Growing phylogenetic trees with TreeLine

Erik S. Wright
October 24, 2023

Contents

1 Introduction 1
2 Performance Considerations 1
3 Growing a Phylogenetic Tree 2
4 Plotting Branch Support Values 5
5 Ancestral State Reconstruction 11
6 Exporting the Tree 13
7 Session Information 13

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 building maximum likelihood (ML) and maximum parsimony (MP) phylogenetic trees starting from sequences, but TreeLine can also be used to build additive trees from a distance matrix.

Why is the function called TreeLine? The goal of TreeLine is to find the most likely/parsimonious tree for a given sequence alignment. There are often many trees with nearly maximal likelihood/parsimony. Therefore, TreeLine seeks to find a tree as close as possible to the treeline, analogous to how no trees can 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 when relying on its defaults. 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 likelihood or parsimony landscapes. Since TreeLine is a stochastic optimizer, it optimizes many trees to prevent luck from influencing the final result. The main difference from most other approaches to tree optimization is that TreeLine heavily uses past trees to generate new trees as it optimizes. With any luck it’ll find the treeline!

2 Performance Considerations

Finding a tree with very high likelihood/parsimony is no easy feat. TreeLine systematically optimizes hundreds to thousands 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 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 below, and remove any sequences that are not necessary. This concern is shared for all tree building programs, and TreeLine is no exception.

• 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.

• 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 too long then simply specify this argument.

• Compile with OpenMP support: Significant speed-ups can be achieved with multi-threading using OpenMP. See the “Getting Started DECIPHERing” vignette for how to do this on your computer platform. 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 “-O3 -march=native” to the end of PKG_CFLAGS in the “DECIPHER/src/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.

3 Growing a Phylogenetic Tree

TreeLine takes as input a multiple sequence alignment when constructing a maximum likelihood or maximum parsimony phylogenetic tree. 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.

```r
> 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 TGTACACCGCCCCTACA-CGTC...GGGTTTCCGAATGGGAAACC supercont3.1 of S...
[2]  627 NNNNACACCGCCCCTACA-CGTC...GGGTTTCCGAATGGGAAACC supercont3.1 of S...
[3]  627 TGTACACCGCCCCTACA-CGTC...GGGTTTCCGAATGGGAAACC supercont1.1 of S...
[4]  627 CGTACACCGCCCCTACA-CGTC...GGGTTTCCGAATGGGAAACC supercont1.1 of S...
[5]  627 TGTACACCGCCCCTACA-CGTC...GGGTTTCCGAATGGGAAACC supercont1.1 of S...
[6]  627 TGTACACCGCCCCTACA-CGTC...GGGTTTCCGAATGGGAAACC supercont1.1 of S...
[7]  627 TGTACACCGCCCCTACA-CGTC...GGGTTTCCGAATGGGAAACC supercont1.1 of S...
[8]  627 TGTACACCGCCCCTACA-CGTC...GGGTTTCCGAATGGGAAACC supercont1.1 of S...
[9]  627 TGTACACCGCCCCTACA-CGTC...GGGTTTCCGAATGGGAAACC supercont1.1 of S...
[10] 627 TGTACACCGCCCCTACA-CGTC...GGGTTTCCGAATGGGAAACC gi|297189896|ref|...
[11] 627 TGTACACCGCCCCTACA-CGTC...GGGTTTCCGAATGGGAAACC gi|224581106|ref|...
```
Many of these sequences are redundant or from the same genome. We can de-replicate the sequences to accelerate tree building:

```r
> seqs <- unique(seqs) # remove duplicated sequences
> ns <- gsub("^.*Streptomyces\s*\( subsp\. | sp\. | | sp_\)\([^ \]+\).*", \"\2\", names(seqs))
> names(seqs) <- ns # name by species
> seqs <- seqs[!duplicated(ns)] # remove redundant sequences from the same species
> seqs
```

DNAStringSet object of length 19:

<table>
<thead>
<tr>
<th>width</th>
<th>seq</th>
<th>names</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>627</td>
<td>TGTACACACCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC albus</td>
</tr>
<tr>
<td>[2]</td>
<td>627</td>
<td>TGTACACACCGCCTGCA-CGTC...GGGGTTTCCGAATGGGGAAACC clavuligerus</td>
</tr>
<tr>
<td>[3]</td>
<td>627</td>
<td>TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC ghanaensis</td>
</tr>
<tr>
<td>[4]</td>
<td>627</td>
<td>TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC griseoflavus</td>
</tr>
<tr>
<td>[5]</td>
<td>627</td>
<td>TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC lividans</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>[15]</td>
<td>627</td>
<td>TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC cattleya</td>
</tr>
<tr>
<td>[16]</td>
<td>627</td>
<td>TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC bingchenggensis</td>
</tr>
<tr>
<td>[17]</td>
<td>627</td>
<td>TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC avermitilis</td>
</tr>
<tr>
<td>[18]</td>
<td>627</td>
<td>TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC C</td>
</tr>
<tr>
<td>[19]</td>
<td>627</td>
<td>TGTACACACCGCCCGTCA-CGTC...GGGGTTTCCGAATGGGGAAACC Tu6071</td>
</tr>
</tbody>
</table>

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.

Note that `TreeLine` automatically selects a substitution model based on Akaike information criterion (by default). It is possible to specify 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)

Optimizing model parameters:
GTR+G4 \(-\ln(L) = 4368\), AICc = 8832, BIC = 9017

PHASE 1 OF 3: INITIAL TREES

1/3. Optimizing initial tree #1 of 10 to 100:
\(-\ln(L) = 4366.5\) (-0.044%), 1 Climb
1/3. Optimizing initial tree #2 of 10 to 100:
\(-\ln(L) = 4365.2\) (-0.029%), 7 Climbs
1/3. Optimizing initial tree #3 of 11 to 100:
\(-\ln(L) = 4364.1\) (-0.026%), 5 Climbs

PHASE 2 OF 3: REGROW GENERATION 1 OF 10 TO 20

2/3. Optimizing regrown tree #1 of 10 to 100:
\(-\ln(L) = 4366.5\) (+0.054%), 2 Climbs
2/3. Optimizing regrown tree #2 of 10 to 100:
\(-\ln(L) = 4366.5\) (+0.054%), 3 Climbs
2/3. Optimizing regrown tree #3 of 10 to 100:
\(-\ln(L) = 4364.8\) (+0.016%), 3 Climbs
2/3. Optimizing regrown tree #4 of 10 to 100:
\(-\ln(L) = 4365.5\) (+0.054%), 1 Climb
2/3. Optimizing regrown tree #5 of 10 to 100:
\(-\ln(L) = 4363.7\) (-0.010%), 2 Climbs
2/3. Optimizing regrown tree #6 of 15 to 100:
\(-\ln(L) = 4363.7\) (-0.000%), 3 Climbs
2/3. Optimizing regrown tree #7 of 15 to 100:
\(-\ln(L) = 4363.1\) (-0.014%), 5 Climbs
2/3. Optimizing regrown tree #8 of 17 to 100:
\(-\ln(L) = 4366.8\) (+0.085%), 3 Climbs

PHASE 3 OF 3: SHAKEN TREES

Grafting 0 tree to the best tree:
\(-\ln(L) = 4363.1\) (0.000%), 0 Grafts of 0

3/3. Optimizing shaken tree #1 of 3 to 1000:
\(-\ln(L) = 4362.4\) (-0.016%), 6 Climbs
3/3. Optimizing shaken tree #2 of 3 to 1000:
\(-\ln(L) = 4362.2\) (-0.003%), 5 Climbs
3/3. Optimizing shaken tree #3 of 3 to 1000:
\(-\ln(L) = 4364.9\) (+0.061%), 16 Climbs

Grafting 2 trees to the best tree:
\(-\ln(L) = 4362.2\) (0.000%), 0 Grafts of 2

3/3. Optimizing shaken tree #4 of 6 4 to 1000:
\(-\ln(L) = 4378.9\) (+0.381%), 5 Climbs
3/3. Optimizing shaken tree #5 of 6 to 1000:
\(-\ln(L) = 4364.3\) (+0.047%), 8 Climbs
3/3. Optimizing shaken tree #6 of 6 to 1000:
\(-\ln(L) = 4374.5\) (+0.280%), 4 Climbs

Model parameters:
Frequency(A) = 0.175
Frequency(C) = 0.245
Frequency(G) = 0.345
Frequency(T) = 0.235
Rate A <-> C = 0.719
Rate A <-> G = 3.301
Rate A <-> T = 1.089
Rate C <-> G = 0.585
Rate C <-> T = 2.875
Rate G <-> T = 1.000
Alpha = 0.191
Time difference of 54.47 secs

> set.seed(NULL) # reset seed
> plot(tree)

Figure 1: Maximum likelihood tree showing the relationships between *Streptomyces* species.
4 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:

```r
> attributes(tree) # view all attributes
$members
[1] 19

$height
[1] 2.350704

$state
[1] "--------CACCGCCGCACA-CGTCACGAAAGTCGTAACACCGAAGCCGCTGGGCCCCACCCCG-GGGAGGGACCCGCTGAGG/

$class
[1] "dendrogram"

$siteLnLs
[19] 0.000000 -2.086736 -1.722276 -2.169651 -2.086736 -2.169651
[37] -5.920308 -4.579034 -5.920308 -2.086736 -2.086736 -2.086736
[67] -1.722276 -1.722276 -1.722276 -2.514896 -1.722276 -1.722276
[73] -1.722276 -2.514896 -1.722276 -8.707183 -12.085434 -1.722276
[85] -1.722276 -2.169651 -1.722276 -1.722276 -1.722276 -2.514896
[97] -2.514896 -2.169651 -2.169651 -1.722276 -1.722276 -1.722276
[121] -2.514896 -1.722276 -2.086736 -2.086736 -1.722276 -2.169651
[133] -2.514896 -1.722276 -1.722276 -2.169651 -1.722276 -2.086736
```
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 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.lwd=1e-5, t.col="#CC55AA", t.cex=0.7) 
  if (is.leaf(x)) 
    attr(x, "nodePar") <- list(lab.font=3, pch=NA) 
  x 
}), 
  horiz=TRUE, 
yaxt='n')
> # add a scale bar
> 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 2: Tree with (aBayes) support probabilities at each internal node.
We lucked out because maximum likelihood and maximum parsimony trees both provide branch supports in the form of the fraction of optimized trees that contained a given partition (branch). These are accessible from the “support” attribute. As luck would have it, support values and (aBayes) probabilities are correlated, but support tends to be more conservative.
> getSupports <- function(x) {
  if (is.leaf(x)) {
    NULL
  } else if (is.null(attr(x, "support"))) {
    rbind(getSupports(x[[1]]), getSupports(x[[2]]))
  } else {
    rbind(cbind(attr(x, "support"),
               attr(x, "probability"),
               attr(x, "members")),
           getSupports(x[[1]]), getSupports(x[[2]]))
  }
}

> support <- getSupports(tree)
> plot(support[, 1],
      support[, 2],
      cex=log10(support[, 3]),
      xlab="Support", ylab="aBayes probability", asp=1)
> abline(a=0, b=1, lty=2) # line of identity (y=x)

Figure 3: Comparison of aBayes probabilities and branch support values.
5 Ancestral State Reconstruction

We’re in luck — one of the advantages of maximum likelihood and maximum parsimony tree building methods is that they automatically predict states at each internal node on the tree. This feature is enabled when reconstruct is set to TRUE. These character states can be used by the function MapCharacters to determine state transitions along each edge of the tree.
> new_tree <- MapCharacters(tree, labelEdges=TRUE)
> plot(new_tree, edgePar=list(p.col=NA, p.lwd=1e-5, 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" "A209C" "G211A" "G220C" "G224C"
[19] "G227T" "G229C" "T234G" "G259C" "C264T" "G271T" "G272C" "G276T" "G277C"
[28] "G280A" "G282A" "G287C" "G288C" "G302A" "G303A" "G314C" "G321T" "G323A"
[37] "G324T" "G325C" "G326T" "G327T" "G328C" "G331A" "G333C" "G337T" "G338T"
[46] "G339C" "T343C" "G371C" "G379C" "G385A" "G389C" "G393T" "G396T" "G397C"
[55] "G419C" "G435A" "G437C" "G440T" "G584C"

Figure 4: Edges labeled with the number of state transitions.
6 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:

```r
> WriteDendrogram(tree, file="")

("ghanaensis":0.07217672,("scabiei":0.1928848,((("coelicolor":1.533e-07,="lividans":0.06275875,"griseoflavus...4767):0.04461541,("avermitilis":0.03091427,"albus":0.06506649):0.004593969):0.1235258):0.1639449):1.36382,"AA4":2.350704);
```

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:

```r
> params <- attr(tree, "parameters")
> cat("[", paste(names(params), params, sep="=", collapse=","), "]", sep="", append=TRUE, file=

[FreqA=0.175345208555346,FreqC=0.24469103542414,FreqG=0.344501085928946,FreqT=NA,FreqI=NA,
```

7 Session Information

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

- **R version 4.3.1 (2023-06-16), x86_64-pc-linux-gnu**
- **Running under:** Ubuntu 22.04.3 LTS
- **Matrix products:** default
- **BLAS:** /home/biocbuild/bbs-3.18-bioc/R/lib/libRblas.so
- **LAPACK:** /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.10.0
- **Base packages:** base, datasets, grDevices, graphics, methods, parallel, stats, stats4, utils
- **Other packages:** BiocGenerics 0.48.0, Biostrings 2.70.0, DECIPHER 2.30.0, GenomeInfoDb 1.38.0, IRanges 2.36.0, RSQLite 2.3.1, S4Vectors 0.40.0, XVector 0.42.0
- **Loaded via a namespace (and not attached):** DBI 1.1.3, GenomeInfoDbData 1.2.11, KernSmooth 2.23-22, RCurl 1.98-1.12, bit 4.0.5, bit64 4.0.5, bitops 1.0-7, blob 1.2.4, cachem 1.0.8, cli 3.6.1, compiler 4.3.1, crayon 1.5.2, fastmap 1.1.1, memoise 2.0.1, pkgconfig 2.0.3, rlang 0.6.4, zlibbioc 1.48.0

References
