Grafts of 1
Tree #34. -ln(L) = 4362.251 (0.000%), 5 Climbs, 0 Grafts of 0
Tree #35. -ln(L) = 4362.251 (0.000%), 6 Climbs, 0 Grafts of 1
Tree #36. -ln(L) = 4362.251 (0.000%), 9 Climbs, 0 Grafts of 1
Tree #37. -ln(L) = 4362.251 (0.000%), 7 Climbs, 0 Grafts of 1
Tree #38. -ln(L) = 4362.251 (0.000%), 12 Climbs, 0 Grafts of 0
Tree #39. -ln(L) = 4362.251 (0.000%), 15 Climbs, 0 Grafts of 0
Tree #40. -ln(L) = 4362.251 (0.000%), 4 Climbs, 0 Grafts of 1
Tree #41. -ln(L) = 4362.251 (0.000%), 9 Climbs, 0 Grafts of 0
Tree #42. -ln(L) = 4362.251 (0.000%), 4 Climbs, 0 Grafts of 0
Tree #43. -ln(L) = 4362.251 (0.000%), 11 Climbs, 0 Grafts of 0
Tree #44. -ln(L) = 4362.251 (0.000%), 6 Climbs, 0 Grafts of 1
Tree #45. -ln(L) = 4362.251 (0.000%), 3 Climbs, 0 Grafts of 1
Tree #46. -ln(L) = 4362.251 (0.000%), 4 Climbs, 0 Grafts of 1
Tree #47. -ln(L) = 4362.251 (0.000%), 5 Climbs, 0 Grafts of 1
```

4 Ancestral State Reconstruction

We're in luck —since we set *reconstruct* to `TRUE` `Treeline` automatically predict states at each internal node on the tree [3]. 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, 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] "G65T"  "G168T" "G171T" "G172C" "G173A" "G180T" "G181C" "G182C" "G184T"
[10] "G185T" "G186A" "G199A" "G201A" "G208T" "G209C" "G211A" "G227T" "G229C"
[19] "G256C" "C264T" "G271T" "G272C" "G276T" "G277C" "G280A" "G282A" "G287C"
[28] "G288C" "G302A" "G303A" "G314C" "G321T" "G323A" "G324T" "G325C" "G326T"
[37] "G327T" "G328C" "G333C" "G337T" "G338T" "G339C" "G343C" "G371C" "G379C"
[46] "G385A" "G389C" "G393T" "G396T" "G397C" "G419C" "G435A" "G437C" "G440T"
[55] "G584C"
```

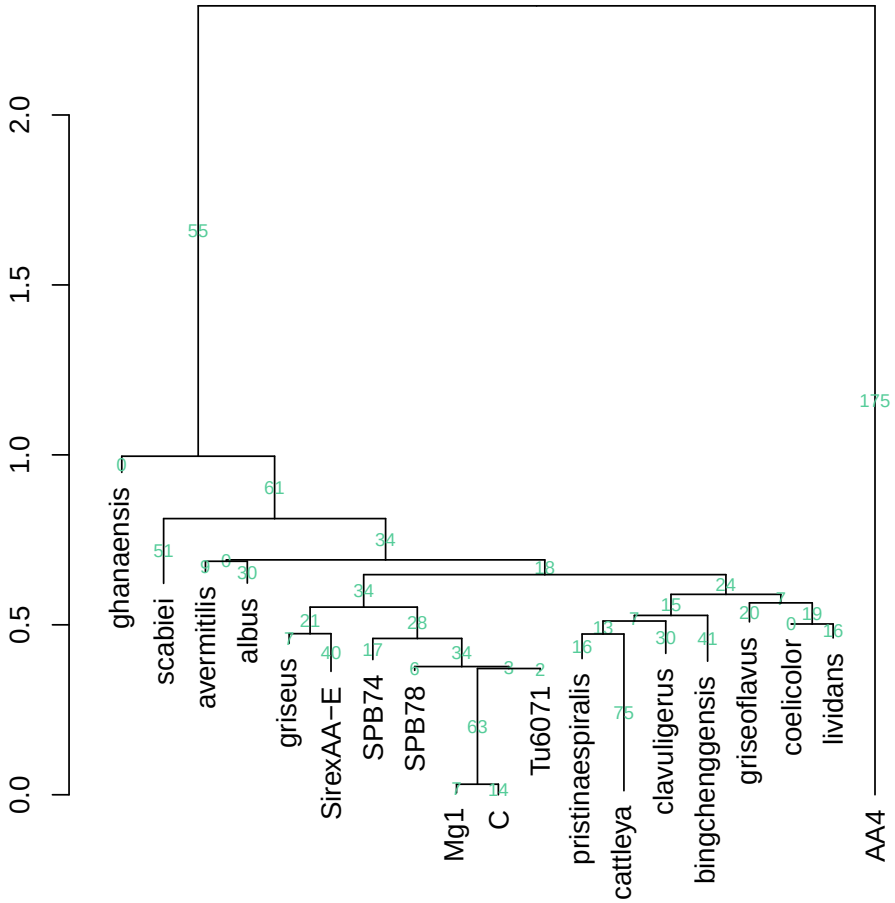


Figure 2: Edges labeled with the number of state transitions.

5 Plotting Branch Support Values

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

```
> #attributes(tree) # view all attributes
> 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] 2.321039
> attr(tree, "state") # ancestral state reconstruction (if reconstruct=TRUE)
[1] "......CACCGCCCGTCA.CGTCACGAAAGTCGGTAACACCCGAAGCCGGTGGCCCAACCCCCCG.GGGAGGGAGCCGTCGAA
> head(attr(tree, "siteLnLs")) # LnL for every alignment column (site)
[1] -2.023843 -1.579655 -2.023843 -2.364456 -1.942333 -2.364456
> attr(tree, "score") # best score (in this case, the -LnL)
[1] 4362.248
> 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
0.1746414 0.2444480 0.3461240        NA        NA 3.2612939 2.9442009 0.7171464
      A/T       C/G    Indels     alpha
1.1130626 0.5856481        NA 0.1916634
> attr(tree, "midpoint") # center of the edge (for plotting)
[1] 9.913086
```

The tree is (virtually) rooted at its midpoint by default. For maximum likelihood trees, all internal nodes include aBayes branch support values [1]. 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).

```
> plot(dendrapply(tree,
        function(x) {
                s <- attr(x, "probability") # choose "probability" (aBayes) or "support
                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(0, 0, 0.4, 0, code=3, angle=90, len=0.05, xpd=TRUE)
> text(0.2, 0, "0.4 subs./site", pos=3, xpd=TRUE)
```

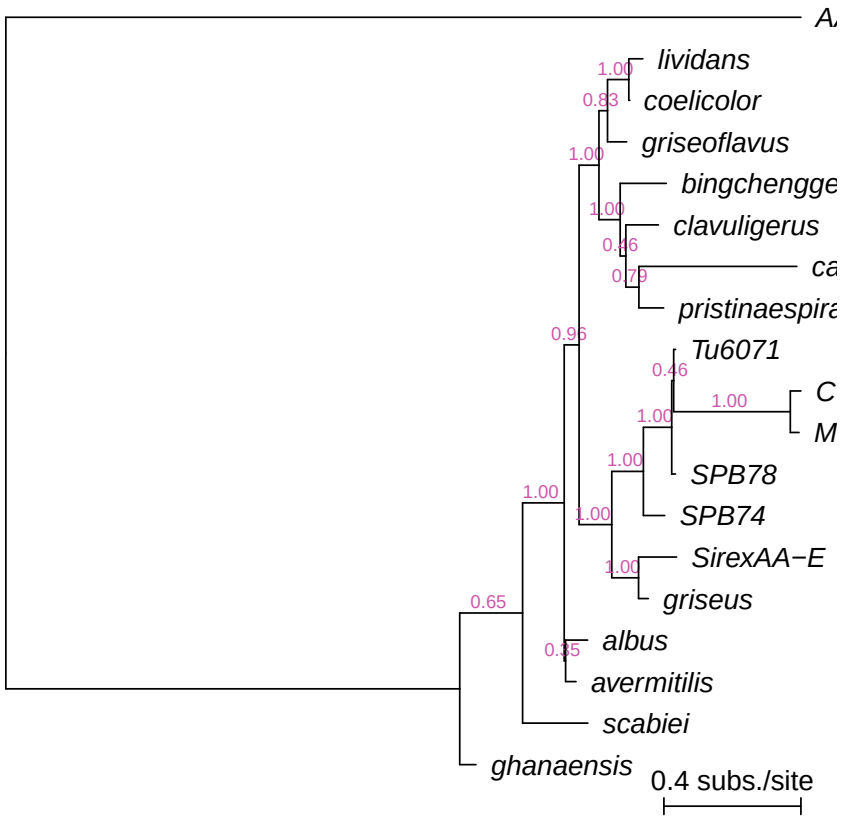


Figure 3: Tree with aBayes probabilities at each internal node.

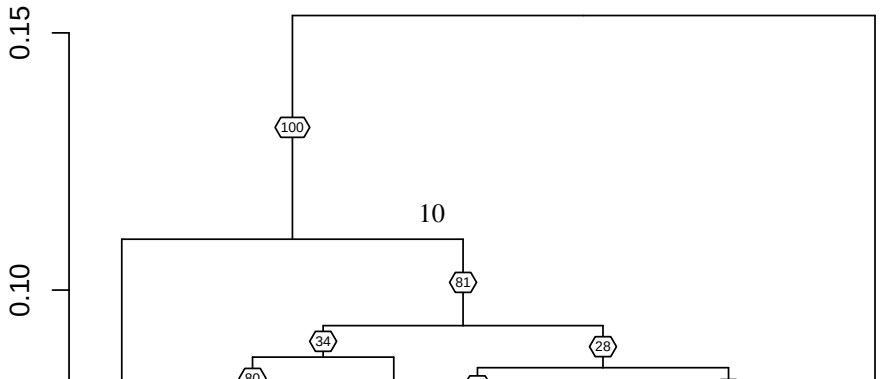
6 Calculating bootstrap support values

The aBayes probabilities are a good proxy for whether a partition in the tree is correct [2], but they are only available for maximum likelihood trees. For the other trees we need to make our own luck by bootstrapping the alignment. The idea behind bootstrapping is to resample columns (sites) of the alignment with replacement and determine whether each partition was found in the original tree. Repeating this process allows us to measure the level of support for each branch.

```
> reps <- 100 # number of bootstrap replicates
> tree1 <- Treeline(seqs, verbose=FALSE, processors=1L)
> partitions <- function(x) {
        if (is.leaf(x))
                return(NULL)
        x0 <- paste(sort(unlist(x)), collapse=" ")
        x1 <- partitions(x[[1]])
        x2 <- partitions(x[[2]])
        return(list(x0, x1, x2))
 }
> pBar <- txtProgressBar()
> bootstraps <- vector("list", reps)
> for (i in seq_len(reps)) {
        r <- sample(width(seqs)[1], replace=TRUE)
        at <- IRanges(r, width=1)
        seqs2 <- extractAt(seqs, at)
        seqs2 <- lapply(seqs2, unlist)
        seqs2 <- DNAStringSet(seqs2)

        temp <- Treeline(seqs2, verbose=FALSE)
        bootstraps[[i]] <- unlist(partitions(temp))
        setTxtProgressBar(pBar, i/reps)
 }
================================================================================
> close(pBar)

> bootstraps <- table(unlist(bootstraps))
> original <- unlist(partitions(tree1))
> hits <- bootstraps[original]
> names(hits) <- original
> w <- which(is.na(hits))
> if (length(w) > 0)
        hits[w] <- 0
> hits <- round(hits/reps*100)
> labelEdges <- function(x) {
        if (is.null(attributes(x)$leaf)) {
                part <- paste(sort(unlist(x)), collapse=" ")
                attr(x, "edgetext") <- as.character(hits[part])
        }
        return(x)
 }
> tree2 <- dendrapply(tree1, labelEdges)
> attr(tree2, "edgetext") <- NULL
> plot(tree2, edgePar=list(t.cex=0.5), nodePar=list(lab.cex=0.7, pch=NA))
```



7 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':0.04757782,('scabiei':0.1906316,(('avermitilis':0.03044386,'albus':0.0642
```

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 for good luck:

```
> params <- attr(tree, "parameters")
> cat("[", paste(names(params), params, sep="=", collapse=","), "]", sep="", append=TRUE
[FreqA=0.174641419960624,FreqC=0.244448043154763,FreqG=0.346123990416285,FreqT=NA,FreqI=
```

8 Session Information

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

- R version 4.4.1 (2024-06-14), `aarch64-apple-darwin20`
- Running under: `macOS Ventura 13.6.7`
- Matrix products: default
- BLAS: `/Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRblas.0.dylib`
- LAPACK: `/Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRlapack.dylib` ; LAPACK version3.12.0
- Base packages: base, datasets, grDevices, graphics, methods, stats, stats4, utils
- Other packages: BiocGenerics 0.52.0, Biostrings 2.74.0, DECIPHER 3.2.0, GenomeInfoDb 1.42.0, IRanges 2.40.0, S4Vectors 0.44.0, XVector 0.46.0
- Loaded via a namespace (and not attached): DBI 1.2.3, GenomeInfoDbData 1.2.13, KernSmooth 2.23-24, R6 2.5.1, UCSC.utils 1.2.0, compiler 4.4.1, crayon 1.5.3, httr 1.4.7, jsonlite 1.8.9, tools 4.4.1, zlibbioc 1.52.0

References

[1] 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.

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

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