Phylogenetic Tree processing HOWTO

Jason Stajich
Dept Molecular Genetics and Microbiology, Duke University

jason AT bioperl.org

This HOWTO is intended to show usage of the Bioperl Tree objects for manipulate phylogenetic trees. It shows how to read and write trees, query them for information about specific nodes or overall statistics. Advanced topics include discussion of generating random trees and extensions of the basic structure for integration in other modules in Bioperl.

1. Introduction

Generating and manipulating phylogenetic trees is an important part of modern systematics and molecular evolution research. The construction of trees is the subject of a rich literature and active research. This HOWTO and the modules described within are focused on querying and manipulating trees once they have been created.

The data we intend to capture with these objects is for notion of Trees and their Nodes. A Tree is made up of Nodes and the relationships which connect these nodes. The basic representation of parent and child nodes which are intended to represent the directionality of evolution. This is to capture the idea that some ancestor species gave rise, though speciation events, to a number of children species. The data in the trees need not be a strictly bifurcating tree (or binary trees to the CS types), a parent node can give rise to 1 or many children nodes.

In practice this this implementation there are just a few main objects (or modules) you need to know about. There is the main Tree object *Bio::Tree::Tree* which is main entry point to the data represented by a tree. A Node is represented generically by *Bio::Tree::Node*, however there are subclasses of this object to handle particular cases where we need a richer object (see *Bio::PopGen::Simulations::Coalescent* and the PopGen HOWTO for more information). Nodes are connected to one another through a few simple data. There is the concept of pointers or references where a particular Node keeps track of who its parent is and its children. A Node can only have one parent and it can have 1 or many children. In fact all of the information in a tree pertaining to the relationships between Nodes and specific data like bootstrap values and labels are all stored in the Node objects while the *Bio::Tree::Tree* object is just a container for some summary information about the tree and keeps track of the tree's root node.

2. Simple Usage

Trees are used to representations of the history of a collection of taxa, sequences, or populations.

3. Reading and Writing Trees

Using *Bio::TreeIO* one can read trees from files or datastreams and create*Bio::Tree::Tree* objects. This is analagous to how we read sequences from sequence files with *Bio::SeqIO* to create Bioperl sequence objects which can be queried and manipulated. Similarly we can write *Bio::Tree::Tree* objects out to string representation like the newick or new hampshire format which can be then printed to a file, datastream, stored in database, etc.

The main module for reading and writing trees is the *Bio::TreeIO* factory module which has several driver modules which plug into it. These drivers include *Bio::TreeIO::newick* for New Hampshire or Newick format, *Bio::TreeIO::nhx* for New Hampshire eXtended format from Sean Eddy and Christian Zmeck as part of their RIO and ATV system [reference here]. The driver *Bio::TreeIO::nexus* supports parsing tree data from PAUP's Nexus format, however this driver currently only supports parsing and not writing of nexus format tree files.

4. Example Code

Here is some code which will read in a Tree from file called "tree.tre" and produce a Bio::Tree::Tree object which is stored in the variable *$tree*.

Like most modules which do input/output you can also specify the argument -fh in place of -file provide a glob or filehandle in place of the filename.

```
use Bio::TreeIO;
# parse in newick/new hampshire format
my $input = new Bio::TreeIO(-file   => "tree.tre",
                            -format => "newick");
my $tree = $input->next_tree;
```

Once you have a Tree object you can do a number of things with it. These are all methods required in *Bio::Tree::TreeI*

4.1. Bio::Tree::TreeI methods

Request the taxa (leaves of the tree)

```
my @taxa = $tree->get_leaf_nodes;
```

Get the root node.

```
my $root = $tree->get_root_node;
```

Get the total length of the tree (sum of all the branch lengths), only useful if the nodes actually have branch length stored of course.

```
my $total_length = $tree->total_branch_length;
```

4.2. Bio::Tree::TreeFunctionsI

An additional interface was written which implements utility functions which are useful for manipulating a Tree.

Find a particular node, either by name or some other field that is stored in a Node. The field type should be the function name we can call on all of the Nodes in the Tree.

```
   # find all the nodes named ’node1’ (there should be only one)
   my @nodes = $tree->find_node(-id => ’node1’);
   # find all the nodes which have description ’BMP’
   my @nodes = $tree->find_node(-description => ’BMP’);
   # find all the nodes with bootstrap value of 70
   my @nodes = $tree->find_node(-bootstrap => 70);
```

If you would like to do more sophisticated searches, like find all the nodes with boostrap value better than 70 you can easily implement this yourself.

```
     my @nodes = grep { $_->bootstrap > 70 } $tree->get_nodes;
```

Remove a Node from the Tree (and update the children/ancestor links where the Node is an intervening one)

```
# provide the node object to remove from the Tree
$tree->remove_Node($node);
# or specify the node Name to remove
$tree->remove_Node('Node12');
```

Get the lowest common ancestor for a set of Nodes. This method is used to find and internal Node of the Tree which can be traced, through its children, to the requested set of Nodes. It is used in tha calculations of monophyly and paraphyly and in determining the distance between two nodes.

```
# Provide a list of Nodes that are in this tree
my $lca = $tree->get_lca(-nodes => \@nodes);
```

Get the distance between two nodes by adding up the branch lengths of all the connecting edges between two nodes.

```
my $distances = $tree->distance(-nodes => [$node1,$node2]);
```

Perform a test of monophyly for a set of nodes and a given outgroup node. This means the common ancestor for the members of the internal_nodes group is more recent than the common ancestor any of them share with the outgroup node.

```
if( $tree->is_monophyletic(-nodes    => \@internal_nodes,
                           -outgroup => $outgroup) ) {
  print "these nodes are monophyletic: ",
       join(",",map { $_->id } @internal_nodes ), "\n";
}
```

Perform a test of paraphyly for a set of nodes and a given outgroup node. This means a common ancestor 'A' for the members of the group is more recent than a common ancestor 'B' of them share with the outgroup node AND there are no other nodes in the tree which have 'A' as a common ancestor before 'B'.

```
if( $tree->is_paraphyletic(-nodes    => \@internal_nodes,
                           -outgroup => $outgroup) ) {
  print "these nodes are monophyletic: ",
       join(",",map { $_->id } @internal_nodes ), "\n";
}
```

Reroot a tree, specifying a different node as the root (and a different node as the outgroup).

```
# node can either be a Leaf node in which case it becomes the
# outgroup and its ancestor is the new root of the tree
# or it can be an internal node which will become the new
# root of the Tree
$tree->reroot($node);
```

5. Advanced Topics

It is possible to generate random tree topologies with a Bioperl object called *Bio::Tree::RandomFactory*. The only factory requires the specification of the total number of taxa to simulate a history for. Once can request different methods for generating the random phylogeny, at present, however only the simple Yule backward is implemented and is the default choice. The trees can be generated with the following code. You can either specify the names of taxa or just a count of total number of taxa in the simulation.

```
use Bio::TreeIO;
use Bio::Tree::RandomFactory;
# initialize a TreeIO writer to output the trees as we create them
my $out = Bio::TreeIO->new(-format => ’newick’,
                           -file   => ">randomtrees.tre");
my @listoftaxa = qw(A B C D E F G H);
my $factory = new Bio::Tree::RandomFactory(-taxa => \@listoftaxa);
# generate 10 random trees
for( my $i = 0; $i < 10; $i++ ) {
   $out->write_tree($factory->next_tree);
}
# Once can also just request a total number of taxa (8 here) and
# not provide labels for them
# In addition one can specify the total number of trees
# the object should return so we can call this in a while
# loop
$factory = new Bio::Tree::RandomFactory(-num_taxa => 8
                                        -max_count=> 10);
while( my $tree = $factory->next_tree) {
  $out->write_tree($tree);
}
```

There are more sophisticated operations that you may wish to pursue with these objects. We have tried to create a framework for this type of data, but by no means should this be looked at as the final product. If you have a particular statistic or function that applies to trees that you would like to see included in the toolkit, we encourage you to send details to the Bioperl list.

6. References and More Reading

For more reading and some references for the techniques above see these titles.