PhyloNetworks Documentation

Version 0.0.1

Claudia Solís-Lemus, Cécile Ané and John Spaw

October 28, 2015

1 Introduction

PhyloNetworks is a Julia package with several functions for phylogenetic networks, among which we can highlight reading from and writing in parenthetical format, re-rooting and plotting. However, the main function in PhyloNetworks is the implementation of the method SNaQ (Solís-Lemus and Ané, 2015): a statistical method to infer a phylogenetic network from input gene trees.

Assumption on the phylogenetic network. For now, we assume that the hybridization cycles do not intersect, and these networks are called *level-1 networks* (Huson et al., 2010). See the appendix for more details on the definition of phylogenetic networks used here. This restriction (and others mentioned in the appendix) is enforced in the optimization, but not in the read/write/plot/root functions. However, the only rule strictly enforced for all functions is that a hybrid node cannot have more than two hybrid edges pointing at it. This is a restriction that we hope to eliminate in the future.

Why Julia? Julia is a high-level and interactive programming language (like R or Matlab), but it is also high-performance (like C). So, it is fast. However, there is a minor drawback in Julia versions 0.3 (or lower): Julia code is just-in-time compiled which means that the first time you run a function, it will take a lot of time because it will compile it. It is important to keep this in mind as we use Julia, because the next calls for a function will be much much faster. So, please be patient! Trying out small toy examples for the first calls of functions is always a good idea.

2 Setup

2.1 Installation of Julia

To install Julia version 0.3.* (if Julia is already installed, skip this step): go to http://julialang.org/downloads/.
PhyloNetworks was developed under Julia version 0.3.5, and has been tested on different versions of 0.3.X. We have not tested its robustness on Julia version 0.4 or above, but we will soon. Version 0.4 allows for precompilation, so the drawback described in the introduction will no longer apply.

2.2 Installation of PhyloNetworks

To install the package, open Julia and type:

```
Pkg.clone("https://github.com/crsl4/PhyloNetworks.git")
Pkg.build("PhyloNetworks")
```

The PhyloNetworks package has the following dependencies, but everything is installed when PhyloNetworks is added.

- GraphViz (version 0.0.3)
- NLopt (version 0.2.0)

The version in parenthesis correspond to the ones used when implementing PhyloNetworks. To test that you installed correctly the PhyloNetworks package, try the following example:

2.2.1 Example to test correct PhyloNetworks installation

Open Julia and type

```
Pkg.test("PhyloNetworks")
```

This will take a couple of minutes as the package needs to compile all the functions. If the installation was successful, you will see a message at the end: **Tests passed**. Otherwise, an error will be thrown.

3 Usage of PhyloNetworks

Before each session, need to type in Julia:

```
using PhyloNetworks
```

This will take a couple of minutes as it needs to precompile the functions. The first run of a function in Julia will also compile, so it will be slower than any subsequent runs.

4 Update of PhyloNetworks

It is important to regularly update the version of the Julia packages:

```
Pkg.update()
```

This will take a couple of minutes as it needs to precompile the functions in every package. This is particulary important for the package `PhyloNetworks` since it is a new package under constant development.

WARNING: There is a known bug for Mac users where the `Pkg.update` function does not update to the latest version. We recommend Mac users to do the following through the terminal:
`cd HOME/.julia/v0.3/PhyloNetworks/`
`git pull`
where HOME is replaced by your home directory and v0.3 could be replaced by v0.4 if you have version 0.4 of Julia.

5 Description of main functions

Functions in `Julia` that end with exclamation sign `!` are modifying one (or all) of the arguments.

Important: Functions in Julia have required arguments and optional arguments. Required arguments **should not be named**. For example, you would call `readTopology("(A,(B,(C,D)));")` not `readTopology(net="(A,(B,(C,D)));")`. In the following descriptions, it is specified which arguments are optional and which arguments are required. If there is only one argument, it is required.

The description of the functions in `PhyloNetworks` is next:

- **readTopology:** Function to read a tree or network from parenthetical format. Input can be a string or a name of a text file (text file should contain only one line with the tree and end in ;, no quotes. To read more than one tree, see `readInputTrees` funcion). This function returns a `HybridNetwork` object. This function allows for many topologies that can be forbidden with the restriction described above. That is, this function allows for polytomies in tree nodes (tree node with more than two children) and internal nodes with only two edges.
  Usage:

  ```
  net=readTopology(filename);
  net=readTopology(string);
  ```

  WARNING: it is preferable to end the command with `;` to avoid printing to screen some inner details on the network object.

- **readTopologyLevel1:** Same as `readTopology`, but this function enforces the restrictions on the network. `readTopology` should be used when you want more variety of topologies, but these topologies cannot be used directly as starting point in the `SNaQ` optimization method (which is taken care automatically inside the optimization function). This function also returns a `HybridNetwork` object.
  Usage:

```
net=readTopologyLevel1(filename);
net=readTopologyLevel1(string);
```

  WARNING: it is preferable to end the command with `;` to avoid printing to screen some inner details on the network object.

- **tipLabels:** Prints list of taxon names in the `HybridNetwork` object.
  Usage:

```
tipLabels(net)
```

- **writeTopology:** Function to write the parenthetical format of a `HybridNetwork` object. The first argument is required and it is a `HybridNetwork` object. It has four optional arguments:

  - *di=true*: to print in a format that Dendroscope can read, that is, without the γ values (default false)
  - *outgroup=taxon name*: to root by the outgroup before printing (default none). The outgroup has to be a single taxon. We will update to allow a clade in the future.
  - *names=true*: to print the taxon names in the leaves as opposed to the node numbers (default true).

  Usage:

```
writeTopology(net)
writeTopology(net,di=true)
```

- **root!:** Function to root a network at a node or outgroup (single taxon). When a node number is used as parameter, you have the *resolve* option. If true, a branch with zero length is added. The node numbers can be known with the plot function described in section 7.2. All arguments are required.
  Usage:

```
root!(net,nodeNumber,resolve)
root!(net,outgroup)
```

- **deleteLeaf!:** Function to delete a leaf from a `HybridNetwork` object. Both arguments are required.
  Usage:

```
deleteLeaf!(net,taxonName)
```

- **printEdges:** Function to print out the information on all the edges in a `HybridNetwork` object. Not all information is useful for the user, but information like *edge length* and *gamma* are printed.
  Usage:

```
printEdges(net)
```

- **printNodes:** Function to print out the information on all the nodes in a `HybridNetwork` object. Not all information is useful for the user, but information like *node number* is printed.
  Usage:

```
printNodes(net)
```

6 SNaQ: estimation of phylogenetic network

6.1 Read your data

`SNaQ` estimates a phylogenetic network with the statistical methodology described in (Solís-Lemus and Ané, 2015). There are two possible input data:

- List of estimated gene trees: Estimated gene trees can be obtained from sequence data using RAxML (Stamatakis, 2014) or MrBayes (Huelsenbeck and Ronquist, 2001). Trees need to be in parenthetical format. Other formats will be available in future versions of the package.
- Table of estimated concordance factors (CF): Estimated CF can be obtained from estimated gene trees with BUCKy Ané et al. (2007).

The methodology also requires a starting topology for the search. This can be read from parenthetical format.
Functions to read data into `Julia`:

- **readTrees2CF:** function that read a text file with a list of trees in parenthetical format (one tree per line, but it can have extra lines like headline that the function will ignore. It will assume that any line starting with "(" is a tree). The function will calculate the observed CF for the corresponding 4-taxon subsets. This function has the following optional arguments:

- *quartetfile* (default empty): the list of desired 4-taxon subsets to analyze can be given in a text file. If not such file is specified, the function will take either all the possible 4-taxon subsets or a random sample based on the following options.
- *whichQ* (default "all"): if set to *"rand"*, the 4-taxon subsets will be chosen at random. If *quartetfile* was defined, then the random sample is taken from the 4-taxon subsets in that file. If not, then the sample is drawn from all the possible 4-taxon subsets for the taxa in the input trees ($\binom{n}{x}$). The number of subsets to be chosen is set by the argument *numQ*. If no such number is defined, by default the function will set it as 10% of the total number of possible 4-taxon subsets. **Important:** If *numQ* is specified, but *whichQ* is not set to "rand", then *numQ* will be ignored.
- *CFfile* (default "tableCF.txt"): name of the file to save the computed observed CF from the input trees
- *writetab* (default true): if set to false, then no table of observed CF is written.
- *writeFile* (default false): if set to true, then text files with the list of all 4-taxon subsets and sampled subsets are created. Keep in mind that these files can be very big depending on the number of taxa.
- *taxa* (default union of all taxa in the tree file): you can choose the set of taxa for the quartets with the argument (vector of string names for chosen taxa).

The function does not need all the arguments, the optional arguments are followed by an equal sign `=`. See below examples for details.

WARNING: This function has not yet been tested with missing data. That is, it has been tested in examples where all the trees have the same taxa.
Usage:

```
readTrees2CF(treefile, quartetfile=..., whichQ=..., numQ=...,
             writetab=..., CFfile=..., writeFile=...)
```

- **readTableCF:** Function to read a table of observed CF. The table should contain 7 columns as in Table 1. If the table has more than 7 columns, all columns from 8th on will be ignored. The required argument is the name of the table file, but it also has an optional argument *sep* for the character that separates columns in the table file.

Taxon1	Taxon2	Taxon3	Taxon4	$CF12\|34$	$CF13\|24$	$CF14\|23$

Table 1: Example of column format for the table of CF to be used as input data

Usage:

```
readTableCF(filename);
readTableCF(filename, sep=';');
```

WARNING: It is important to use single quotes: ' not double quotes " when specifying the separator in the table.

Both functions `readTrees2CF` and `readTableCF` return a data structure called `DataCF` that contains the following attributes:

- quartet: Array with the 4-taxon subsets either read from a table of CF or chosen to be analyzed from a list of trees.
- numQuartets: Number of 4-taxon subsets
- tree: Array of `HybridNetwork` objects that represent the list of estimated trees read. If the input data was not a list of trees, this attribute will be empty.
- numTrees: Number of trees read.

- **readInputTrees:** read a text file with a list of trees in parenthetical format (one per line, ignores any line that does not start on "(") and returns a vector of trees.
  Usage:

```
readInputTrees(filename)
```

- **summarizeCFdata:** takes as input a data structure `DataCF` (from previous functions) and provides a few descriptive information. By default, it prints the information on the screen, but it can be saved to a file. The option *pc* is used only when CF were computed from a collection of gene trees: CF for 4-taxon subsets that were computed with less than *pc* percentage of the gene trees will be listed. The option *pc* should be a number between 0 and 1. The required argument is *d*, the optional arguments are *filename,pc* (followed by = below).
  Usage:

```
summarizeCFdata(d)
summarizeCFdata(d,filename=...)
summarizeCFdata(d,filename=...,pc=...)
```

- **readStartTop:** read a tree in parenthetical format from a text file and updates its branch lenghts with the CF data on the data structure `DataCF`. It is equivalent to run readTopologyUpdate first and then updateBL!(net,d)
  WARNING: updateBL only works for a tree topology, not proven for a network yet. Both arguments are required.
  Usage:

```
readStartTop(treefile, d)
```

6.1.1 Small example on reading data

All the files for this section can be downloaded from the *examples* folder in github repository. You need to have them in your working directory. Suppose you have a file with a list of gene trees called *treefile.txt* and you want to use all the possible 4-taxon subsets for the taxa in those trees to calculate the CF:

```
d1=readTrees2CF("treefile.txt")
```

If you want to use a random sample of 100 4-taxon subsets:

```
d2=readTrees2CF("treefile.txt",whichQ="rand",numQ=100)
```

On the contrary, if you have already the CF in a file *tableCF.txt* in the format on Table 1, you would read it like:

```
d=readTableCF("tableCF.txt")
```

If you have a tree *startTree.tre* in parenthetical format to use as starting point for the optimization and want to update the branch lengths according to the CF already read in the data structure *d*:

```
T=readStartTop("startTree.tre",d);
```

6.2 Estimation method

The function `snaq!` runs the estimation method described in (Solís-Lemus and Ané, 2015) and it needs two parameters: starting topology and a data object `DataCF`:

```
snaq!(startingTopology,data)
```

The function has the following optional arguments (with default values in parenthesis):

- **Nfail** *(100)*: number of proposal failed allowed before stopping the optimization
- **hmax** *(1)*: maximum number of hybridizations allowed
- **ftolRel** *(1e-5)*: relative tolerance on the function for the numerical optimization
- **ftolAbs** *(1e-6)*: absolute tolerance on the function for the numerical optimization
- **xtolRel** *(1e-3)*: relative tolerance on the parameters for the numerical optimization
- **xtolAbs** *(1e-4)*: absolute tolerance on the parameters for the numerical optimization
- **verbose** *(false)*: if true, it prints the details of the numerical optimization
- **runs** *(10)*: number of independent starting points. The first two runs start with the starting topology, and subsequent runs modify it by an NNI move. The addition of the initial hybridizations (if the starting topology has fewer hybridizations than *hmax* are done at random and the seeds are saved)

- **outgroup** *(none)*: name of outgroup taxon to root the estimated network at the end
- **filename** *(snaq)*: rootname for the output files: .log, .out, .err
- **returnNet** *(true)*: if true, the `snaq` function returns the resulting `HybridNetwork` object. If false, the resulting network is only written in the .out file.
- **seed** *(0)*: seed to replicate the analyses. This is the main seed from which one seed per run will be drawn randomly. To replicate the results for all the runs, simply set the same seed. If you want to replicate the results of a given run, set *runs=1* and as seed the seed reported in the log file for the given run. With default, the clock time is used to define the main seed.
- **probST** *(0.3)*: probability to use the starting topology as the starting point of each run. To improve the optimization, it is important that each run starts in a different place. At the beginning of every run, a biased coin is thrown so that with probability *probST*, the starting topology is not changed and with probability *1-probST*, an NNI move is performed on the starting topology. If on top the starting topology is a network, a second biased coin is flipped with the same probability so that with probabiliy *1-probST* we propose half the times move origin or and half the times move target.

WARNING: the method does not currently have a way to control for the complexity of the network, so it is important to try to avoid overparametrizing the model by selecting a very big `hmax` from start. It is best to start with `hmax=1`, and increase arithmetically trying to keep the estimated network interpretable. Keep in mind also that the fact that we estimate *level 1 networks* means that the hybridization cycles cannot overlap. If the number of taxa is small, and `hmax` is set very big, then the method will not be able to place that many hybridizations.

The function ends with ! because it modifies the DataCF object by including the expected CF.

6.2.1 Small example on estimating phylogenetic network

You have d, T from previous example (6.1.1), you can estimate the phylogenetic network by:

```
net=snaq!(T,d);
net=snaq!(T,d,hmax=2);
```

These runs might take a few minutes. The estimated network will be stored in `net`, so that the user that write in parenthetical format, re root or plot:

```
PhyloNetworks.plotPhylonet(net)
```

More on the plot function below. There will also be output files created with information on the estimation procedure.

6.3 Read .out file

- **readSnaqNetwork:** reads the .out file generated by the `snaq!` function and returns a `HybridNetwork` object.
  Usage:

```
readSnaqNetwork(outfile)
```

The .log file contains information on the heuristic optimization such as the seed for each run, the reason why the optimization stopped (there are different criteria, mostly either there have been too many failed proposals or the likelihood is not changing anymore), the number of iterations needed for the algorithm to converge, the value of the likelihood for the estimated network, and the number of moves proposed and accepted.

6.4 Multiple alleles

The usual settings for SNaQ consider each allele in the gene trees to be its own tip in the network, however, if there is a known mapping file of allele names to species, and only the species-level network wants to be estimated, this can be done with the following functions:

- **mapAllelesCFtable:** This function will read the original CF table with allele names, and a mapping file matching allele names to species names, and return a CF data frame with the allele names changed to species names. If the optional argument *filename* is defined, then the new table is also saved as a csv file.
  Usage:

```
new_df = mapAllelesCFtable(mappingFile, CFtable);
new_df = mapAllelesCFtable(mappingFile, CFtable,
         filename="newTable.txt");
```

  This dataframe should be read also to create a DataCF object:

```
new_d = readTableCF(new_df);
```

  The mapping file should have two columns names *allele* and *species*. The function will prompt an error if the column names do not match. It allows for extra columns, and the function will ignore them. The estimation will work in the same way:

```
new_net = snaq!(newT,new_d);
```

  where *newT* should be a starting topology on the species names.
  WARNING: the current function works best if all alleles from the same individual are given the same name (the individual's 'name') across all genes for which that individual was sequenced.

6.5 Optimizing branch lengths and inheritance probabilities for a given network

- **topologyMaxQPseudolik!:** For a given network, you can optimize the branch lengths and inheritance probabilities with the pseudolikelihood. Minus the logarithm of the pseudolikelihood value for the network will be stored in the attribute: `net.loglik` and it will be printed to screen. The optional argument `verbose` will print the iterations to the screen. The user can also define the absolute and relative tolerance with the same options as described before for the `snaq!` function. The maximum value allowed for branch lengths is 10 (coalescent units).
  Usage:

  ```
  topologyMaxQPseudolik!(net,d)
  topologyMaxQPseudolik!(net,d,verbose=true)
  ```

  This function is useful to compare the pseudolikelihood of different network alternatives, and choose the best one among them.

- **topologyQPseudolik!:** For a given network with branch lengths and inheritance probabilities, you can compute the pseudolikelihood value without optimizing. This function is not maximizing, it is simply computing the pseudolikelihood for the given branch lenghts and probabilities of inheritance.
  Usage:

  ```
  topologyQPseudolik!(net,d)
  topologyQPseudolik!(net,d,verbose=true)
  ```

6.6 Debugging: the .err file

`SNaQ` is a complex computational algorithm, so despite our best efforts, there are probably many undetected bugs and errors. The user can be extremely helpful in fixing this. After you do any analysis, please check the .err file to check how many runs failed because of a bug:

```
Total errors: 1 in seeds [4545]
```

The seed that caused the error and the description of the error (which will not necessarily be informative for the user) will be listed in the .log file. To help us out to debug `SNaQ`, please use the same settings under which you found the error to run the function and the seed in the .err file associated to the bug:

```
snaqDebug(T,d,hmax=2,seed=4545);
```

This will create two text files: *snaqDebug.log* and *debug.log*.

You can send them to *claudia@stat.wisc.edu* with subject *Snaq bug found* or something similar. I will not have access to any of your data, the files simply print the steps I need to retrace the bug and hopefully, fix it.

7 Visualization

7.1 Goodness-of-fit plot

We can plot the observed CF to the expected CF of a given estimated network to have a rough idea of the fitness to the data. If a network is a good fit to the data, then the dots in the plot will be close to the $y = x$ line.

- **dfObsExpCF**: function that will take a DataCF object (after running `snaq!`) and will provide a dataframe with the observed and expected CF for plotting.
  Usage:

  ```
  df = dfObsExpCF(d)
  ```

We can now plot the observed and expected CF with any Julia plotting packages. In particular using Gadfly (`http://dcjones.github.io/Gadfly.jl/`):

```
using Gadfly
p = plot(df,layer(x="obsCF1",y="expCF1",Geom.point,
    Theme(default_color=colorant"orange")),
    layer(x="obsCF2",y="expCF2",Geom.point,
    Theme(default_color=colorant"purple")),
    layer(x="obsCF3",y="expCF3",Geom.point,
    Theme(default_color=color("blue"))),
    layer(x=0:1,y=0:1),Geom.line,
    Theme(default_color=color("black")))
```

This will pop out a browser window with the plot. The plot can be saved to PDF file (or many other formats based on the Gadfly tutorial):

```
draw(PDF("plot.pdf", 4inch, 3inch), p)
```

7.2 Phylogenetic Network Visualization

This package contains functionality for creating visual plots of phylogenies generated using the `SNaQ` estimation method (Solís-Lemus and Ané, 2015). The included visualization tools allow the user to plot entire networks with many hybridization events or the underlying tree structure. Plots created by this package include a dynamic representation of probability of inheritance for hybridization events, which are represented by variation in hybrid edge thickness. Numerous customization options are available and are described in detail below.

7.2.1 Basic Network Plotting

Plotting with the `PhyloNetworks` package is simple, being entirely contained in the function `plotPhylonet`. Although there are many optional arguments available for this function, the only *required* input is the network itself. Networks may be input into `plotPhylonet` as one of two possible formats, depending on the user's particular preference:

1. Newick parenthetical format

2. HybridNetwork data type

Calling this function on a network will generate a .svg image file in the user's working directory as well as the corresponding .dot file used for rendering.

The .svg file type can be opened and viewed using a number of different programs including most web browsers (Inkscape, Chrome, Safari, etc.). If the user wishes to use a different image type, there are a number of options available (two of which we will describe here). First, the user can access one the many conversion tools that are available on the web. Many of these websites can convert a .svg image into a wide variety of standard image file types. Another option is to open the corresponding .dot file using GraphViz (which should already be installed) and exporting into the desired format.

7.2.2 Customization options

The `plotPhylonet` function contains a variety of optional arguments that may be used to tailor the output image to a particular use. A complete list of optional arguments is given below along with default values and argument descriptions.

- **mainTree** *(false)*: When true, only the underlying tree structure is plotted as determined by gammaThreshold. Otherwise, the entire network is shown.
- **imageName** *(netImage)*: Name for the output plot.
- **gammaThreshold** *(0.5)*: Set's the lowest gamma value to be included when plotting the underlying tree structure.
- **width** *(6.0)*: Sets the width of the image in inches.
- **height** *(8.0)*: Sets the height of the image in inches.
- **vert** *(true)*: When true, the hierarchy of the plot is directed vertically with the root node being place on top and the leaf nodes on bottom. Otherwise, the hierarchy is directed horizontally with the root on the left and leaf nodes on the right.
- **internalLabels** *(false)*: When true, node labels are included on all internal nodes. Otherwise, they are only included for leaf nodes.
- **fontSize** *(16.0)*: Sets the font size for node and edge labels in points.
- **layoutStyle** *("dot")*: Chooses the layout engine used by GraphViz for determining node and edge placement (more details can be found at `http://www.graphviz.org/Home.php`. Alternative options include "neato", "fdp", "sfdp", "circo", and "twopi".

- **hybridColor** *("green4")*: Sets the color for hybrid edges. Complete list of color options can be found at `http://www.graphviz.org/doc/info/colors.html`
- **forcedLeaf** *(true)*: When true, leaf nodes are placed on the same level, ranked at the bottom of the network.
- **unrooted** *(false)*: Plots an unrooted network or tree using the *neato* engine.
- **nodeSeparation** *(0.8)*: Sets the minimum distance between any two nodes in inches.
- **edgeStyle** *("line")*: Chooses the edge style used in the plot. Additional options include "ortho", "curved", "composite", "spline", and "false".
- **labelAngle** *(180.0)*: Sets the angle of leaf label placement relative to its parent edge.
- **labelDistance** *(3.0)*: Sets the distance of leaf label placement relative to its corresponding node.
- **includeGamma** *(false)*: When true, gamma labels are included on hybrid edges.
- **IncludeLength** *(false)*: When true, length labels are included on all edges.

7.2.3 Visualization Examples

This section will provide explicit examples for using the visualization tools provided in the PhyloNetworks package. Plotting examples will use $(((((1,2))\#H1,(\#H1,(3,4))),5),6)$; as an example network.

Example 1: As previously mentioned, the most simple way to use the included visualization tools is to call the `plotPhylonet()` function with the network as an argument. Using our example network, this would be done by typing in `Julia`

```
PhyloNetworks.plotPhylonet("(((((1,2))#H1,(#H1,(3,4))),5),6);")
```

This will result in the image below being saved in your working directory as `netImage.svg`. The file name can be pre-specified when calling `plotPhylonet` by including the argument `imageName="newfilename"`, which saves the plot as `newfilename.svg`.
Calling additional customization options is as simple as including additional arguments separated by a comma. The user can combine any number of the available arguments to tailor a plot to their particular example. Given below are a few examples that exhibit the different visualization options available.

Example 2: For the sake of tidiness, we define the network as its own variable, `net`. In addition, we include the arguments `vert = false`, which plots the hierarchy horizontally, `mainTree=true`, which only plots the underlying tree structure, and `fontSize = 20.0`, which increases the label font size from 16.0 to 18.0.

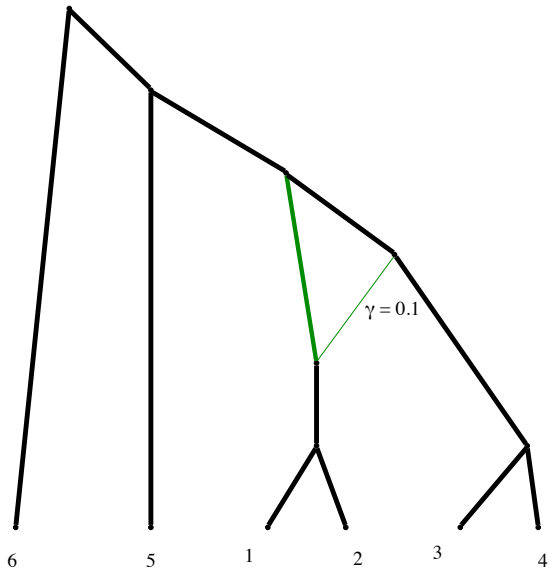


Figure 1: Basic plot using the `plotPhylonet` function. Note that if the gamma value for a hybrid edge is not explicitly defined, it will assume a default value of 0.1.

```
net = "(((((1,2))#H1,(#H1,(3,4))),5),6);"
PhyloNetworks.plotPhylonet(net, vert = false,
                           mainTree = true, fontSize = 20.0)
```

7.2.4 Style Notes

Although there are default argument values given by `plotPhylonet`, they do not always result in the ideal plot for a particular example. Many of the included arguments were included for the purpose of the user being able to adjust certain layout parameters to best fit their own network. In particular, there is sometimes difficulty in neatly placing and orienting leaf labels and gamma labels. This is especially noticeable as the number of leaf nodes becomes large or if the the names associated with leaf nodes are long. We have included some tips for fixing common issues below.

- To avoid edges overlapping gamma labels, include the argument `edgeStyle = true`. This will allow the layout engine to include curved splines, which will avoid overlaps.
- If leaf names are long, include the `vert = false` argument to set horizontal hierarchy.
- Label overlap can also be finely altered by changing the `labelDistance` and `labelAngle` arguments.

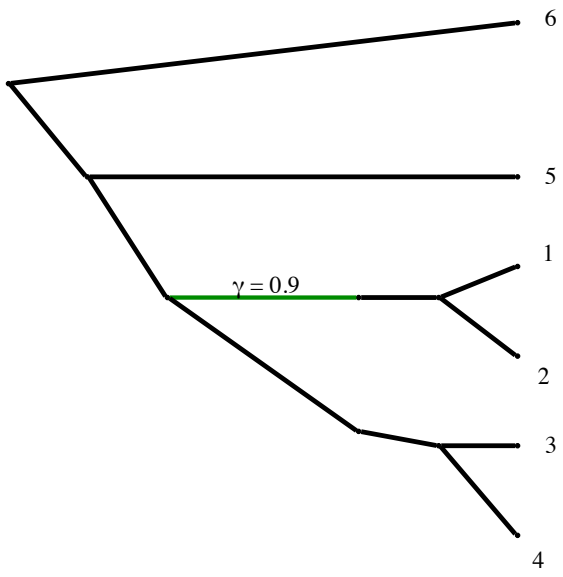


Figure 2: Plot of the underlying tree structure, oriented horizontally, with a changed font size.

- Issues between in text readability can be fixed by adjusting the `fontSize`, `height`, or `width`.
- Although the arguments `layoutStyle` and `edgeStyle` have been included, some of options available are not guaranteed to be ideal for certain network plots.

When plotting, a known issue can occur if Julia and GraphViz are not linked. To test the GraphViz installation, see the following section:

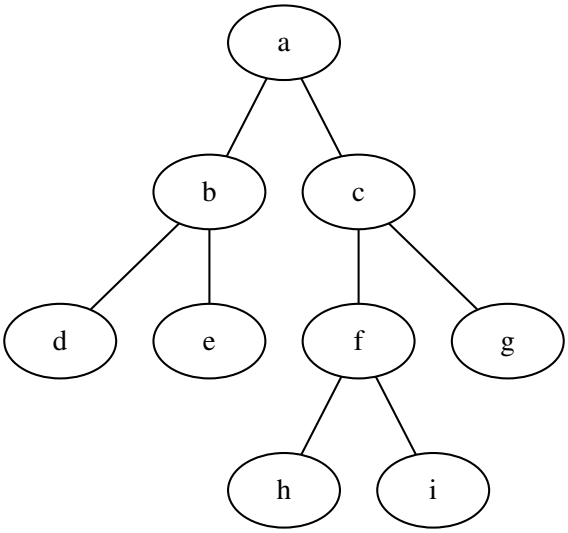
7.2.5 Example to test correct `GraphViz` installation

The `GraphViz` package installs the program `GraphViz`. To verify that the link between `Julia` and `GraphViz` is working properly. Everything should be done automatically, but it is worth testing. Open `Julia` and type

```
s=open("graph.dot","w")
write(s,"graph {
                a -- b;
                a -- c;
                b -- d;
                b -- e;
                c -- f;
```

```
                    c -- g;
        f -- h;
        f -- i;
        }")
close(s)
PhyloNetworks.generalExport("graph.dot")
```

This will turn `graph.dot` into a file called `genImage.svg` in the working directory.
If this worked correctly, the console should display a series of prompts indicating its completion. A file called `scratchimage.svg` should be located in your working directory.



Plot of the underlying tree structure, oriented horizontally, with a changed font size.

WARNING: There is a known bug in the `plotPhylonet` function, see the issue in the `PhyloNetworks` Github repository for details. The error can be sometimes fixed by changing the position of the root with the `root` function.

References

Ané C, Larget B, Baum DA, Smith SD, Rokas A. 2007. Bayesian estimation of concordance among gene trees. *Molecular biology and evolution*. 24:412–26.

Huelsenbeck JP, Ronquist F. 2001. MrBayes: Bayesian inference of phylogeny. *Bioinformatics*. 17:754–755.

Huson D, Rupp R, Scornavacca C. 2010. Phylogenetic Networks. New York, NY: Cambridge University Press, first edition.

Pardi F, Scornavacca C. 2015. Reconstructible Phylogenetic Networks: Do Not Distinguish the Indistinguishable. *PLOS Computational Biology*. 11:e1004135.

Solís-Lemus C, Ané C. 2015. Inferring phylogenetic networks with maximum pseudolikelihood under incomplete lineage sorting. *arXiv*. pp. 1–32.

Stamatakis A. 2014. {RAxML} version 8: a tool for phylogenetic analysis and post-analysis of large phylogenies. *Bioinformatics*. 30:1312–1313.

A Definition of Phylogenetic Network

For the present work, we will use the following definition (but refer to Huson et al. (2010) for other types of evolutionary networks). A *rooted phylogenetic network* for a set of taxa X is a connected directed acyclic graph with vertices $V = V_L \cup V_H \cup V_T$, edges $E = E_H \cup E_T$ and a bijective leaf-labeling function $f : V_L \to X$ with the following characteristics:

- The root r has indegree$(r) = 0$ and outdegree$(r) = 2$.
- For any $v \in V_L$ (leaf), $indegree(v) = 1$ and $outdegree(v) = 0$.
- For any $v \in V_T$ (tree node), $indegree(v) = 1$ and $outdegree(v) = 2$.
- For any $v \in V_H$ (hybrid node), $indegree(v) = 2$ and $outdegree(v) = 1$.
- A tree edge $e \in E_T$ is an edge whose child is a tree node.
- A hybrid edge $e \in E_H$ is an edge whose child is a hybrid node.

Thus, we are not allowing internal nodes with only two edges, nor polytomies. We also do not allow a leaf to be hybrid node, and only 2 hybrid edges per hybrid node.

We also assume that the hybridization cycles do not intersect, and these networks are called *level-1 networks* (Huson et al., 2010), which have been shown to be identifiable (Pardi and Scornavacca, 2015; Solís-Lemus and Ané, 2015). These restrictions are enforced in the optimization, but not in the read/write/plot/root functions. However, the only rule strictly enforced for all functions is that a hybrid node cannot have more than two hybrid edges pointing at it. This is a restriction that we hope to eliminate in the future.