An introduction to the python Environment for Tree Exploration (ETE)

by Jaime Huerta-Cepas

November 27, 2009

About ETE

ETE is a python programming toolkit that assists in the automated manipulation, analysis and visualization of hierarchical trees. Besides a broad set of tree handling options, ETE's current version provides specific methods to analyze phylogenetic and clustering trees. It also supports large tree data structures, node annotation, independent editing and analysis of tree partitions, and the association of trees with external data such as multiple sequence alignments or numerical matrices.

ETE is currently developed at the comparative genomics group in the Centre for Genomic Regulation (CRG). Barcelona, Spain.

You can cite ETE as:

Jaime Huerta-Cepas, Joaquín Dopazo and Toni Gabaldón. **ETE: A python Environment for Tree Exploration**. (2009, unpublished) (check http://ete.cgenomics.org for updates)

About this tutorial

This document compiles a set of guided examples about general ETE usage. It does not aim to cover exhaustively all available ETE's methods and parameters, but to illustrate most important aspects of each module. A comprehensive reference guide for programmers can also be found at http://ete.cgenomics.org/. In addition, examples used in this tutorial and documentation updates can be downloaded from the same web site.

CHAPTER 1

WORKING WITH TREE DATA STRUCTURES

According to the wikipedia (www.wikipedia.org, 2009), trees are a widely-used type of data structure that emulates a tree design with a set of linked nodes. Formally, a tree is considered an acyclic and connected graph. Each node in a tree has zero or more child nodes, which are below it in the tree (by convention, trees grow down, not up as they do in nature). A node that has a child is called the child's parent node (or ancestor node, or superior). A node has at most one parent.

The height of a node is the length of the longest downward path to a leaf from that node. The height of the root is the height of the tree. The depth of a node is the length of the path to its root (i.e., its root path).

- The topmost node in a tree is called the root node. Being the topmost node, the root node will not have parents. It is the node at which operations on the tree commonly begin (although some algorithms begin with the leaf nodes and work up ending at the root). All other nodes can be reached from it by following edges or links. Every node in a tree can be seen as the root node of the subtree rooted at that node.

- Nodes at the bottommost level of the tree are called leaf nodes. Since they are at the bottommost level, they do not have any children.

- An internal node or inner node is any node of a tree that has child nodes and is thus not a leaf node.

- A subtree is a portion of a tree data structure that can be viewed as a complete tree in itself. Any node in a tree T, together with all the nodes below it, comprise a subtree of T. The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree (in analogy to the term proper subset).

In bioinformatics, trees are the result of many analyses, such as phylogenetics or clustering. Although each case entails specific considerations, many properties remains constant among them. The Environment for Tree Exploration is a python toolkit that assists in the automated manipulation, analysis and visualization of hierarchical trees. Besides general tree handling options, ETE's current version provides specific methods to analyze phylogenetic and clustering trees. Moreover a programmable tree drawing engine is implemented that can be used to automatize the graphical rendering of trees with customized node-specific visualizations.

1.1 Understanding Tree Topology

A tree is a succession of TreeNodes connected in a hierarchical way. Therefore, the topology of a tree is defined by the connections of its nodes. Each node instance has two basic attributes: **node.up** and **node.children**. Up is a pointer to the parent's node, while children is the list of nodes hanging for the current node instance. Although it is possible to modify the structure of a tree by changing these attributes, we strongly recommend not to do it. Several methods are provided to manipulate each node's connections in a safe way (see 1.6). Other three attributes are always present in a tree node instance: i) **node.support**, which stores the branch support of the partition defined by a given node (i.e. bootstrap value) ii) **node.dist**, which stores the branch length/distance from a given node to its parent iii) and **node.name**, which stores the node's name. Thus, a tree is internally encoded as a succession of node instances connected in a hierarchical way. Given that all nodes in a tree are the same type, any node of the tree contains the same basic methods and attributes.

When a tree is loaded, a pointer to the top-most node is returned. This is called the tree root, and it exists even if the tree is considered as theoretically unrooted. The root node can be considered as the master tree node, since it represents the whole tree structure. Internal an terminal nodes (all hanging from the master node) represent different partitions of the tree, and therefore can be used to access or manipulate all the possible sub-trees within a general tree structure. Thus, once a tree is loaded, it can be split in different subparts that will function as independent trees instances.

In order to evaluate the basic attributes of a node, you can use the following methods: **tree.is_leaf(),** returns True if node has no children; **tree.is_root()**, returns True if node has no parent; **len(node)**, returns the number of terminal nodes (leaves) under a given internal node; and **len(node.children)**, returns the number of node's children. Additionally tree node instances can be queried or iterated as normal built-in python objects. For example, the following operations are allowed i) **[for leaf in node if leaf.dist>0.5]** ii) **if leaf in node: print "true"** iii) **print tree**. The following example illustrates some basic things we can do with trees:

Example 1

```
from ete2 import Tree
# Creates an empty tree and populates it with some new
# nodes
t = Tree()
A = t.add_child(name="A")
B = t.add_child(name="B")
C = A.add_child(name="C")
D = A.add_child(name="D")
print t
#                    /-C
#          /--------|
#---------|          \-D
#         |
#          \-B
print 'is "t" the root?', t.is_root() # True
print 'is "A" a terminal node?', A.is_leaf() # False
print 'is "B" a terminal node?', B.is_leaf() # True
print 'B.get_tree_root() is "t"?', B.get_tree_root() is t # True
print 'Number of leaves in tree:', len(t) # returns number of leaves under node (3)
print 'is C in tree?', C in t # Returns true
print "All leaf names in tree:", [node.name for node in t]
```

1.2 Reading and Writing Newick Trees

The Newick format is the standard representation of trees in computer science. It uses nested parentheses to represent hierarchical data structures as text strings. The original newick standard is able to encode information about the tree topology, branch distances and node names. However different variants of this format are used by different programs. ETE supports many of these formats both for reading and writing operations. Currently, this is the list of supported newick formats:

Example	Description	Format
(,(,(,)));	topology only	100 (strict)
(A,(B,(D,G)));	leaf names	9 (strict)
(A,(B,(D,G)E)C);	all names	8 (strict)
(A:0.35,(B:0.72,(D:0.60,G:0.12)E)C);	all names, leaf distances	7 (strict)
(A,(B,(D,G):0.64):0.56);	leaf names, node distances	6 (strict)
(A:0.35,(B:0.72,(D:0.60,G:0.12):0.64):0.56);	leaf names, all distances	5 (strict)
(A:0.35,(B:0.72,(D:0.60,G:0.12)));	leaf names, leaf distances	4 (strict)
(A:0.35,(B:0.72,(D:0.60,G:0.12)E:0.64)C:0.56);	leaf names, all distances	3 (strict)
(A:0.35,(B:0.72,(D:0.60,G:0.12)1.0:0.64)1.0:0.56);	leaf names, all distances, branch support	2 (strict)
(A:0.35,(B:0.72,(D:0.60,G:0.12)E:0.64)C:0.56);	all names, all distances	1 (flexible)
(A:0.35,(B:0.72,(D:0.60,G:0.12)1.0:0.64)1.0:0.56);	leaf names, all distances, branch support	0 (flexible) **(default)**

Table 1.1: List of supported newick formats.

Formats labeled as **flexible** allow missing information. For instance, format 0 will be able to load a newick tree even if they not contain branch support information (it will be initialized with the default value), however, format 2 will rise a parsing error. If you want to control that your newick files strictly follow a given pattern you should use **strict** format definitions.

Reading newick trees

In order to load a tree from a newick text string you can use the constructor **Tree()**, provided by the main module **ete2**. You only need to pass a text string containing the newick structure and the format that should be used to parse it (0 by default). Alternatively, you can pass the path to a text file containing the newick string.

Example 2

```
from ete2 import Tree
# Loads a tree structure from a newick string. The returned variable
# 't' is the root node for the tree.
t = Tree('(A:1,(B:1,(E:1,D:1):0.5):0.5);' )
# Load a tree structure from a newick file.
t = Tree('genes_tree.nh')
# You can also specify the newick format. For instance, for named
# internal nodes we will format 1.
t = Tree('(A:1,(B:1,(E:1,D:1)Internal_1:0.5)Internal_2:0.5)Root;', format=1)
```

Writing newick trees

Any ETE tree can be exported to the newick standard. To do it, you must call the method **write(),** present in any Tree node instance, and choose the preferred format (0 by default). This method will return the text string representing a given tree. You can also specify an output file.

Example 3

```
from ete2 import Tree
# Loads a tree with internal node names
t = Tree('(A:1,(B:1,(E:1,D:1)Internal_1:0.5)Internal_2:0.5)Root;', format=1)
# And prints its newick representation omiting all the information but
# the tree topology
print t.write(format=100) # (,(,(,)));
# We can also write into a file
t.write(format=100, outfile="/tmp/tree.new")
```

1.3 Some Basis on ETE's trees

Once loaded into the Environment for Tree Exploration, trees can be manipulated as normal python objects. Given that a tree is actually a collection of nodes connected in a hierarchical way, what you usually see as a tree will be the root node instance from which the tree structure is hanging. However, every node within a ETE's tree structure can be also considered a subtree. This means, for example, that all the operational methods that we will review in the following sections are available at any possible level within a tree. Moreover, this feature will allow you to separate large trees into smaller partitions, or concatenate several trees into a single structure.

1.4 Browsing trees

One of the most basic operations related with tree analysis is tree browsing, which is, essentially, the task of visiting nodes within a tree. In order to facilitate this, ETE provides a number of methods to search for specific nodes or to navigate over the hierarchical structure of a tree.

1.4.1 Getting leaves, Descendants and Node's Relatives

The list of internal or leaf nodes within a given partition can be obtained by using the **get_leaves()** and **get_descendants()** methods. The former will return the list of terminal nodes (leaves) under a given internal node, while **get_descendants()** will return the list of all nodes (terminal and internal) under a given tree node. You can iterate over the returned list of nodes or filter those meeting certain properties.

Example 4

```
from ete2 import Tree
# Loads a basic tree
t = Tree( '(A:0.2,(B:0.4,(C:1.1,D:0.45):0.5):0.1);' )
print t
#          /-A
#---------|
#         |          /-B
#          \--------|
#                   |          /-C
#                    \--------|
#                              \-D
# Counts leaves within the tree
nleaves = 0
for leaf in t.get_leaves():
    nleaves += 1
print "This tree has", nleaves, "terminal nodes"
# But, like this is much simpler :)
nleaves = len(t)
print "This tree has", nleaves, "terminal nodes [proper way: len(tree) ]"
# Counts leaves within the tree
ninternal = 0
for node in t.get_descendants():
    if not node.is_leaf():
 ninternal +=1
print "This tree has", ninternal,  "internal nodes"
# Counts nodes with whose distance is higher than 0.3
nnodes = 0
for node in t.get_descendants():
    if node.dist >  0.3:
 nnodes +=1
# or, translated into a better pythonic
nnodes = len([n for n in t.get_descendants() if n.dist>0.3])
print "This tree has", nnodes,  "nodes with a branch length > 0.3"
```

In addition, other methods are available to find nodes according to their hierarchical relationships, namely: **get_sisters()** , **get_children()** and **get_common_ancestor()**. Note that get_children returns an independent list of children rather than the **node.children** attribute. This allows you to operate with such list without affecting the integrity of the tree. The **get_common_ancestor()** method is specially useful for finding internal nodes, since it allows to search for the first internal node that connects several leaf nodes.

Example 5

```
from ete2 import Tree
#Loads a tree
tree = Tree( '((H:1,I:1):0.5, A:1,  (B:1,(C:1,D:1):0.5):0.5);' )
print "this is the original tree:"
print tree
#                    /-H
#          /--------|
#         |          \-I
#         |
#---------|--A
#         |
#         |          /-B
#          \--------|
#                    |          /-C
#                     \--------|
#                              \-D
# Finds the first common ancestor between B and C.
ancestor = tree.get_common_ancestor("D", "C")
print "The ancestor of C and D is:"
print ancestor
#          /-C
#---------|
#          \-D
# You can use more than two nodes in the search
ancestor = tree.get_common_ancestor("B", "C", "D")
print "The ancestor of B, C and D is:"
print ancestor
#          /-B
#---------|
#         |          /-C
#          \--------|
#                    \-D
# Finds the first sister branch of the ancestor node. Because
# multifurcations are allowed, many sister branches are possible.
sisters = ancestor.get_sisters()
print "which has has", len(sisters), "sister nodes"
print "and the first of such sister nodes like this:"
print sisters[0]
#
#          /-H
#---------|
#          \-I
```

1.4.2 Traversing (browsing) trees

Often, when processing trees, all nodes need to be visited. This is called tree traversing. There are different ways to traverse a tree structure depending on the order in which children nodes are visited. ETE implements the two most common strategies: **pre-** and **post-order**. The following scheme shows the differences in the strategy for visiting nodes (note that in both cases the whole tree is browsed):

- preorder
  - Visit the root.
  - Traverse the left subtree.
  - Traverse the right subtree.
- postorder
  - Traverse the left subtree.
  - Traverse the right subtree.
  - Visit the root.

Every node in a tree includes a **traverse()** method, which can be used to visit, one by one, every node node under the current partition.

Example 6

```
from ete2 import Tree
t = Tree( '(A:1,(B:1,(C:1,D:1):0.5):0.5);' )
# Visit nodes in preorder (this is the default strategy)
for n in t.traverse():
    print n
# It Will visit the nodes in the following order:
#            /-A
# ---------|
#           |          /-B
#            \--------|
#                      |          /-C
#                       \--------|
#                                 \-D
# --A
#            /-B
# ---------|
#           |          /-C
#            \--------|
#                      \-D
# --B
#            /-C
# ---------|
#            \-D
# --C
# --D
# Visit nodes in postorder
for n in t.traverse("postorder"):
    print n
# It Will visit the nodes in the following order:
# --A
# --B
# --C
# --D
#            /-C
# ---------|
#            \-D
#            /-B
# ---------|
#           |          /-C
#            \--------|
#                      \-D
#            /-A
# ---------|
#           |          /-B
#            \--------|
#                      |          /-C
#                       \--------|
#                                 \-D
```

Additionally, you can implement your own traversing function using the structural attributes of nodes. In the following example, only nodes between a given leaf and the tree root are visited.

Example 7

```
from ete2 import Tree
tree = Tree( '(A:1,(B:1,(C:1,D:1):0.5):0.5);' )
# Browse the tree from a specific leaf to the root
node = t.search_nodes(name="C")[0]
while node:
    print node
    node = node.up
# --C
#            /-C
# ---------|
#           \-D
#
#           /-B
# ---------|
#          |         /-C
#           \--------|
#                     \-D
#
#           /-A
# ---------|
#          |         /-B
#           \--------|
#                    |         /-C
#                     \--------|
#                               \-D
```

1.4.3 Finding Nodes by Their Attributes

Both terminal and internal nodes can be located by searching along the tree structure. You can find, for instance, all nodes matching a given name. The **search_nodes()** method is the most direct way to find specific nodes. Given that every node has its own **search_nodes** method, you can start your search from different points of the tree. Any node's attribute can be used as a filter to find nodes.

Example 8

```
from ete2 import Tree
#Loads a tree
t = Tree( '((H:1,I:1):0.5, A:1, (B:1,(C:1,D:1):0.5):0.5);' )
print t
#                    /-H
#          /--------|
#         |          \-I
#         |
#---------|--A
#         |
#         |          /-B
#          \--------|
#                   |          /-C
#                    \--------|
#                              \-D
# I get D
D = t.search_nodes(name="D")
# I get all nodes with distance=0.5
nodes = t.search_nodes(dist=0.5)
print len(nodes), "nodes have distance=0.5"
```

A limitation of this method is that you cannot use complex conditional statements to find specific nodes. However you can user traversing methods to meet your custom filters. A possible general strategy would look like this:

Example 9

```
from ete2 import Tree
t = Tree( '((H:0.3,I:0.1):0.5, A:1, (B:0.4,(C:1,D:1):0.5):0.5);' )
# Create a small function to filter your nodes
def conditional_function(node):
    if node.dist > 0.3:
return True
    else:
return False
# Use previous function to find matches. Note that we use the traverse
# method in the filter function. This will iterate over all nodes to
# assess if they meet our custom conditions and will return a list of
# matches.
matches = filter(conditional_function, t.traverse())
print len(matches), "nodes have ditance >0.3"
# depending on the complexity of your conditions you can do the same
# in just one line with the help of lambda functions:
matches = filter(lambda n: n.dist>0.3 and n.is_leaf(), t.traverse() )
print len(matches), "nodes have ditance >0.3 and are leaves"
```

Finally, ETE implements a built-in method to find the **first node matching a given name**, which is one of the most common tasks needed for tree analysis. This can be done through the operator **&** (AND). Thus, **MyTree&"A"** will always return the first node whose name is "A" and that is under the tree "MyTree". The syntaxis may seem confusing, but it can be very useful in some situations.

Example 10

```
from ete2 import Tree
t = Tree( '((H:0.3,I:0.1):0.5, A:1, (B:0.4,(C:1,(J:1, (F:1, D:1):0.5):0.5):0.5):0.5);' )
# Get the node D in a very simple way
D = t&"D"
# Get the path from B to the root
node = D
path = []
while node.up:
    path.append(node)
    node = node.up
print t
# I substract D node from the total number of visited nodes
print "There are", len(path)-1, "nodes between D and the root"
# Using parentheses you can use by-operand search syntax as a node
# instance itself
Dsparent= (t&"C").up
Bsparent= (t&"B").up
Jsparent= (t&"J").up
# I check if nodes belong to certain partitions
print "It is", Dsparent in Bsparent, "that C's parent is under B's ancestor"
print "It is", Dsparent in Jsparent, "that C's parent is under J's ancestor"
```

1.4.4 Iterating instead of Getting

Methods starting with **"get_"** are all prepared to return results as a closed list of items. This means, for instance, that if you want to process all tree leaves and you ask for them using the **get_leaves()** method, the whole tree structure will be browsed before returning the final list of terminal nodes. This is not a problem in most of the cases, but in large trees, you can speed up the browsing process by using iterators.

Most **"get_"-like** methods have their homologous iterator function. Thus, **get_leaves()** can sometimes be substituted by **iter_leaves()**. The same occurs with **iter_descendants()** and **iter_search_nodes().**

When iterators are used (note that is only applicable for looping), only one step is processed at a time. For example, **iter_search_nodes()** will return one match in each iteration. In practice, this makes no differences in the final result, but it may increase the performance of loop functions (i.e. in case of finding a match which interrupts the loop).

Example 11

```
import time
from ete2 import Tree
# Creates a random tree with 10,000 leaf nodes
tree = Tree()
tree.populate(10000)
# This code should be faster
t1 = time.time()
for leaf in tree.iter_leaves():
    if "aw" in leaf.name:
        print "found a match:", leaf.name,
        break
print "Iterating: ellapsed time:", time.time()-t1
# This slower
t1 = time.time()
for leaf in tree.get_leaves():
    if "aw" in leaf.name:
        print "found a match:", leaf.name,
        break
print "Getting: ellapsed time:", time.time()-t1
# Results in something like:
# found a match: guoaw Iterating: ellapsed time: 0.00436091423035 secs
# found a match: guoaw Getting: ellapsed time: 0.124316930771 secs
```

1.5 Extending Node's Features

Although newick standard was only thought to contain branch lengths and node names information, the truth is that many other features are usually required to be linked to the different tree nodes. ETE allows to associated any kind of extra information to the tree nodes. Extra information can be regarded as a single numeric value, a text label or even as a reference to a more complex python structure (i.e. lists, dictionaries or any other python object). Thus, for example, with ETE it is possible to have fully annotated trees. The methods **add feature()**, **add_features()** and **del_feature()** are prepared to handle the task of adding and deleting information to a given node.

Once extra features are added, you can access their values at any time during the analysis of a tree. To do so, you only need to access to the **node.featurename** attributes. Let's see this with some examples:

Example 12

```
import random
from ete2 import Tree
# Creates a normal tree
t = Tree( '((H:0.3,I:0.1):0.5, A:1, (B:0.4,(C:0.5,(J:1.3, (F:1.2, D:0.1):0.5):0.5):0.5):0.5);' )
print t
# Let's locate some nodes using the get common ancestor method
ancestor=t.get_common_ancestor("J", "F", "C")
# the search_nodes method (I take only the first match )
A = t.search_nodes(name="A")[0]
# and using the shorcut to finding nodes by name
C= t&"C"
```

```
H= t&"H"
I= t&"I"
# Let's now add some custom features to our nodes. add_features can be
#  used to add many features at the same time.
C.add_features(vowel=False, confidence=1.0)
A.add_features(vowel=True, confidence=0.5)
ancestor.add_features(nodetype="internal")
# Or, using the oneliner notation
(t&"H").add_features(vowel=False, confidence=0.2)
# But we can automatize this. (note that i will overwrite the previous
# values)
for leaf in t.traverse():
    if leaf.name in "AEIOU":
 leaf.add_features(vowel=True, confidence=random.random())
    else:
 leaf.add_features(vowel=False, confidence=random.random())
# Now we use these information to analyze the tree.
print "This tree has", len(t.search_nodes(vowel=True)), "vowel nodes"
print "Which are", [leaf.name for leaf in t.iter_leaves() if leaf.vowel==True]
# But features may refer to any kind of data, not only simple
# values. For example, we can calculate some values and store them
# within nodes.
#
# Let's detect leaf nodes under "ancestor" with distance higher thatn
# 1. Note that I'm traversing a subtree which starts from "ancestor"
matches = [leaf for leaf in ancestor.traverse() if leaf.dist>1.0]
# And save this pre-computed information into the ancestor node
ancestor.add_feature("long_branch_nodes", matches)
# Prints the precomputed nodes
print "These are nodes under ancestor with long branches", \
    [n.name for n in ancestor.long_branch_nodes]
# We can also use the add_feature() method to dynamically add new features.
label = raw_input("custom label:")
value = raw_input("custom label value:")
ancestor.add_feature(label, value)
print "Ancestor has now the [", label, "] attribute with value [", value, "]"
```

Unfortunately, newick format does not support adding extra features to a tree. Because of this drawback, several improved formats haven been (or are being) developed to read and write tree based information. Some of these new formats are based in a completely new standard (PhyloXML, NeXML), while others are extensions of the original newick formar (NHX http://phylosoft.org/NHX/). Currently, ETE includes support for the New Hampshire eXtended format (NHX), which uses the original newick standard and adds the possibility of saving additional date related to each tree node. Here is an example of a extended newick representation in which extra information is added to an internal node:

```
(A:0.35,(B:0.72,(D:0.60,G:0.12):0.64[&&NHX:conf=0.01:name=INTERNAL]):0.56);
```

As you can notice, extra node features in the NHX format are enclosed between brackets. ETE is able to read and write features using such format, however, the encoded information is expected to be text-formattable. In the future, support for more advanced formats such as PhyloXML will be included.

The NHX format is automatically detected when reading a newick file, and the detected node features are added using the "**add_feature()**" method. Consequently, you can access the information by using the normal ETE's feature notation: **node.featurename**. Similarly, features added to a tree can be included within the normal newick representation using the NHX notation. For this, you can call the **write()** method using the **features** argument, which is expected to be a list with the features names that you want to include in the newick string. Note that all nodes containing the suplied features will be exposed into the newick string. Use an empty features list (**features=[ ]**) to include all node's data into the newick string.

Example 13

```
import random
from ete2 import Tree
# Creates a normal tree
t = Tree('((H:0.3,I:0.1):0.5, A:1,(B:0.4,(C:0.5,(J:1.3,(F:1.2, D:0.1):0.5):0.5):0.5):0.5);')
print t
# Let's locate some nodes using the get common ancestor method
ancestor=t.get_common_ancestor("J", "F", "C")
# Let's label  leaf nodes
for leaf in t.traverse():
    if leaf.name in "AEIOU":
 leaf.add_features(vowel=True, confidence=random.random())
    else:
 leaf.add_features(vowel=False, confidence=random.random())
# Let's detect leaf nodes under "ancestor" with distance higher thatn
# 1. Note that I'm traversing a subtree which starts from "ancestor"
matches = [leaf for leaf in ancestor.traverse() if leaf.dist>1.0]
# And save this pre-computed information into the ancestor node
ancestor.add_feature("long_branch_nodes", matches)
print
print "NHX notation including vowel and confidence attributes"
print
print t.write(features=["vowel", "confidence"])
print
print "NHX notation including all node's data"
print
# Note that when all features are requested, only those with values
# equal to text-strings or numbers are considered. "long_branch_nodes"
# is not included into the newick string.
print t.write(features=[])
print
print "basic newick formats are still available"
print
print t.write(format=9, features=["vowel"])
# You don't need to do anything speciall to read NHX notation. Just
# specify the newick format and the NHX tags will be automatically
# detected.
nw = """
(((ADH2:0.1[&&NHX:S=human:E=1.1.1.1], ADH1:0.11[&&NHX:S=human:E=1.1.1.1])
:0.05[&&NHX:S=Primates:E=1.1.1.1:D=Y:B=100], ADHY:0.1[&&NHX:S=nematode:
E=1.1.1.1],ADHX:0.12[&&NHX:S=insect:E=1.1.1.1]):0.1[&&NHX:S=Metazoa:
E=1.1.1.1:D=N], (ADH4:0.09[&&NHX:S=yeast:E=1.1.1.1],ADH3:0.13[&&NHX:S=yeast:
E=1.1.1.1], ADH2:0.12[&&NHX:S=yeast:E=1.1.1.1],ADH1:0.11[&&NHX:S=yeast:E=1.1.1.1]):0.1
 [&&NHX:S=Fungi])[&&NHX:E=1.1.1.1:D=N];"""
# Loads the NHX example found at http://www.phylosoft.org/NHX/
t = Tree(nw)
# And access node's attributes.
for n in t.traverse():
    if hasattr(n,"S"):
 print n.name, n.S
```

1.6 Modifying Tree Topology

1.6.1 Creating Trees from Scratch

If no arguments are passed to the **Tree** class constructor, an empty tree node will be returned. Then, you can use such an orphan node to populate a tree from scratch. For this, you should never manipulate the **up**, and **children** attributes of a node (unless it is strictly necessary). Instead, you must use the methods created to this end. **add_child()**, **add_sister()**, and **populate()** are the most common methods to create a tree structure. While the two first adds one node at a time, populate() is able to create a custom number of random nodes. This is useful to quickly create random trees.

Example 14

```
from ete2 import Tree
t = Tree() # Creates an empty tree
A = t.add_child(name="A") # Adds a new child to the current tree root
      # and returns it
B = t.add_child(name="B") # Adds a second child to the current tree
      # root and returns it
C = A.add_child(name="C") # Adds a new child to one of the branches
D = C.add_sister(name="D") # Adds a second child to same branch as
        # before, but using a sister as the starting
        # point
R = A.add_child(name="R") # Adds a third child to the
      # branch. Multifurcations are supported
# Next, I add 6 random leaves to the R branch names_library is an
# optional argument. If no names are provided, they will be generated
# randomly.
R.populate(6, names_library=["r1","r2","r3","r4","r5","r6"])
# Prints the tree topology
print t
#                     /-C
#                    |
#                    |--D
#                    |
#           /--------|                              /-r4
#          |         |                    /--------|
#          |         |          /--------|          \-r3
#          |         |         |         |
#          |         |         |          \-r5
#          |          \--------|
# ---------|                   |                    /-r6
#          |                   |          /--------|
#          |                    \--------|          \-r2
#          |                             |
#          |                              \-r1
#          |
#           \-B
# a common use of the populate method is to quickly create example
# trees from scratch. Here we create a random tree with 100 leaves.
t = Tree()
t.populate(100)
```

1.6.2 Deleting (eliminating) and Removing (detaching) nodes

As currently implemented, there is a difference between removing or deleting a node. The former (removing) detaches a node's partition from the tree structure, so all its descendants are also disconnected from the tree. There are two methods to perform this action: **node.remove_child(ch)** and **child.detach()**. In contrast, deleting a node means eliminating such node without affecting its descendants. Children from the deleted node are automatically connected to the next possible parent. This is better understood with the following example:

Example 15

```
from ete2 import Tree
# Loads a tree. Note that we use format 1 to read internal node names
t = Tree('((((H,K)D,(F,I)G)B,E)A,((L,(N,Q)O)J,(P,S)M)C);', format=1)
print "original tree looks like this:"
# This is an alternative way of using "print t". Thus we have a bit
# more of control on how tree is printed. Here i print the tree
# showing internal node names
print t.get_ascii(show_internal=True)
#
#                                        /-H
#                              /D-------|
#                             |          \-K
#                    /B-------|
#                   |         |          /-F
#          /A-------|          \G-------|
#         |         |                    \-I
#         |         |
#         |          \-E
#-NoName--|
#         |                    /-L
#         |          /J-------|
#         |         |         |          /-N
#         |         |          \O-------|
#          \C-------|                    \-Q
#                   |
#                   |          /-P
#                    \M-------|
#                              \-S
# Get pointers to specific nodes
G = t.search_nodes(name="G")[0]
J = t.search_nodes(name="J")[0]
C = t.search_nodes(name="C")[0]
# If we remove J from the tree, the whole partition under J node will
# be detached from the tree and it will be considered an independent
# tree. We can do the same thing using two approaches: J.detach() or
# C.remove_child(J)
removed_node = J.detach() # = C.remove_child(J)
# if we know print the original tree, we will see how J partition is
# no longer there.
print "Tree after REMOVING the node J"
print t.get_ascii(show_internal=True)
#                                        /-H
#                              /D-------|
#                             |          \-K
#                    /B-------|
#                   |         |          /-F
#          /A-------|          \G-------|
#         |         |                    \-I
#         |         |
#-NoName--|          \-E
#         |
#         |                    /-P
#          \C------- /M-------|
#                              \-S
# however, if we DELETE the node G, only G will be eliminated from the
# tree, and all its descendants will then hang from the next upper
# node.
G.delete()
```

```
print "Tree after DELETING the node G"
print t.get_ascii(show_internal=True)
#                                          /-H
#                               /D-------|
#                              |          \-K
#                     /B-------|
#                    |         |--F
#           /A-------|         |
#          |         |          \-I
#          |         |
#-NoName--|          \-E
#          |
#          |                    /-P
#           \C------- /M-------|
#                               \-S
```

1.7 Pruning trees

Pruning a tree means to obtain the topology that connects a certain group of items by removing the unnecessary edges. To facilitate this task, ETE implements the **prune()** method, which can be used in two different ways: by providing the list of terminal nodes that must be kept in the tree; or by providing a list of nodes that must be removed. In any case, the result is a pruned tree containing the topology that connects a custom set of nodes.

Example 16

```
from ete2 import Tree
# Let's create simple tree
t = Tree('((((H,K),(F,I)G),E),((L,(N,Q)O),(P,S)));')
print "Original tree looks like this:"
print t
#
#                                          /-H
#                               /--------|
#                              |          \-K
#                     /--------|
#                    |         |          /-F
#           /--------|          \--------|
#          |         |                    \-I
#          |         |
#          |          \-E
#---------|
#          |                    /-L
#          |          /--------|
#          |         |         |          /-N
#          |         |          \--------|
#           \--------|                    \-Q
#                    |
#                    |          /-P
#                     \--------|
#                               \-S
# Prune the tree in order to keep only some leaf nodes.
t.prune(["H","F","E","Q", "P"])
print "Pruned tree method=keep"
print t
#
#                               /-F
#                     /--------|
#           /--------|          \-H
#          |         |
#---------|          \-E
#         |
#         |          /-Q
#          \--------|
#                    \-P
# Let's re-create the same tree again
t = Tree('((((H,K),(F,I)G),E),((L,(N,Q)O),(P,S)));')
print "Pruned tree method=crop"
t.prune(["H","F","E","Q", "P"], method="crop")
print t
#
#                              /-L
#                    /--------|
#          /--------|          \-N
#         |         |
#---------|          \-S
#         |
#         |          /-K
#          \--------|
#                    \-I
```

1.8 Concatenating trees

Given that all tree nodes share the same basic properties, they can be connected freely. In fact, any node can add a whole subtree as a child, so we can actually *cut & paste* partitions. To do so, you only need to call the **add_child()** method using another tree node as a first argument. If such a node is the root node of a different tree, you will concatenate two structures. But caution!!, this kind of operations may result into circular tree structures if add an node's ancestor as a new node's child. Some basic checks are internally performed by the ETE topology related methods, however, a fully qualified check of this issue would affect seriously to the performance of the program. For this reason, users should take care about not creating circular structures by mistake.

Example 17

```
from ete2 import Tree
# Loads 3 independent trees
t1 = Tree('(A,(B,C));')
t2 = Tree('((D,E), (F,G));')
t3 = Tree('(H, ((I,J), (K,L)));')
print "Tree1:", t1
#            /-A
#   ---------|
#            |          /-B
#             \--------|
#                       \-C
print "Tree2:", t2
#                       /-D
#             /--------|
#            |          \-E
#   ---------|
#            |          /-F
#             \--------|
#                       \-G
print "Tree3:", t3
#            /-H
#            |
#   ---------|                    /-I
#            |          /--------|
#           |         |          \-J
#            \--------|
#                     |          /-K
#                      \--------|
#                                \-L
# Locates a terminal node in the first tree
A = t1.search_nodes(name='A')[0]
# and adds the two other trees as children.
A.add_child(t2)
A.add_child(t3)
print "Resulting concatenated tree:", t1
#                                          /-D
#                                /--------|
#                               |          \-E
#                      /--------|
#                     |         |          /-F
#                     |          \--------|
#            /--------|                    \-G
#           |         |
#           |         |          /-H
#           |         |         |
#           |          \--------|                    /-I
#           |                   |          /--------|
#  ---------|                   |         |          \-J
#           |                    \--------|
#           |                             |          /-K
#           |                              \--------|
#           |                                        \-L
#           |
#           |          /-B
#            \--------|
#                      \-C
# But remember!!!You should never do things like:
#
# A.add_child(t1)
#
```

1.9 Tree Rooting

Tree rooting is understood as the technique by with a given tree is conceptually polarized from more basal to more terminal nodes. In phylogenetics, for instance, this a crucial step prior to the interpretation of trees, since it will determine the evolutionary relationships among the species involved. The concept of rooted trees is different than just having a root node, which is always necessary to handle a tree data structure. Usually, the way in which a tree is differentiated between rooted and unrooted, is by counting the number of branches of the current root node. Thus, if the root node has more than two child branches, the tree is considered unrooted. By contrast, when only two main branches exist under the root node, the tree is considered rooted. Having an unrooted tree means that any internal branch within the tree could be regarded as the root node, and there is no conceptual reason to place the root node where it is placed at the moment. Therefore, in an unrooted tree, there is no information about which internal nodes are more basal than others. By setting the root node between a given edge/branch of the tree structure the tree is polarized, meaning that the two branches under the root node are the most basal nodes. In practice, this is usually done by setting an **outgroup node**, which would represent one of these main root branches. The second one will be, obviously, the brother node. When you set an outgroup on unrooted trees, the multifurcations at the current root node are solved.

In order to root an unrooted tree or re-root a tree structure, ETE implements the **set_outgroup()** method, which is present in any tree node instance. Similarly, the **unroot()** method can be used to perform the opposite action.

Example 18

```
from ete2 import Tree
# Load an unrooted tree. Note that three branches hang from the root
# node. This usually means that no information is available about
# which of nodes is more basal.
t = Tree('(A,(H,F)(B,(E,D)));')
print "Unrooted tree"
print t
#          /-A
#         |
#         |          /-H
#---------|---------|
#         |          \-F
#         |
#         |          /-B
#          \--------|
#                   |          /-E
#                    \--------|
#                              \-D
#
# Let's define that the ancestor of E and D as the tree outgroup.  Of
# course, the definition of an outgroup will depend on user criteria.
ancestor = t.get_common_ancestor("E","D")
t.set_outgroup(ancestor)
print "Tree rooteda at E and D's ancestor is more basal that the others."
print t
#
#                    /-B
#          /--------|
#         |         |          /-A
#         |          \--------|
#         |                   |          /-H
#---------|                    \--------|
#         |                              \-F
#         |
#         |          /-E
#          \--------|
#                    \-D
#
# Note that setting a different outgroup, a different interpretation
# of the tree is possible
t.set_outgroup( t&"A" )
print "Tree rooted at a terminal node"
print t
#                              /-H
#                    /--------|
#                   |          \-F
#          /--------|
#         |         |          /-B
#         |          \--------|
#---------|                   |          /-E
#         |                    \--------|
#         |                              \-D
#         |
#          \-A
```

Note that although **rooting** is usually regarded as a whole-tree operation, ETE allows to root subparts of the tree without affecting to its parent tree structure.

Example 19

```
from ete2 import Tree
t = Tree('(((A,C),((H,F),(L,M))),((B,(J,K))(E,D)));')
print "Original tree:"
print t
#                              /-A
#                    /--------|
#                   |          \-C
#                   |
#          /--------|                    /-H
#         |         |          /--------|
#         |         |         |          \-F
#         |          \--------|
#         |                   |          /-L
#         |                    \--------|
#---------|                              \-M
#         |
#         |                    /-B
#         |          /--------|
#         |         |         |          /-J
#         |         |          \--------|
#          \--------|                    \-K
#                   |
#                   |          /-E
#                    \--------|
#                              \-D
#
# Each main branch of the tree is independently rooted.
node1 = t.get_common_ancestor("A","H")
node2 = t.get_common_ancestor("B","D")
node1.set_outgroup("H")
node2.set_outgroup("E")
print "Tree after rooting each node independently:"
print t
#
#                              /-F
#                             |
#                    /--------|                    /-L
#                   |         |          /--------|
#                   |         |         |          \-M
#                   |          \--------|
#          /--------|                   |          /-A
#         |         |                    \--------|
#         |         |                              \-C
#         |         |
#         |          \-H
#---------|
#         |                    /-D
#         |          /--------|
#         |         |         |          /-B
#         |         |          \--------|
#          \--------|                   |          /-J
#                   |                    \--------|
#                   |                              \-K
#                   |
#                    \-E
```

1.10 Working with branch distances

The branch length between one node an its parent is encoded as the **node.dist** attribute. Together with tree topology, branch lengths define the relationships among nodes.

1.10.1 Getting distances between nodes

The **get_distance()** method can be used to calculate the distance between two connected nodes. There are two ways of using this method: a) by querying the distance between two descendant nodes (two nodes are passed as arguments) b) by querying the distance between the current node and any other relative node (parental or descendant).

Additionally to this, ETE incorporates two more methods to calculate the most distant node from a given point in a tree. You can use the **get_farthest_node()** method to retrieve the most distant point from a node within the whole tree structure. Alternatively, **get_farthest_leaf()** will return the most distant descendant (always a leaf). If more than one node matches the farthest distance, the first occurrence is returned.

Distance between nodes can also be computed as the number of nodes between them (considering all branch lengths equal to 1.0). To do so, the **topology_only** argument must be set to **True** for all the above mentioned methods.

Example 20

```
from ete2 import Tree

# Loads a tree with branch lenght information. Note that if no
# distance info is provided in the newick, it will be initialized with
# the default dist value = 1.0
nw = """(((A:0.1, B:0.01):0.001, C:0.0001):1.0,
(((((D:0.00001:0,I:0):0,F:0):0,G:0):0,H:0):0,
E:0.000001):0.0000001):2.0;"""
t = Tree(nw)
print t
#                               /-A
#                     /--------|
#           /--------|          \-B
#          |         |
#          |          \-C
#          |
#          |                                                  /-D
#          |                                        /--------|
#----------|                              /--------|          \-I
#          |                             |         |
#          |                   /--------|          \-F
#          |                  |         |
#          |         /--------|          \-G
#          |        |         |
#           \--------|          \-H
#                    |
#                     \-E
#
# Locate some nodes
A = t&"A"
C = t&"C"
# Calculate distance from current node
print "The distance between A and C is",  A.get_distance("C")
# Calculate distance between two descendants of current node
print "The distance between A and C is",  t.get_distance("A","C")
# Calculate the toplogical distance (number of nodes in between)
```

```
print "The number of nodes between A and D is ",  \
    t.get_distance("A","D", topology_only=True)
# Calculate the farthest node from E within the whole structure
farthest, dist = (t&"E").get_farthest_node()
print "The farthest node from E is", farthest.name, "with dist=", dist
# Calculate the farthest node from E within the whole structure,
# regarding the number of nodes in between as distance value
# Note that the result is diffrent.
farthest, dist = (t&"E").get_farthest_node(topology_only=True)
print "The farthest (topologically) node from E is", \
    farthest.name, "with", dist, "nodes in between"
# Calculate farthest node from an internal node
farthest, dist = t.get_farthest_node()
print "The farthest node from root is is", farthest.name, "with dist=", dist
#
# The program results in the following information:
#
# The distance between A and C is 0.1011
# The distance between A and C is 0.1011
# The number of nodes between A and D is  8.0
# The farthest node from E is A with dist= 1.1010011
# The farthest (topologically) node from E is I with 5.0 nodes in between
# The farthest node from root is is A with dist= 1.101
```

1.10.2 getting midpoint outgroup

In order to obtain a balanced rooting of the tree, you can set as the tree outgroup that partition which splits the tree in two equally distant clusters (using branch lengths). This is called the midpoint outgroup.

The **get_midpoint_outgroup()** method will return the outgroup partition that splits current node into two balanced branches in terms of node distances.

Example 21

```
from ete2 import Tree
# generates a random tree
t = Tree();
t.populate(15);
print t
#
#
#                    /-qogjl
#          /--------|
#         |          \-vxbgp
#         |
#         |          /-xyewk
#---------|         |
#         |         |                   /-opben
#         |         |                  |
#         |         |         /--------|                    /-xoryn
#          \--------|        |         |          /--------|
#                   |        |         |         |         |          /-wdima
#                   |        |          \--------|          \--------|
#                   |        |                   |                    \-qxovz
#                   |        |                   |
#                   |        |                    \-isngq
#                    \--------|
#                             |                   /-neqsc
#                             |                  |
#                             |                  |                              /-waxkv
#                             |         /--------|                    /--------|
#                             |         |         |          /--------|          \-djeoh
#                             |         |         |         |         |
#                             |         |          \--------|          \-exmsn
#                              \--------|                   |
#                                       |                   |          /-udspq
#                                       |                    \--------|
#                                       |                              \-buxpw
#                                       |
#                                        \-rkzwd
# Calculate the midpoint node
R = t.get_midpoint_outgroup()
# and set it as tree outgroup
t.set_outgroup(R)
print t
#                              /-opben
#                             |
#                    /--------|                    /-xoryn
#                   |         |          /--------|
#                   |         |         |         |          /-wdima
#                   |          \--------|          \--------|
#          /--------|                   |                    \-qxovz
#         |         |                   |
#         |         |                    \-isngq
#         |         |
#         |         |          /-xyewk
#         |          \--------|
#         |                   |          /-qogjl
#         |                    \--------|
#---------|                              \-vxbgp
#         |
#         |                    /-neqsc
#         |                   |
#         |                   |                              /-waxkv
#         |          /--------|                    /--------|
#         |         |         |          /--------|          \-djeoh
#         |         |         |         |         |
#         |         |          \--------|          \-exmsn
#          \--------|                   |
#                   |                   |          /-udspq
#                   |                    \--------|
#                   |                              \-buxpw
#                   |
#                    \-rkzwd
#
```

CHAPTER 2

THE PROGRAMMABLE TREE DRAWING ENGINE

ETE's treeview extension provides a highly programmable drawing system to render any hierarchical tree structure into a custom image. Although a number or predefined visualization layouts are included with the default installation, custom styles can be easily created from scratch. To do so, ETE makes use of there main concepts (node **styles**, node **faces** and **layout** functions), which allow the user to define the rules in which trees are rendered. Briefly, a node **style** defines the general aspect of a given tree node (size, color, background, line type, etc.). Node **faces** are small graphical pieces (representing, for instance, any node's extra information) that are added to nodes and that drawn at the same node position. Finally, **layouts functions** are custom python functions that define the rules on how faces and styles are added to nodes when they are going to be drawn. By combining this elements, the aspect of trees can be controlled by custom criteria.

Treeview extension provides an interactive Graphical User Interface (GUI) to visualize trees using custom layouts. Alternatively, images can be directly rendered as PNG or PDF files. Every node within a given tree structure has its own **show()** and **render()** methods, thus allowing to visualize or render its subtree structure.

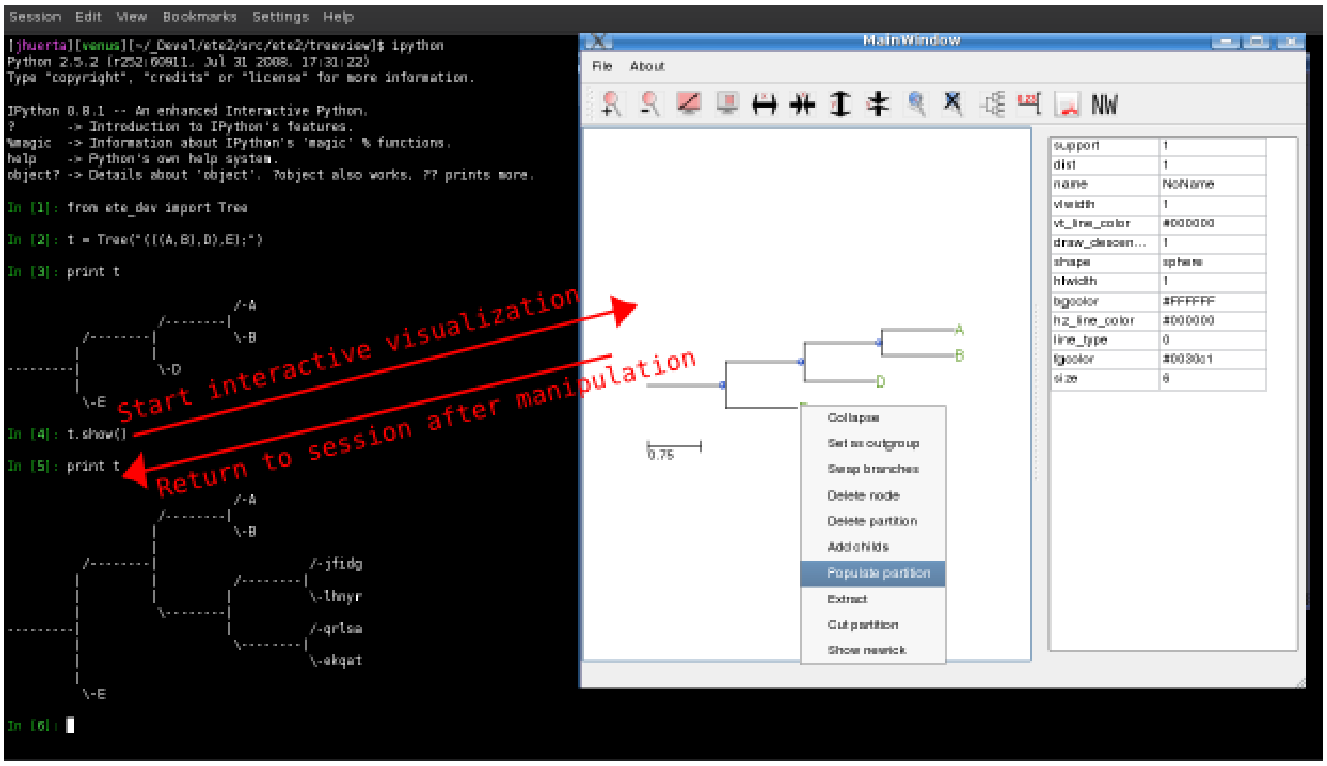
2.1 Interactive visualization of trees

ETE's tree drawing engine is fully integrated with a built-in graphical user interface (GUI). Thus, ETE allows to render tree structures directly on an interactive interface that can be used to explore and manipulate trees node's properties and topology. The GUI is based on Qt4 , a cross platform and open source application and UI framework which allows to handle, virtually, images of any size. Of course, this will depend on you computer and graphical card performance.

To start the visualization of a given tree or subtree, you can simply call the **show()** method present in every node:

```
from ete2 import Tree
t = Tree(" ((A,B),C); ")
t.show()
ancestor = t.get_common_ancestor("A", "B")
ancestor.show()
```

One of the advantages of this on-line GUI visualization is that you can use it to interrupt a given program/analysis, explore trees, manipulate them, and continuing with the execution thread. Note that **changes made using the GUI will be kept in the tree structure after quiting the visualization interface** (Figure 2.1). This feature is specially useful for using during python sessions, in which analyses are performed interactively.

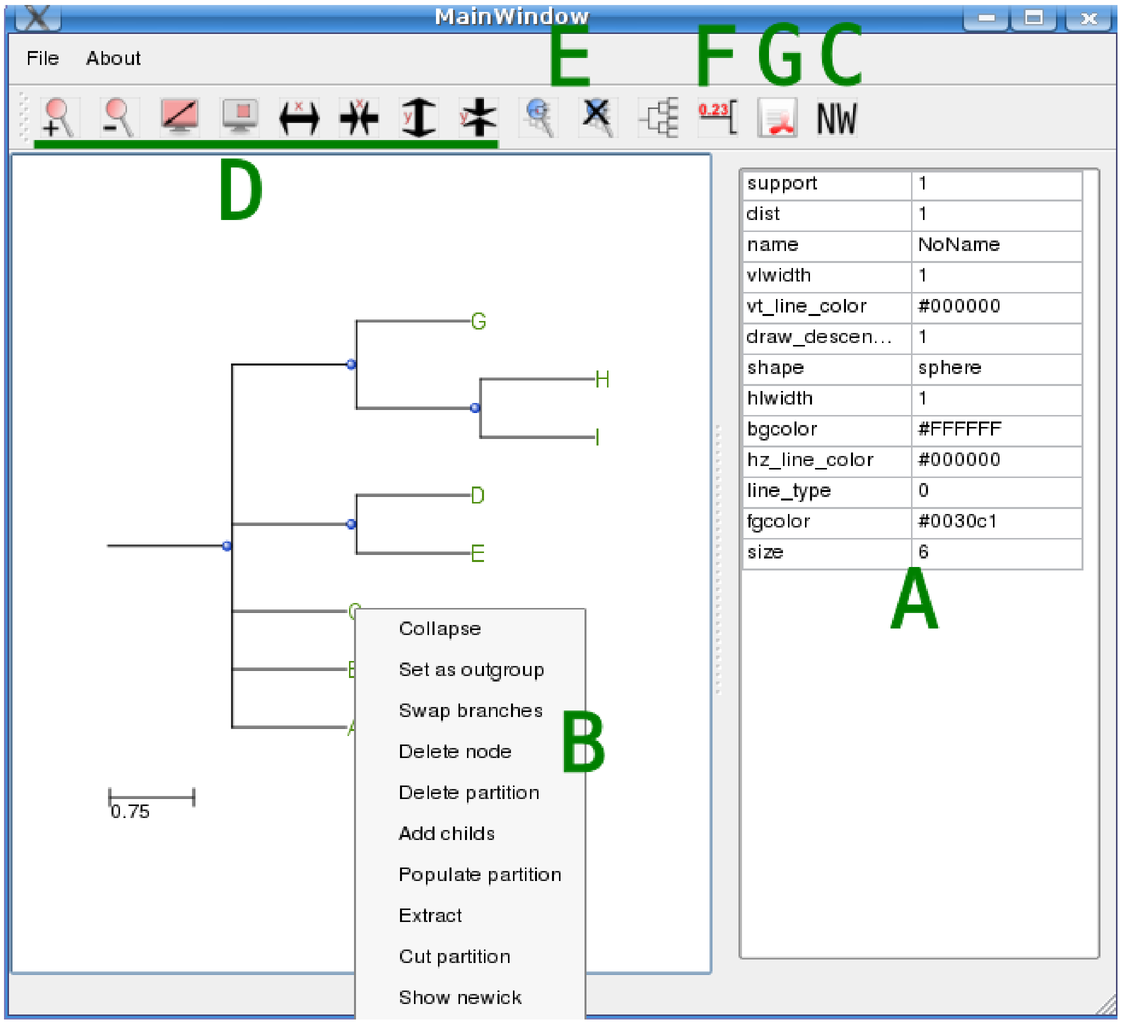


```
[jhuerta][venus][~/_Devel/ete2/src/ete2/treeview]$ ipython
Python 2.5.2 (r252:60911, Jul 31 2008, 17:31:22)
Type "copyright", "credits" or "license" for more information.

IPython 0.8.1 -- An enhanced Interactive Python.
?       -> Introduction to IPython's features.
%magic  -> Information about IPython's 'magic' % functions.
help    -> Python's own help system.
object? -> Details about 'object'. ?object also works, ?? prints more.

In [1]: from ete_dev import Tree

In [2]: t = Tree("(((A,B),D),E);")

In [3]: print t

                              /-A
                    /--------|
          /--------|          \-B
         |         |
---------|          \-D
         |
          \-E

In [4]: t.show()

In [5]: print t

                              /-A
                    /--------|
                   |          \-B
                   |
          /--------|                    /-jfidg
         |         |          /--------|
         |         |         |          \-lhnyr
         |          \--------|
---------|                   |          /-qrlsa
         |                    \--------|
         |                              \-ekqat
         |
          \-E

In [6]: █
```

support	1
dist	1
name	NoName
vlwidth	1
vt_line_color	#000000
draw_descen...	1
shape	sphere
hlwidth	1
bgcolor	#FFFFFF
hz_line_color	#000000
line_type	0
fgcolor	#0030c1
size	6

Figure 2.1: Interactive manipulation of a tree using the GUI

The GUI allow many operations to be performed graphically, however it does not implement all the possibilities of the programming toolkit. These are some of the allowed GUI options:



support	1
dist	1
name	NoName
vlwidth	1
vt_line_color	#000000
draw_descen...	1
shape	sphere
hlwidth	1
bgcolor	#FFFFFF
hz_line_color	#000000
line_type	0
fgcolor	#0030c1
size	6

Figure 2.2: ETE's GUI screenshot showing some of its options. A) Look up and change node's features. **B)** Manipulate tree topology (add/remove nodes, populate partitions, cut and paste nodes). Browse tree topology (expand and collapse partitions). **C)** Extract newick representation of any sub part of the tree. **D)** Zoom in/out of node tree regions. **E)** Advanced search dialog. Allows to find nodes meeting custom criteria. **F)** Show branch lengths, support values and force strict tree topology**. G)** Export image as PDF.

2.2 Render trees into image files

Alternatively, images can be directly written info a file. PNG and PDF formats are supported. While PNG will store only a normal image, PDF will keep vector graphics format, thus allowing to better edit or resize images using the proper programs (Such as inkscape in GNU/linux).

To generate an image, the **render()** method should be used instead of **show()**. The only required argument is the file name, which will determine the final format of the file (.pdf or .png). By default, the resulting image is scaled to 7 inches, approximately the width of an A4 paper. However, you can change this by setting a custom width and height. If only one of this values is provided, the other is imputed to keep the original aspect ratio.

Example 22

```
from ete2 import Tree
t = Tree()
# Generate a random tree with 50 leaves
t.populate(50)
# Render  tree in png and pdf format using the default size
t.render("./random_tree.png")
t.render("./random_tree.pdf")
# Render tree in pdf setting a custom width. height will be imputed
t.render("./random_tree.pdf", w=300)
# Render tree in pdf setting a custom height. Width will be imputed
t.render("./random_tree.pdf", h=600)
# Render tree in pdf setting a custom width and height
t.render("./random_tree.pdf", w=300, h=300)
```

2.3 Customizing tree aspect

There are three basic elements that control the general aspect of trees: **node's style**, **node's faces** and **layouts functions**. In brief, layout functions can be used to set the rules that control the way in which certain nodes are drawn (setting its style and adding specific faces).

2.3.1 styles

A '**style**' is a set of special node attributes that are used by the drawing algorithm to set the colours, and general aspect of nodes and branches. Styles are internally encoded as python dictionaries. Each node has its own style dictionary, which is accessible as **node.img_style**. A default style is associated to every tree node, but you can modify them at any time. Note that **nodes styles must only be modified inside a layout function**. Otherwise, custom settings may be missing or overwritten by default values.

Style attribute	Description	Valid value
node.img_style[“bgcolor”]	Color (in RGB format) used to draw the background of node's partition	i.e.: ”#FFFFFF”
node.img_style[“fgcolor”]	Color (in RGB format) used to draw the node	i.e.: ”#FFAA55”
node.img_style[“line_color”]	Color (in RGB format) used to draw the node's branch	i.e.: ”#000000”
node.img_style[“line_type”]	Line style used to draw node's brach.	0=Solid, 1=Dashed
node.img_style[“shape”]	Shape used to draw the node.	“square”, “sphere”, “circle”
node.img_style[“size”]	Node's size.	number>0 (i.e.: 10)
node.img_style[“draw_descendants”]	Flag that controls whether node's descendants must be drawn or hidden.	1=Draw, 0=Hide

Table 2.1: Supported node's style values

2.3.2 Faces

Node's faces are more sophisticated drawing features associated to nodes. They represent independent images that can be linked to nodes, usually representing a given node's feature. Faces can be loaded **from external image files**, **created from scratch** using any drawing library, **or generated as text labels**.

The complexity of faces may go from simple text tags to complete plots showing the average expression pattern associated to a given partition in a microarray clustering tree. Given that faces can be loaded from external images and added *on the fly*, any way of producing external images could be easily connected to the drawing engine. For instance, the statistical framework R could be used to analyze a given node's property, and to generate a plot that can be used as a node's face.

To create a face, the following general constructors can be used, which are **available through the face module**:

Create a face from an external image file	ImgFace("path_to_image")
Create a face using a given text string	TextFace("text", [args])
Create a text face based on a given node attribute	AttrFace("attrName", [args])

Table 2.2: Basic node's face constructors

Once a face is created, it can be linked to one or more nodes. To do so, you must use the **add_face_to_node()** method within the **faces** module. By doing this, when a node is drawn, their linked faces will be drawn beside it. Since several faces can be added to the same node, you must specify the relative position in which they will be placed. Each node reserves a virtual space that controls how faces are positioned. The position of each face is determined by an imaginary grid at the right side of each node (Figure 2.3). Each column from the grid is internally treated as a stack of faces. Thus, faces can be added to any column and its row position will be determined by insertion order: **first inserted is first row**. In the case of trees leaves, nodes can handle an independent list of faces that will be drawn aligned with the farthest leaf in the tree. To add an aligned face you can use the **aligned=True** argument when calling the **add_face_to_node()** method. By knowing this rules, you can easily fill virtual node grids with any external image or text label and the algorithm will take care of positioning. Note that **add_face_to_node()** must only be used inside a layout function.

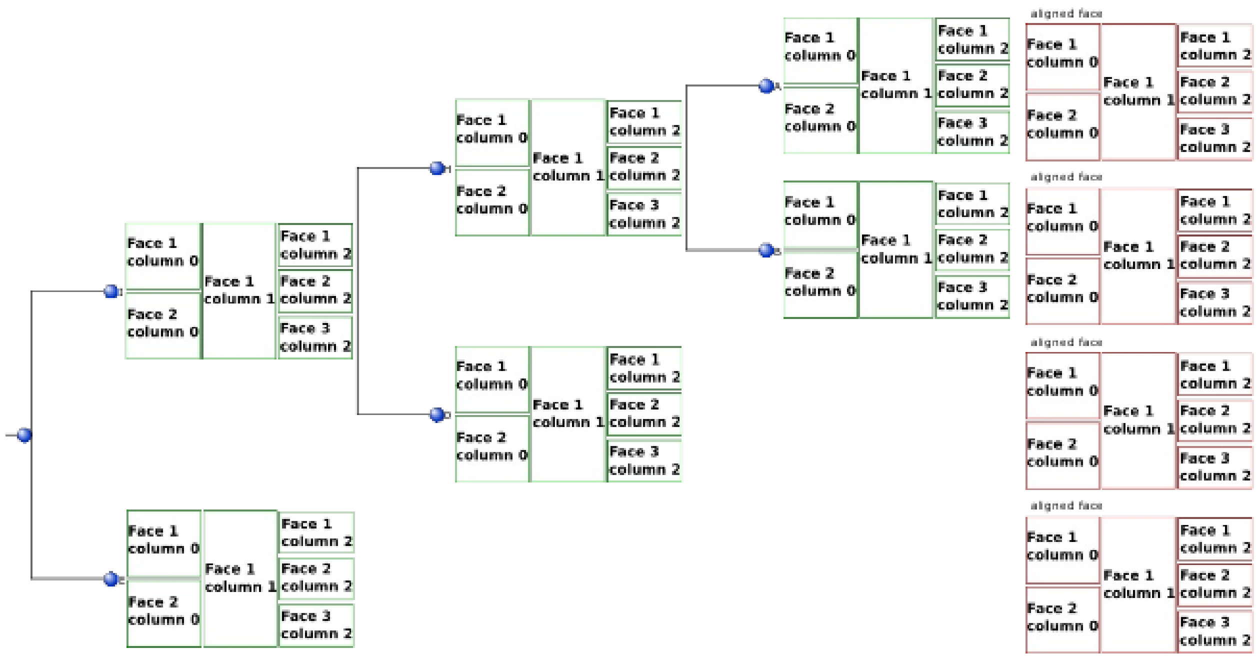


Figure 2.3: Scheme of node's faces positioning. Green boxes represent non-aligned faces. Red boxes are aligned faces (only applicable for terminal nodes)

2.3.3 layouts

Layout functions are the key component of the tree drawing customization. Any python function accepting a node instance as a first argument can be used as a layout function. Essentially, such function will be called just before drawing each tree node, so you can use it perform any operation prior to render nodes. In practice, layout functions are used to define the set of rules that control nodes style attributes and the faces that will be linked to them. Of course, such rules can be based on a previous node analysis. For instance: `if node has more than 5 descendants, then add a text label, set a different background color, perform an analysis on leaves and associate an external image` with node. As you imagine, rules can be are as sophisticated as you want. Thus, the advantage of this method is that you can create your own drawing algorithms to render trees dynamically and fitting very specific needs.

In order to apply your custom layouts functions, function's name (the reference to it) can be passed to both **render()** and **show()** methods: `node.render("filename.pdf", layout=mypython` **or** `node.show(layout=mypythonFn).`

2.3.4 Example: combining styles, faces and layouts

Example 23

```
# Import Tree instance and faces module
from ete2 import Tree, faces

# Loads an example tree
```

```
nw = """
(((Dre:0.008339,Dme:0.300613)1.000000:0.596401,
(Cfa:0.640858,Hsa:0.753230)1.000000:0.182035)1.000000:0.106234,
((Dre:0.271621,Cfa:0.046042)1.000000:0.953250,
(Hsa:0.061813,Mms:0.110769)1.000000:0.204419)1.000000:0.973467);
"""
t = Tree(nw)

# You can create any random tree containing the same leaf names, and
# layout will work equally
#
# t = Tree()
# Creates a random tree with 8 leaves using a given set of names
# t.populate(8, ["Dme", "Dre", "Hsa", "Ptr", "Cfa", "Mms"])

# Set the path in which images are located
img_path = "./"
# Create faces based on external images
humanFace = faces.ImgFace(img_path+"human.png")
mouseFace = faces.ImgFace(img_path+"mouse.png")
dogFace = faces.ImgFace(img_path+"dog.png")
chimpFace = faces.ImgFace(img_path+"chimp.png")
fishFace = faces.ImgFace(img_path+"fish.png")
flyFace = faces.ImgFace(img_path+"fly.png")

# Create a faces ready to read the name attribute of nodes
#nameFace = faces.TextFace(open("text").readline().strip(), fsize=20, fgcolor="#009000")
nameFace = faces.AttrFace("name", fsize=20, fgcolor="#009000")

# Create a conversion between leaf names and real names
code2name = {
 "Dre":"Drosophila melanogaster",
 "Dme":"Danio rerio",
 "Hsa":"Homo sapiens",
 "Ptr":"Pan troglodytes",
 "Mms":"Mus musculus",
 "Cfa":"Canis familiaris"
}

# Creates a dictionary with the descriptions of each leaf name
code2desc = {
 "Dre":"""
The zebrafish, also known as Danio rerio,
is a tropical freshwater fish belonging to the
minnow family (Cyprinidae).""",
 "Dme":"""
True flies are insects of the order Diptera,
possessing a single pair of wings on the
mesothorax and a pair of halteres, derived from
the hind wings, on the metathorax""",
 "Hsa":"""
A human is a member of a species
of bipedal primates in the family Hominidae.""",
 "Ptr":"""
Chimpanzee, sometimes colloquially
chimp, is the common name for the
two extant species of ape in the genus Pan.""",
 "Mms":"""
A mouse is a small mammal belonging to the
order of rodents.""",
 "Cfa": """
The dog (Canis lupus familiaris) is a
domesticated subspecies of the Gray Wolf,
a member of the Canidae family of the
orderCarnivora."""
}

# Creates my own layout function. I will use all previously created
# faces and will set different node styles depending on the type of
# node.
def mylayout(node):
 # If node is a leaf, add the nodes name and a its scientific
 # name
 if node.is_leaf():
  # Add an static face that handles the node name
  faces.add_face_to_node(nameFace, node, column=0)
  # We can also create faces on the fly
  longNameFace = faces.TextFace(code2name[node.name])
  faces.add_face_to_node(longNameFace, node, column=0)

  # text faces support multiline. We add a text face
  # with the whole description of each leaf.
  descFace = faces.TextFace(code2desc[node.name], fsize=10)
  # Note that this faces is added in "aligned" mode
  faces.add_face_to_node(descFace, node, column=0, aligned=True)

  # Sets the style of leaf nodes
  node.img_style["size"] = 12
  node.img_style["shape"] = "circle"
#If node is an internal node
 else:
  # Sets the style of internal nodes
  node.img_style["size"] = 6
  node.img_style["shape"] = "circle"
  node.img_style["fgcolor"] = "#000000"

# If an internal node contains more than 4 leaves, add the
 # images of the represented species sorted in columns of 2
 # images max.
        if len(node)>=4:
     col = 0
     for i, name in enumerate(set(node.get_leaf_names())):
  if i>0 and i%2 == 0:
      col += 1
  # Add the corresponding face to the node
                if name.startswith("Dme"):
                        faces.add_face_to_node(flyFace, node, column=col)
                elif name.startswith("Dre"):
                        faces.add_face_to_node(fishFace, node, column=col)
                elif name.startswith("Mms"):
                        faces.add_face_to_node(mouseFace, node, column=col)
                elif name.startswith("Ptr"):
                        faces.add_face_to_node(chimpFace, node, column=col)
                elif name.startswith("Hsa"):
                        faces.add_face_to_node(humanFace, node, column=col)
                elif name.startswith("Cfa"):
   faces.add_face_to_node(dogFace, node, column=col)

  # Modifies this node's style
  node.img_style["size"] = 16
  node.img_style["shape"] = "sphere"
  node.img_style["fgcolor"] = "#AA0000"

# If leaf is "Hsa" (homo sapiens), highlight it using a
 # different background.
 if node.is_leaf() and node.name.startswith("Hsa"):
  node.img_style["bgcolor"] = "#9db0cf"

# And, finally, Visualize the tree using my own layout function
t.show(mylayout)
```

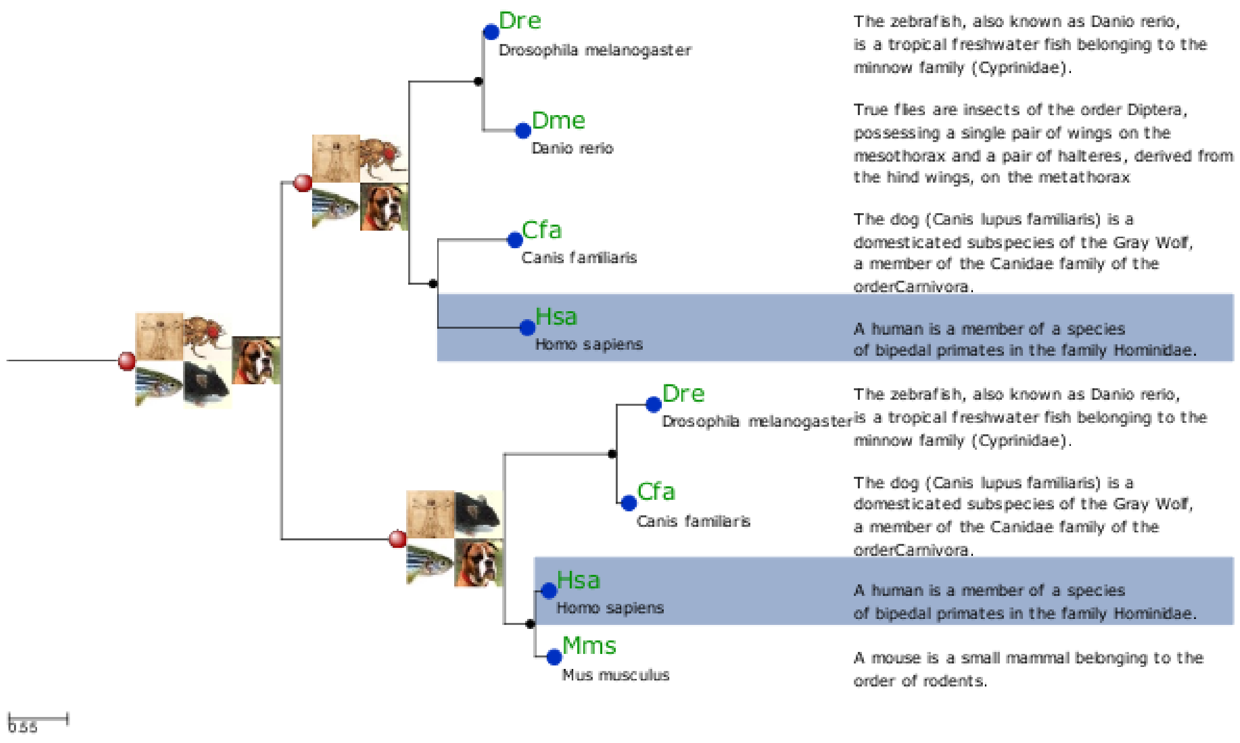

Dre
Drosophila melanogaster
The zebrafish, also known as Danio rerio, is a tropical freshwater fish belonging to the minnow family (Cyprinidae).
Dme
Danio rerio
True flies are insects of the order Diptera, possessing a single pair of wings on the mesothorax and a pair of halteres, derived from the hind wings, on the metathorax
Cfa
Canis familiaris
The dog (Canis lupus familiaris) is a domesticated subspecies of the Gray Wolf, a member of the Canidae family of the orderCarnivora.
Hsa
Homo sapiens
A human is a member of a species of bipedal primates in the family Hominidae.
Dre
Drosophila melanogaster
The zebrafish, also known as Danio rerio, is a tropical freshwater fish belonging to the minnow family (Cyprinidae).
Cfa
Canis familiaris
The dog (Canis lupus familiaris) is a domesticated subspecies of the Gray Wolf, a member of the Canidae family of the orderCarnivora.
Hsa
Homo sapiens
A human is a member of a species of bipedal primates in the family Hominidae.
Mms
Mus musculus
A mouse is a small mammal belonging to the order of rodents.
0.55

Figure 2.4: Image resulting from previous example

CHAPTER 3

PHYLOGENETIC TREES

Phylogenetic trees are the result of most evolutionary analyses. They represent the evolutionary relationships among a set of species or, in molecular biology, a set of homologous sequences.

The **PhyloTree** class provides a proper way to deal with phylogenetic trees. Thus, while leaves are assumed to represent species (or sequences from a given species genome), internal nodes are considered the ancestral states leading to current species. A consequence of this is, for instance, that each bifurcation can be considered as a speciation or a duplication event.

PhyloTree instances extend the **Tree** class with several specific method that apply only for the analysis of phylogenetic trees.

3.1 Linking Phylogenetic Trees and Multiple Sequence Alignments

PhyloTree instances allow molecular phylogenies to be linked to the Multiple Sequence Alignments (MSA). To associate a MSA with a phylogenetic tree you can use the **link_to_alignment()** method present in any PhyloTree instance, which receives the path of an MSA file as first argument or, alternatively, a text string containing the MSA. Currently, **the following sequence file formats are supported:**

format name	**alg_format key**
fasta	fasta
interleaved phylip	iphylip
sequential phylip	phylip

Table 3.1: Supported Multi Sequence File formats

Fasta format is assumed by default, but you can change this by setting the **alg_format** argument. Given that such formats are not only applicable for MSA but also for **Unaligned Sequences**, you may also associate sequences of different lengths with tree nodes. Alternatively to this method, MSAs can be directly passed to the PhyloTree constructor and sequences will be automatically linked with terminal nodes: i.e.) **PhyloTree("mytreeFile", "myAlginmentFile", format=0, alg_format="iphylip")**

Fasta format example:

```
>sequence_1
-----MKVIL LFVLAVFTVFFLEFQDKFNK KY-SHEEYLE
>sequence_2
MAHARVLLLA LAVLATAAVAFARFAVRYGK SYESAAEVRR
>sequence_3
------MWAT LPLLCAGAWLFKSWMSKHRK TY-STEEYHH
```

Interleaved phylip format example:

```
3 20
sequence_1   -----MKVIL LFVLAVFTVF
sequence_2   MAHARVLLLA LAVLATAAVA
sequence_3   ------MWAT LPLLCAGAWL

             FLEFQDKFNK KY-SHEEYLE
             FARFAVRYGK SYESAAEVRR
             FKSWMSKHRK TY-STEEYHH
```

Sequential phylip format example:

```
3 20
sequence_1   -----MKVIL LFVLAVFTVFFLEFQDKFNK KY-SHEEYLE
sequence_2   MAHARVLLLA LAVLATAAVAFARFAVRYGK SYESAAEVRR
sequence_3   ------MWAT LPLLCAGAWLFKSWMSKHRK TY-STEEYHH
```

Figure 3.1: Examples of the supported multi sequence file formats

As currently implemented, sequence linking process is not strict, which means that a perfect match between all node names and sequences names is **not required**. Thus, if only one match is found between sequences names within the MSA file and tree node names, only one tree node will contain an associated sequence. Also, it is important to note that sequence linking is not limited to terminal nodes. If internal nodes are named, and such names find a match within the provided MSA file, their corresponding sequences will be also loaded into the tree structure. Once a MSA is linked, sequences will be available for every tree node through its **node.sequence** attribute.

Example 24

```
from ete2 import PhyloTree
fasta_txt = """
 >seqA
 MAEIPDETIQQFMALT---HNIAVQYLSEFGDLNEALNSYYASQTDDIKDRREEAH
 >seqB
 MAEIPDATIQQFMALTNVSHNIAVQY--EFGDLNEALNSYYAYQTDDQKDRREEAH
 >seqC
 MAEIPDATIQ---ALTNVSHNIAVQYLSEFGDLNEALNSYYASQTDDQPDRREEAH
 >seqD
 MAEAPDETIQQFMALTNVSHNIAVQYLSEFGDLNEAL--------------REEAH
"""
iphylip_txt = """
 4 76
      seqA   MAEIPDETIQ QFMALT---H NIAVQYLSEF GDLNEALNSY YASQTDDIKD RREEAHQFMA
      seqB   MAEIPDATIQ QFMALTNVSH NIAVQY--EF GDLNEALNSY YAYQTDDQKD RREEAHQFMA
      seqC   MAEIPDATIQ ---ALTNVSH NIAVQYLSEF GDLNEALNSY YASQTDDQPD RREEAHQFMA
      seqD   MAEAPDETIQ QFMALTNVSH NIAVQYLSEF GDLNEAL--- ---------- -REEAHQ---
             LTNVSHQFMA LTNVSH
             LTNVSH---- ------
             LTNVSH---- ------
             -------FMA LTNVSH
"""
# Load a tree and link it to an alignment. As usual, 'alignment' can
# be the path to a file or data in text format.
t = PhyloTree("(((seqA,seqB),seqC),seqD);", alignment=fasta_txt, alg_format="fasta")

#We can now access the sequence of every leaf node
print "These are the nodes and its sequences:"
```

```
for leaf in t.iter_leaves():
    print leaf.name, leaf.sequence
#seqD MAEAPDETIQQFMALTNVSHNIAVQYLSEFGDLNEAL--------------REEAH
#seqC MAEIPDATIQ---ALTNVSHNIAVQYLSEFGDLNEALNSYYASQTDDQPDRREEAH
#seqA MAEIPDETIQQFMALT---HNIAVQYLSEFGDLNEALNSYYASQTDDIKDRREEAH
#seqB MAEIPDATIQQFMALTNVSHNIAVQY--EFGDLNEALNSYYAYQTDDQKDRREEAH
#
# The associated alignment can be changed at any time
t.link_to_alignment(alignment=iphylip_txt, alg_format="iphylip")
# Let's check that sequences have changed
print "These are the nodes and its re-linked sequences:"
for leaf in t.iter_leaves():
    print leaf.name, leaf.sequence
#seqD MAEAPDETIQQFMALTNVSHNIAVQYLSEFGDLNEAL--------------REEAHQ----------FMALTNVSH
#seqC MAEIPDATIQ---ALTNVSHNIAVQYLSEFGDLNEALNSYYASQTDDQPDRREEAHQFMALTNVSH----------
#seqA MAEIPDETIQQFMALT---HNIAVQYLSEFGDLNEALNSYYASQTDDIKDRREEAHQFMALTNVSHQFMALTNVSH
#seqB MAEIPDATIQQFMALTNVSHNIAVQY--EFGDLNEALNSYYAYQTDDQKDRREEAHQFMALTNVSH----------
#
# The sequence attribute is considered as node feature, so you can
# even include sequences in your extended newick format!
print t.write(features=["sequence"], format=9)
#
#
# (((seqA[&&NHX:sequence=MAEIPDETIQQFMALT---HNIAVQYLSEFGDLNEALNSYYASQTDDIKDRREEAHQF
# MALTNVSHQFMALTNVSH],seqB[&&NHX:sequence=MAEIPDATIQQFMALTNVSHNIAVQY--EFGDLNEALNSY
# YAYQTDDQKDRREEAHQFMALTNVSH----------]),seqC[&&NHX:sequence=MAEIPDATIQ---ALTNVSHNIA
# VQYLSEFGDLNEALNSYYASQTDDQPDRREEAHQFMALTNVSH----------]),seqD[&&NHX:sequence=MAEAPD
# ETIQQFMALTNVSHNIAVQYLSEFGDLNEAL--------------REEAHQ----------FMALTNVSH]);
#
# And yes, you can save this newick text and reload it into a PhyloTree instance.
sametree = PhyloTree(t.write(features=["sequence"]))
print "Recovered tree with sequence features:"
print sametree
#
#                              /-seqA
#                    /--------|
#          /--------|          \-seqB
#         |         |
#---------|          \-seqC
#         |
#          \-seqD
#
print "seqA sequence:", (t&"seqA").sequence
# MAEIPDETIQQFMALT---HNIAVQYLSEFGDLNEALNSYYASQTDDIKDRREEAHQFMALTNVSHQFMALTNVSH
```

3.2 Using Taxonomic Data

PhyloTree instances allow to deal with leaf names and species names separately. This is useful when working with molecular phylogenies, in which leaves are usually encoded using sequence names but species names. You could easily solve this by annotating each terminal node according to its source species. However, PhyloTree instances can automatically deal with this issue. Thus, when a phylogenetic tree is loaded, species names (codes, key names or fingerprints) are automatically derived from the **three first letters of leaf names**. Although, you can indeed change this behavior by using a custom parsin function. By doing this, you can easily load taxonomy-aware molecular phylogenies. The attribute **node.species** will be present in every node and stores the inferred species name, while the method **get_species()** can be used to retrieve all species names under a given ancestral node.

There are two ways of setting the automatic species name generation:

1. using the PhyloTree **sp_naming_function** argument. The whole tree structure will be initialized to use the provided parsing function to obtain species name information.

2. using the **set_species_naming_function** method (present in all tree nodes), which can be used to change the behavior in a previously loaded tree, or to set different parsing function to different

3. parts of the tree.

In both cases, possible values are **None** (to disable automatic generation of species names) or the **reference to a custom python function**.

Example 25

```
from ete2 import PhyloTree
# Reads a phylogenetic tree (using default species name encoding)
t = PhyloTree("(((Hsa_001,Ptr_001),(Cfa_001,Mms_001)),(Dme_001,Dme_002));")
#                              /-Hsa_001
#                    /--------|
#                   |          \-Ptr_001
#          /--------|
#         |         |          /-Cfa_001
#         |          \--------|
#---------|                    \-Mms_001
#         |
#         |          /-Dme_001
#          \--------|
#                    \-Dme_002
#
# Prints current leaf names and species codes
print "Deafult mode:"
for n in t.get_leaves():
    print "node:", n.name, "Species name:", n.species
# node: Dme_001 Species name: Dme
# node: Dme_002 Species name: Dme
# node: Hsa_001 Species name: Hsa
# node: Ptr_001 Species name: Ptr
# node: Cfa_001 Species name: Cfa
# node: Mms_001 Species name: Mms
#
# We can also use our own leaf name parsing function to obtain species
# names. All we need to do is create a python function that takes
# node's name as argument and return its corresponding species name.
def get_species_name(node_name_string):
    # Species code is the first part of leaf name (separated by an
    #  underscore character)
    spcode = node_name_string.split("_")[0]
    # We could even translate the code to complete names
    code2name = {
      "Dme":"Drosophila melanogaster",
      "Hsa":"Homo sapiens",
      "Ptr":"Pan troglodytes",
      "Mms":"Mus musculus",
      "Cfa":"Canis familiaris"
      }
    return code2name[spcode]
# Now, let's ask the tree to use our custom species naming function
t.set_species_naming_function(get_species_name)
print "Custom mode:"
for n in t.get_leaves():
    print "node:", n.name, "Species name:", n.species
# node: Dme_001 Species name: Drosophila melanogaster
# node: Dme_002 Species name: Drosophila melanogaster
# node: Hsa_001 Species name: Homo sapiens
# node: Ptr_001 Species name: Pan troglodytes
# node: Cfa_001 Species name: Canis familiaris
# node: Mms_001 Species name: Mus musculus
#
# Of course, you can disable the automatic generation of species
# names. To do so, you can set the species naming function to
# None. This is useful to set the species names manually or for
# reading them from a newick file. Other wise, species attribute would
# be overwriten
mynewick = """
(((Hsa_001[&&NHX:species=Human],Ptr_001[&&NHX:species=Chimp]),
(Cfa_001[&&NHX:species=Dog],Mms_001[&&NHX:species=Mouse])),
(Dme_001[&&NHX:species=Fly],Dme_002[&&NHX:species=Fly]));
"""
t = PhyloTree(mynewick, sp_naming_function=None)
print "Disabled mode (manual set):"
for n in t.get_leaves():
    print "node:", n.name, "Species name:", n.species
# node: Dme_001 Species name: Fly
# node: Dme_002 Species name: Fly
# node: Hsa_001 Species name: Human
# node: Ptr_001 Species name: Chimp
# node: Cfa_001 Species name: Dog
# node: Mms_001 Species name: Mouse
#
# Of course, once this info is available you can query any internal
# node for species covered.
human_mouse_ancestor = t.get_common_ancestor("Hsa_001", "Mms_001")
print "These are the species under the common ancestor of Human & Mouse"
print '\n'.join( human_mouse_ancestor.get_species() )
# Mouse
# Chimp
# Dog
# Human
```

3.3 Dating Phylogenetic Nodes

Nodes in molecular phylogenies can be interpreted as evolutionary events. They can represent the duplication of an ancestral sequence or the speciation event that separated the evolution of two ancestral sequences. In any case, because nodes represent ancestral events, they can be located at a given moment in the evolution. This is, we can date evolutionary events.

There are many ways to infer such information. Most approaches are based on the comparison of the sequences affected by a given event. However, these methods suffer from several limitations (REF). An alternative approach that has been shown to overcome some of such limitations is to date evolutionary events according the topology of phylogenetic trees (Jaime Huerta-Cepas and Toni Gabaldon, 2009, submmited).

In brief, the relative age of any evolutionary event can be established by detecting the oldest taxonomic group affected by such event. Given that in phylogenies nodes are events, this is something that can be easily evaluated by looking at the species under each node. Although this task can be done manually, ETE implements a method to automatize the process. Thus, by defining a python dictionary containing the conversion between **species names** and the considered **taxonomic levels,** phylogenetic nodes can be easily dated. The **get_age()** method, found in every node, can be used to this end. Obviously, the more taxonomic levels are defined, the more precise is time estimation. For instance, if we consider a tree in which several vertebrate species are represented, we could define an age dictionary like this:

```
# We consider only the following taxonomic levels
# regarding vertebrates: 1: hominids, 2:primates,
# 3:mammals, 4:amphibians, 5:fishes
taxa_levels = {
  "Human": 1, # Hominids
  "Chimp": 2, # Primates
  "Macaca": 2, # Primates
  "Mouse": 3, # Mammals
  "Cow": 3, # Mammals
  "Frog": 4, # Amphibians
  "Zebrafish": 5,  # fishes
  "Takifugu": 5, #fishes
}
```

In which each number refers to a taxonomic group, and older taxonomic groups have higher values. Then, any internal node could be easily mapped to an evolutionary period by executing: **node.get_date(vertebrates_taxa_levels)**.

Example 26

```
from ete2 import PhyloTree
# Creates a gene phylogeny with several duplication events at
# different levels. Note that we are using the default method for
# detecting the species code of leaves (three first lettes in the node
# name are considered the species code).
nw = """
((Dme_001,Dme_002),(((Cfa_001,Mms_001),((((Hsa_001,Hsa_003),Ptr_001)
,Mmu_001),((Hsa_004,Ptr_004),Mmu_004))),(Ptr_002,(Hsa_002,Mmu_002))));
"""
t = PhyloTree(nw)
print "Original tree:",
print t
#
#             /-Dme_001
#   /--------|
#  |          \-Dme_002
#  |
#  |                              /-Cfa_001
#  |                    /--------|
#  |                   |          \-Mms_001
#  |                   |
#--|                   |                                        /-Hsa_001
#  |                   |                              /--------|
#  |          /--------|                    /--------|          \-Hsa_003
#  |         |         |                   |         |
#  |         |         |          /--------|          \-Ptr_001
#  |         |         |         |         |
#  |         |         |         |          \-Mmu_001
#  |         |          \--------|
#   \--------|                   |                    /-Hsa_004
#            |                   |          /--------|
#            |                    \--------|          \-Ptr_004
#            |                             |
#            |                              \-Mmu_004
#            |
#            |          /-Ptr_002
#              \--------|
#                       |          /-Hsa_002
#                        \--------|
#                                  \-Mmu_002
# Create a dictionary with relative ages for the species present in
# the phylogenetic tree.  Note that ages are only relative numbers to
# define which species are older, and that different species can
# belong to the same age.
species2age = {
  'Hsa': 1, # Homo sapiens (Hominids)
  'Ptr': 2, # P. troglodytes (primates)
  'Mmu': 2, # Macaca mulata (primates)
  'Mms': 3, # Mus musculus (mammals)
  'Cfa': 3, # Canis familiaris (mammals)
  'Dme': 4  # Drosophila melanogaster (metazoa)
}
# We can translate each number to its correspondig taxonomic number
age2name = {
  1:"hominids",
  2:"primates",
  3:"mammals",
  4:"metazoa"
}
event1= t.get_common_ancestor("Hsa_001", "Hsa_004")
event2=t.get_common_ancestor("Hsa_001", "Hsa_002")
print
print "The duplication event leading to the human sequences Hsa_001 and "+\
     "Hsa_004 is dated at: ", age2name[event1.get_age(species2age)]
print "The duplication event leading to the human sequences Hsa_001 and "+\
     "Hsa_002 is dated at: ", age2name[event2.get_age(species2age)]
# The duplication event leading to the human sequences Hsa_001 and Hsa_004
# is dated at:  primates
#
# The duplication event leading to the human sequences Hsa_001 and Hsa_002
# is dated at:  mammals
```

3.4 Detecting evolutionary events

There are several ways to automatically detect duplication and speciation nodes within molecular phylogenies. ETE provides the two most extended methodologies. One implements the algorithm described in [Huerta-Cepas et al, 2007] and is based on the species overlap between partitions and thus does not depend on the availability of a species tree (species overlap). The second one, which requires the comparison between the gene tree and a previously defined species tree, implements a strict tree reconciliation algorithm [Page and Charleston, 1997]. By detecting evolutionary events, orthology and paralogy relationships among sequences are also inferred.

Both methods, species overlap and tree reconciliation, can be used to **label each tree node as a duplication or speciation event**. Thus, after applying any of the algorithms, original tree nodes will contain a new attribute named **evoltype**, which can take the following values: **"D" (duplication), "S" (speciation), "L" (lost linage)**. Additionally, a list of all the detected events is returned. Each event is a python object of type **EvolEvent**, containing its basic information:

`event.etype`: `event type ( "D", "S" or "L")`

`event.in_seqs`: `A list of sequences at one side of the event .`

`event.out_seqs`: `A list of sequences at the other side of the event.`

event.node: `Link to the phylogenetic node that defines the event`

event.sos: `Species Overlap Score (None if tree reconciliation was used)`

Other attributes may be found in events instances, however they are not stable yet.

If an event represents a duplication, `in_seqs` **are all paralogous** to `out_seqs`. Similarly, if an event represents a speciation, `in_seqs` **are all orthologous** to `out_seqs`.

While tree reconciliation must always be used from an internal node, species overlap allows to track only all the evolutionary events involving a specific tree leaf.

3.4.1 Species Overlap (SO) algorithm

In order to apply the SO algorithm, you can use the **node.get_descendant_evol_events()** method (it will map all events under the current node) or the **node.get_my_evol_events()** method (it will map only the events involving the current node, usually a leaf node).

By default the **species overlap score (SOS) threshold** is set to 0.0, which means that a single species in common between two node branches will rise a duplication event. This has been shown to preform the best with real data, however you can adjust the threshold using the **sos_thr** argument present in both methods.

Example 27

```
from ete2 import PhyloTree
# Loads an example tree
nw = """
((Dme_001,Dme_002),(((Cfa_001,Mms_001),((Hsa_001,Ptr_001),Mmu_001)),
(Ptr_002,(Hsa_002,Mmu_002))));
"""
t = PhyloTree(nw)
print t
#                    /-Dme_001
#          /--------|
#         |          \-Dme_002
#         |
#         |                              /-Cfa_001
#         |                    /--------|
#---------|                   |          \-Mms_001
#         |          /--------|
#         |         |         |                    /-Hsa_001
#         |         |         |          /--------|
#         |         |          \--------|          \-Ptr_001
#          \--------|                   |
#                   |                    \-Mmu_001
#                   |
#                   |          /-Ptr_002
#                    \--------|
#                             |          /-Hsa_002
#                              \--------|
#                                        \-Mmu_002
#
# To obtain all the evolutionary events involving a given leaf node we
# use get_my_evol_events method
matches = t.search_nodes(name="Hsa_001")
human_seq = matches[0]
# Obtains its evolutionary events
events = human_seq.get_my_evol_events()
# Print its orthology and paralogy relationships
print "Events detected that involve Hsa_001:"
```

```
for ev in events:
    if ev.etype == "S":
        print '   ORTHOLOGY RELATIONSHIP:', ','.join(ev.in_seqs), "<====>", ','.join(ev.out_seqs)
    elif ev.etype == "D":
        print '   PARALOGY RELATIONSHIP:', ','.join(ev.in_seqs), "<====>", ','.join(ev.out_seqs)

# Alternatively, you can scan the whole tree topology
events = t.get_descendant_evol_events()
# Print its orthology and paralogy relationships
print "Events detected from the root of the tree"
for ev in events:
    if ev.etype == "S":
        print '   ORTHOLOGY RELATIONSHIP:', ','.join(ev.in_seqs), "<====>", ','.join(ev.out_seqs)
    elif ev.etype == "D":
        print '   PARALOGY RELATIONSHIP:', ','.join(ev.in_seqs), "<====>", ','.join(ev.out_seqs)

# If we are only interested in the orthology and paralogy relationship
# among a given set of species, we can filter the list of sequences
#
# fseqs is a function that, given a list of sequences, returns only
# those from human and mouse
fseqs = lambda slist: [s for s in slist if s.startswith("Hsa") or s.startswith("Mms")]
print "Paralogy relationships among human and mouse"
for ev in events:
    if ev.etype == "D":
 # Prints paralogy relationships considering only human and
 # mouse. Some duplication event may not involve such species,
 # so they will be empty
        print '   PARALOGY RELATIONSHIP:', \
     ','.join(fseqs(ev.in_seqs)), \
     "<====>",\
     ','.join(fseqs(ev.out_seqs))

# Note that besides the list of events returned, the detection
# algorithm has labeled the tree nodes according with the
# predictions. We can use such lables as normal node features.
dups = t.search_nodes(evoltype="D") # Return all duplication nodes
```

3.4.2 Example2: Tree reconciliation algorithm

Tree reconciliation algorithm uses a predefined species tree to infer the genes losses that explain a given gene tree topology. By doing this, it infers also the duplication and speciation events. To perform a strict tree reconciliation analysis over a given node in a molecular phylogeny you can use the **node.reconcile()** method, which requires a species tree as its first argument. The species tree (another PhyloTree instance) must contain the topology of the species represented in the gene tree. Moreover, leaf names in the species tree must match the species names in the gene tree (by default, the first 3 letters of the gene tree leaf names) (see 3.2).

As a result, the **reconcile()** method will label the original gene tree nodes as duplication or speciation, will return the list of inferred events, and will return a new **reconcilied tree**, in which inferred gene losses are present and labeled.

Example 28

```
from ete2 import PhyloTree

# Loads a gene tree and its corresponding species tree. Note that
# species names in sptree are the 3 firs letters of leaf nodes in
# genetree.
```

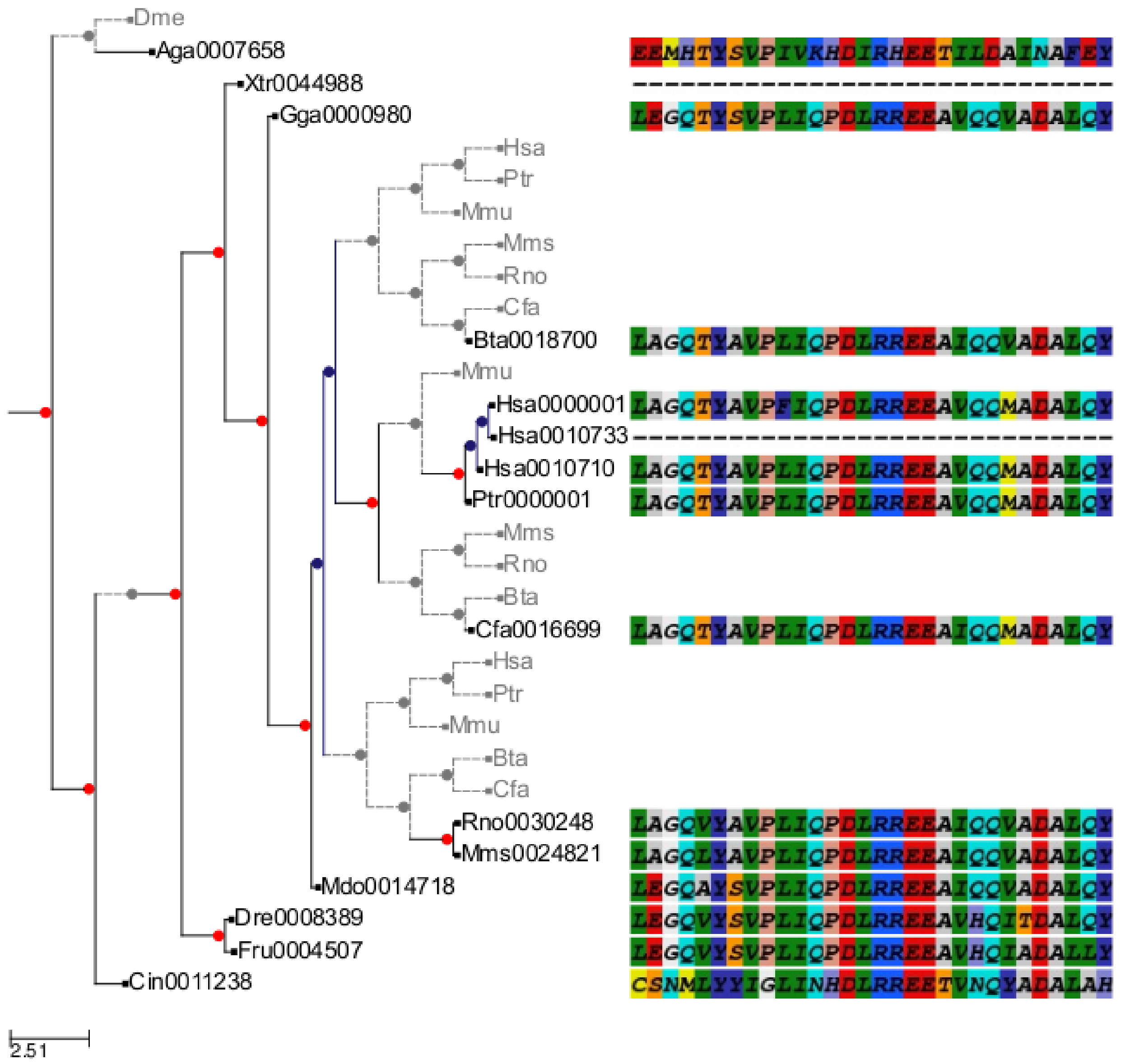
```
gene_tree_nw = ’((Dme_001,Dme_002),(((Cfa_001,Mms_001),((Hsa_001,Ptr_001),Mmu_001)),(Ptr_002,(Hsa_002,Mmu_002)
species_tree_nw = "((((Hsa, Ptr), Mmu), (Mms, Cfa)), Dme);"
genetree = PhyloTree(gene_tree_nw)
sptree = PhyloTree(species_tree_nw)
print genetree
#                    /-Dme_001
#          /--------|
#         |          \-Dme_002
#         |
#         |                              /-Cfa_001
#         |                    /--------|
#---------|                   |          \-Mms_001
#         |          /--------|
#         |         |         |                    /-Hsa_001
#         |         |         |          /--------|
#         |         |          \--------|          \-Ptr_001
#          \--------|                   |
#                   |                    \-Mmu_001
#                   |
#                   |          /-Ptr_002
#                    \--------|
#                             |          /-Hsa_002
#                              \--------|
#                                        \-Mmu_002
#
# Let’s reconcile our genetree with the species tree
recon_tree, events = genetree.reconcile(sptree)
# a new "reconcilied tree" is returned. As well as the list of
# inferred events.
print "Orthology and Paralogy relationships:"
for ev in events:
    if ev.etype == "S":
        print ’ORTHOLOGY RELATIONSHIP:’, ’,’.join(ev.inparalogs), "<====>", ’,’.join(ev.orthologs)
    elif ev.etype == "D":
        print ’PARALOGY RELATIONSHIP:’, ’,’.join(ev.inparalogs), "<====>", ’,’.join(ev.outparalogs)
# And we can explore the resulting reconciled tree
print recon_tree
# You will notice how the reconcilied tree is the same as the gene
# tree with some added branches. They are inferred gene losses.
#
#
#                    /-Dme_001
#          /--------|
#         |          \-Dme_002
#         |
#         |                              /-Cfa_001
#         |                    /--------|
#         |                   |          \-Mms_001
#---------|          /--------|
#         |         |         |                    /-Hsa_001
#         |         |         |          /--------|
#         |         |          \--------|          \-Ptr_001
#         |         |                   |
#         |         |                    \-Mmu_001
#          \--------|
#                   |                    /-Mms
#                   |          /--------|
#                   |         |          \-Cfa
#                   |         |
#                   |         |                              /-Hsa
#                    \--------|                    /--------|
#                             |          /--------|          \-Ptr_002
#                             |         |         |
#                             |         |          \-Mmu
#                              \--------|
#                                       |                    /-Ptr
#                                       |          /--------|
#                                        \--------|          \-Hsa_002
#                                                   |
#                                                    \-Mmu_002
#
# And we can visualize the trees using the default phylogeny
# visualization layout
genetree.show()
recon_tree.show()
```

3.5 Visualization of phylogenetic trees

A special set of visualization rules (see chapter 2) are provided with the phylogenetic extension as the **phylogeny** layout function. By default, this layout function will be used to show and render any PhyloTree instance, thus handling the visualization of MSAs, evolutionary events, and taxonomic information. However, you can change/extend this layout by providing a custom layout function.

The **SeqFace()** class is also provided for convenience. It allows to add nodes faces with the coloured sequence associated to each node.

3.5.1 Example: A reconciled tree showing inferred evolutionary events, gene losses and node's sequences


Dme
Aga0007658
EEMHTYSVPIVKHDIRHEETILDAINAFEY
Xtr0044988

Gga0000980
LEGQTYSVPLIQPDLRREEAVQQVADALQY
Hsa
Ptr
Mmu
Mms
Rno
Cfa
Bta0018700
LAGQTYAVPLIQPDLRREEAIQQVADALQY
Mmu
Hsa0000001
LAGQTYAVPFIQPDLRREEAVQQMADALQY
Hsa0010733

Hsa0010710
LAGQTYAVPLIQPDLRREEAVQQMADALQY
Ptr0000001
LAGQTYAVPLIQPDLRREEAVQQMADALQY
Mms
Rno
Bta
Cfa0016699
LAGQTYAVPLIQPDLRREEAIQQMADALQY
Hsa
Ptr
Mmu
Bta
Cfa
Rno0030248
LAGQVYAVPLIQPDLRREEAIQQVADALQY
Mms0024821
LAGQLYAVPLIQPDLRREEAIQQVADALQY
Mdo0014718
LEGQAYSVPLIQPDLRREEAIQQVADALQY
Dre0008389
LEGQVYSVPLIQPDLRREEAVHQITDALQY
Fru0004507
LEGQVYSVPLIQPDLRREEAVHQIADALLY
Cin0011238
CSNMLYYIGLINHDLRREETVNQYADALAH
2.51

Figure 3.2: A reconciled tree showing duplication events (blue nodes), speciation (red nodes), gene losses (gray dashed lines) and the original sequences used to reconstruct the phylogeny.

CHAPTER 4

CLUSTERING TREES

Cluster analysis is the assignment of a set of observations into subsets (called clusters) so that observations in the same cluster are similar in some sense. Clustering is a method of unsupervised learning, and a common technique for statistical data analysis used in many fields, including machine learning, data mining, pattern recognition, image analysis and bioinformatics. Hierarchical clustering creates a hierarchy of clusters which may be represented in a tree structure called a dendrogram. The root of the tree consists of a single cluster containing all observations, and the leaves correspond to individual observations. [The Wikipedia porject Jun-2009].

ETE provides the **ClusterTree** instance, which inherits all the basic tree methods and and adds support for linking clustering trees with the numerical matrix that were used to compute the clustering. It also implements several clustering validation techniques that aid in the analysis of cluster quality. Currently, inter and intra-cluster distances, cluster std.deviation, Silhouette analysis and Dunn indexes can be computed with ETE. Indeed, this type of trees can be used, not only as clustering trees, but as any other type requiring numerical vectors to be associated with tree nodes: i.e.) phylogenetic profiles, microarray expression datasets or node fingerprint vectors.

ClusterTrees can be linked to a numerical matrix by using the **link_to_arraytable()** method (in nodes) or by passing the reference to the matrix (filename or text string) as the **text_arraytable** argument of PhyloTree constructor. Once this is done, the **node.profile, node.deviation, node.silhouette, node.dunn, node.intracluster_dist and node.intercluster_dist** attributes will are automatically available. As well as the **iter_leaf_profiles()**, **get_leaf_profiles()**, **get_silhouette()** and **get_dunn()** methods.

Example 29

```
from ete2 import ClusterTree

# Example of a minimalistic numerical matrix. It is encoded as a text
# string for convenience, but it usally be loaded from a text file.
matrix = """
#Names\tcol1\tcol2\tcol3\tcol4\tcol5\tcol6\tcol7
A\t-1.23\t-0.81\t1.79\t0.78\t-0.42\t-0.69\t0.58
B\t-1.76\t-0.94\t1.16\t0.36\t0.41\t-0.35\t1.12
C\t-2.19\t0.13\t0.65\t-0.51\t0.52\t1.04\t0.36
D\t-1.22\t-0.98\t0.79\t-0.76\t-0.29\t1.54\t0.93
E\t-1.47\t-0.83\t0.85\t0.07\t-0.81\t1.53\t0.65
F\t-1.04\t-1.11\t0.87\t-0.14\t-0.80\t1.74\t0.48
G\t-1.57\t-1.17\t1.29\t0.23\t-0.20\t1.17\t0.26
H\t-1.53\t-1.25\t0.59\t-0.30\t0.32\t1.41\t0.77
"""
print "Example numerical matrix"
print matrix
# #Names   col1     col2     col3     col4     col5     col6     col7
# A        -1.23    -0.81    1.79     0.78     -0.42    -0.69    0.58
# B        -1.76    -0.94    1.16     0.36     0.41     -0.35    1.12
# C       -2.19   0.13    0.65    -0.51   0.52    1.04    0.36
# D       -1.22   -0.98   0.79    -0.76   -0.29   1.54    0.93
# E       -1.47   -0.83   0.85    0.07    -0.81   1.53    0.65
# F       -1.04   -1.11   0.87    -0.14   -0.80   1.74    0.48
# G       -1.57   -1.17   1.29    0.23    -0.20   1.17    0.26
# H       -1.53   -1.25   0.59    -0.30   0.32    1.41    0.77
#
#
# We load a tree structure whose leaf nodes correspond to rows in the
# numerical matrix. We use the text_array argument to link the tree
# with numerical matrix.
t = ClusterTree("(((A,B),(C,(D,E))),(F,(G,H)));", text_array=matrix)
print "Example tree", t
#                              /-A
#                    /--------|
#                   |          \-B
#          /--------|
#         |         |          /-C
#         |          \--------|
#         |                   |          /-D
#---------|                    \--------|
#         |                             \-E
#         |
#         |          /-F
#          \--------|
#                   |          /-G
#                    \--------|
#                              \-H

# Now we can ask the numerical profile associated to each node
A = t.search_nodes(name='A')[0]
print "A associated profile:\n", A.profile
# [-1.23 -0.81  1.79  0.78 -0.42 -0.69  0.58]
#
# Or we can ask for the mean numerical profile of an internal
# partition, which is computed as the average of all vectors under the
# the given node.
cluster = t.get_common_ancestor("E", "A")
print "Internal cluster mean profile:\n", cluster.profile
#[-1.574 -0.686  1.048 -0.012 -0.118  0.614  0.728]
#
# We can also obtain the std. deviation vector of the mean profile
print "Internal cluster std deviation profile:\n", cluster.deviation
#[ 0.36565558  0.41301816  0.40676283  0.56211743  0.50704635  0.94949671
#  0.26753691]
# If would need to re-link the tree to a different matrix or use
# different matrix for different sub parts of the tree, we can use the
# link_to_arraytable method()
#
# Creates a matrix with all values = 1
matrix_ones = """
#Names\tcol1\tcol2\tcol3\tcol4\tcol5\tcol6\tcol7
A\t1\t1\t1\t1\t1\t1\t1
B\t1\t1\t1\t1\t1\t1\t1
C\t1\t1\t1\t1\t1\t1\t1
D\t1\t1\t1\t1\t1\t1\t1
E\t1\t1\t1\t1\t1\t1\t1
F\t1\t1\t1\t1\t1\t1\t1
G\t1\t1\t1\t1\t1\t1\t1
H\t1\t1\t1\t1\t1\t1\t1
"""
# Creates a matrix with all values = 0
matrix_zeros = """
#Names\tcol1\tcol2\tcol3\tcol4\tcol5\tcol6\tcol7
A\t0\t0\t0\t0\t0\t0\t0
B\t0\t0\t0\t0\t0\t0\t0
C\t0\t0\t0\t0\t0\t0\t0
D\t0\t0\t0\t0\t0\t0\t0
```

```
E\t0\t0\t0\t0\t0\t0\t0
F\t0\t0\t0\t0\t0\t0\t0
G\t0\t0\t0\t0\t0\t0\t0
H\t0\t0\t0\t0\t0\t0\t0
"""
# Re-associate left part of the tree with matrix of 1s, and right part
# with matrix of 0s. Note that rows without matches in both are
# obviated from association.
t.children[0].link_to_arraytable(matrix_ones)
t.children[1].link_to_arraytable(matrix_zeros)
print "A profile (using matrix with 1s", (t&"A").profile
print "H profile (using matrix with 0s)", (t&"H").profile
#A profile (using matrix with 1s [ 1.  1.  1.  1.  1.  1.  1.]
#H profile (using matrix with 0s) [ 0.  0.  0.  0.  0.  0.  0.]
```

4.1 Visualization of matrix associated Trees

Clustering or not, any ClusterTree instance, associated to a numerical matrix, can be visualized together with the graphical representation of its node's numeric profiles. To this end, the **ProfileFace** class is provided. This face type can represent a node's numeric profile in four different ways:

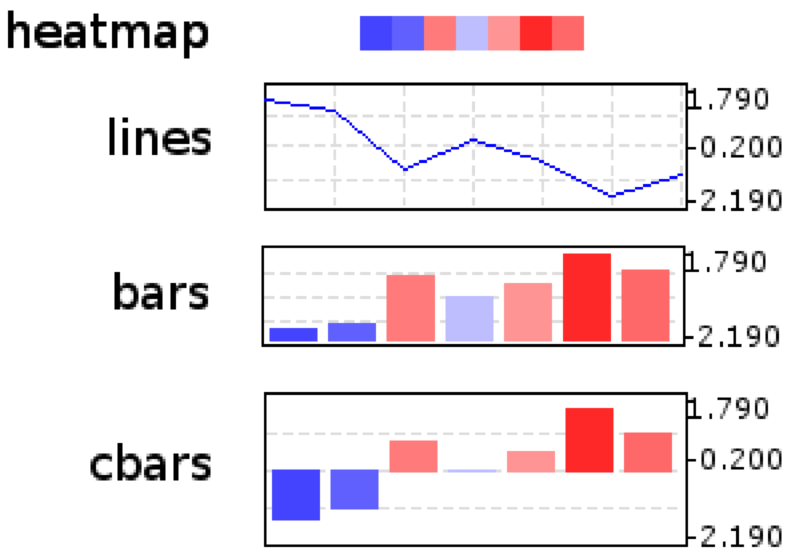


Figure 4.1: Types of ProfileFaces

You can create your own layout functions add one or more ProfileFace instances to any cluster tree node (leaf or internal). Moreover, three basic layouts are provided that use different styles of ProfileFace instances: **heatmap**, **line_profiles**, **bar_profiles**, **cbar_profiles**.

Example 30

```
# Import Tree instance and faces module
from ete2 import ClusterTree

# Example of a minimalistic numerical matrix. It is encoded as a text
# string for convenience, but it usally be loaded from a text file.
```

```
matrix = """
#Names\tcol1\tcol2\tcol3\tcol4\tcol5\tcol6\tcol7
A\t-1.23\t-0.81\t1.79\t0.78\t-0.42\t-0.69\t0.58
B\t-1.76\t-0.94\t1.16\t0.36\t0.41\t-0.35\t1.12
C\t-2.19\t0.13\t0.65\t-0.51\t0.52\t1.04\t0.36
D\t-1.22\t-0.98\t0.79\t-0.76\t-0.29\t1.54\t0.93
E\t-1.47\t-0.83\t0.85\t0.07\t-0.81\t1.53\t0.65
F\t-1.04\t-1.11\t0.87\t-0.14\t-0.80\t1.74\t0.48
G\t-1.57\t-1.17\t1.29\t0.23\t-0.20\t1.17\t0.26
H\t-1.53\t-1.25\t0.59\t-0.30\t0.32\t1.41\t0.77
"""
print "Example numerical matrix"
print matrix
# #Names  col1    col2    col3    col4    col5    col6    col7
# A       -1.23   -0.81   1.79    0.78    -0.42   -0.69   0.58
# B       -1.76   -0.94   1.16    0.36    0.41    -0.35   1.12
# C       -2.19   0.13    0.65    -0.51   0.52    1.04    0.36
# D       -1.22   -0.98   0.79    -0.76   -0.29   1.54    0.93
# E       -1.47   -0.83   0.85    0.07    -0.81   1.53    0.65
# F       -1.04   -1.11   0.87    -0.14   -0.80   1.74    0.48
# G       -1.57   -1.17   1.29    0.23    -0.20   1.17    0.26
# H       -1.53   -1.25   0.59    -0.30   0.32    1.41    0.77
#
#
# We load a tree structure whose leaf nodes correspond to rows in the
# numerical matrix. We use the text_array argument to link the tree
# with numerical matrix.
t = ClusterTree("(((A,B),(C,(D,E))),(F,(G,H)));", text_array=matrix)
# Try the default layout using ProfileFaces
t.show("heatmap")
t.show("cluster_cbars")
t.show("cluster_bars")
t.show("cluster_lines")
```

4.2 Performing a Cluster Validation Analysis

If associated matrix represents the dataset used to produce a given tree, clustering validation values can be used to assess the quality of partitions. To do so, you will need to set the distance function that was used to calculate distances among items (leaf nodes). ETE implements three common distance methods in bioinformatics : **euclidean**, **pearson** correlation and **spearman** correlation distances.

In the following example, a microarray clustering result is analyzed using ETE.

The following example illustrates how to implement a custom clustering validation analysis:

Example 31

```
from ete2 import ClusterTree,  faces

# To operate with numbersd bub efficiently
import numpy

PATH = "./"
# Loads tree and array
t = ClusterTree(PATH+"diauxic.nw", PATH+"diauxic.array")

# nodes are linked to the array table
array =  t.arraytable

# Calculates some values we will use later
```

```
matrix_max = numpy.max(array.matrix)
matrix_min = numpy.min(array.matrix)
matrix_avg = matrix_min+((matrix_max-matrix_min)/2)

# Creates a profile face that will represent node's profile as a
# heatmap
profileFace  = faces.ProfileFace(matrix_max, matrix_min, matrix_avg, \
      200, 14, "heatmap")
cbarsFace = faces.ProfileFace(matrix_max,matrix_min,matrix_avg,200,70,"cbars")
nameFace = faces.AttrFace("name", fsize=8)
# Creates my own layout function that uses previous faces
def mylayout(node):
 # If node is a leaf
        if node.is_leaf():
  # And a line profile
  faces.add_face_to_node(profileFace, node, 0, aligned=True)
  node.img_style["size"]=0
  faces.add_face_to_node(nameFace, node, 1, aligned=True)

# If node is internal
        else:
  # If silhouette is good, creates a green bubble
  if node.silhouette>0:
   validationFace = faces.TextFace("Silh=%0.2f" %node.silhouette, "Verdana", 10, "#056600")
   node.img_style["fgcolor"]="#056600"
  # Otherwise, use red bubbles
  else:
   validationFace = faces.TextFace("Silh=%0.2f" %node.silhouette, "Verdana", 10, "#940000")
   node.img_style["fgcolor"]="#940000"

  # Sets node size proportional to the silhouette value.
  node.img_style["shape"]="sphere"
  if node.silhouette<=1 and node.silhouette>=-1:
   node.img_style["size"]= 15+int((abs(node.silhouette)*10)**2)

  # If node is very internal, draw also a bar diagram
  # with the average expression of the partition
  faces.add_face_to_node(validationFace, node, 0)
  if len(node)>100:
   faces.add_face_to_node(cbarsFace, node, 1)

# Use my layout to visualize the tree
t.show(mylayout)
```

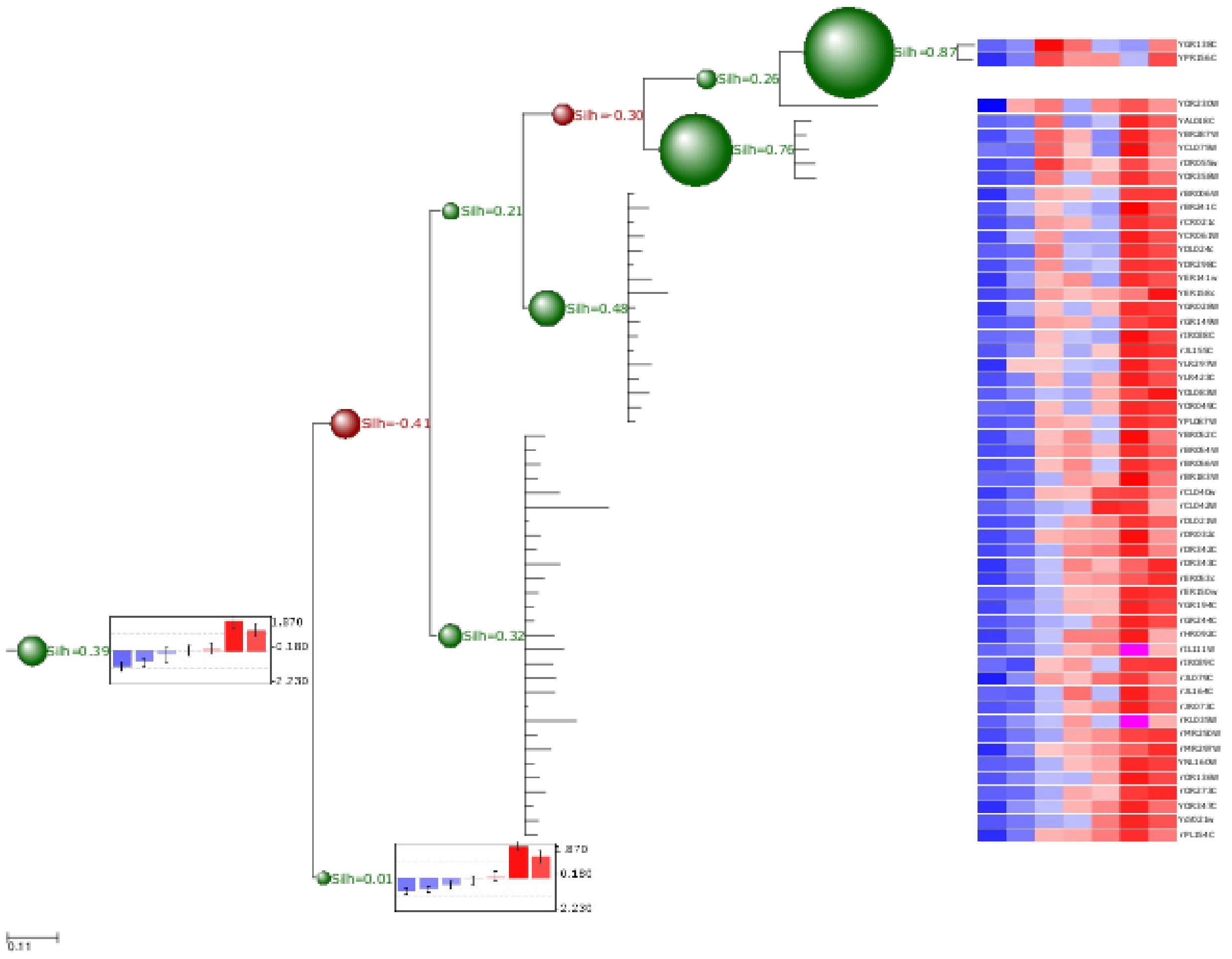


Figure 4.2: Image resulting from a microarray clustering validation analysis. Red bubbles represent a bad silhouette index (S<0), while green represents good silhouette index (S>0). Size of bubbles is proportional to the Silhouette index. Internal nodes are drawn with the average expression profile grouped by their partitions. Leaf node profiles are shown as a heatmap.

.

CHAPTER 5

THE PHYLOMEDB API

PhylomeDB is a public database for complete collections of gene phylogenies (phylomes). It allows users to interactively explore the evolutionary history of genes through the visualization of phylogenetic trees and multiple sequence alignments. Moreover, phylomeDB provides genome-wide orthology and paralogy predictions which are based on the analysis of the phylogenetic trees. The automated pipeline used to reconstruct trees aims at providing a high-quality phylogenetic analysis of different genomes , including Maximum Likelihood or Bayesian tree inference, alignment trimming and evolutionary model testing. PhylomeDB includes also a public download section with the complete set of trees, alignments and orthology predictions.

ETE's phylomeDB extension provides an access API to the main PhylomeDB database, thus allowing to search for and fetch precomputed gene phylogenies.

5.1 Basis of the phylomeDB API usage

In order to explore the database resources, you have to create a connector to the database, which will be used to query it. To do so, you must use the **PhylomeDBConnector** class and specify the parameters of the DB connection.

Example 32

```
from ete2 import PhylomeDBConnector
# This connects to the main phylomeDB server (default parameters)
p = PhylomeDBConnector()
# This connects to a local version of phylomeDB, and you can set the
# user and password arguments
p = PhylomeDBConnector(host="localhost", user="public", passwd="public", port=3306)
```

The PhylomeDBConnector constructor will return a pointer to the DB that you can use to perform queries. All methods starting by **get_** can be used to retrieve information from the database. A complete list of available methods can be found in the ETE's programming guide (available at ete.cgenomics.org) or explored by executing **dir(PhylomeDBConnector)** in a python console.

5.2 PhylomeDB structure

A phylome includes thousands of gene trees associated to the different genes/proteins of a given species. Thus, for example, the human phylome includes more than 20.000 phylogenetic trees; on per human gene. Moreover, the same gene may be associated to different trees within the same phylome differing only in the evolutionary model that assumed to reconstruct the phylogeny.

Given that each phylogenetic tree was reconstructed using a a single gene as the seed sequence to find homologous in other species, the tree takes the name from the seed sequence.

You can obtain a full list of phylomes through the **get_phylomes()** and a full list of seed sequence in a phylome using the **get_seed_ids()** method. Phylogenetic trees within a given phylome were reconstructed in a context of a fixed set of species. In order to obtain the list of proteomes included in a phylome, use the **get_proteomes_in_phylome()** method. PhylomeDB uses its own sequence identifiers, but you can use the **search_id()** to find a match from an external sequence ID.

Example 33

```
from ete2 import PhylomeDBConnector
# This connects to the main phylomeDB server (default parameters)
p = PhylomeDBConnector()
# This connects to a local version of phylomeDB, and you can set the
# user and password arguments
p = PhylomeDBConnector(host="localhost", user="public", passwd="public", port=3306)
```

Each phylome is the collection of all trees associated to a given species. Thus, the human phylome will contain thousands of phylogenetic trees. Each gene/protein in a phylome may be associated to different trees, testing, for example, different evolutionary models. Thus when you query the database for a gene phylogeny you have to specify from which phylome and which specific tree. Alternatively, you can query for the best tree in a given phylomes, which will basically return the best likelihood tree for the queried gene/protein. The get_tree and get_best_tree methods carry out such operations. When trees are fetched from the phylomeDB database, the are automatically converted to the PhyloTree class, thus allowing to operate with them as phylogenetic trees.

Example 34

```
from ete2 import PhylomeDBConnector
# This connects to the main phylomeDB server (default parameters)
p = PhylomeDBConnector()
# Obtains the phylomeDB internal ID for my gene of interest
idmatches = p.search_id("ENSG00000146556")
# Take the only match (several would be possible)
geneid = idmatches[0]
# Gets the 'geneid' tree in phylome 1 reconstructed using WAG evolutionary model
t, likelihood = p.get_tree(geneid, "WAG", 1)
print t
#
#                         /-Xtr0044988
#                        |
#                        |     /-Gga0000980
#                        |    |
#                    /---|    |               /-Bta0018700
#                   |    |    |              |
#                   |    |    |              |                    /-Hsa0000001
#                   |    |    |              |               /---|
#                   |    |    |          /---|          /---|     \-Hsa0010733
#                   |     \---|         |    |         |    |
#                   |         |         |    |     /---|     \-Hsa0010710
#                   |         |         |    |    |    |
#                   |         |     /---|     \---|     \-Ptr0000001
#               /---|         |    |    |         |
#              |    |         |    |    |          \-Cfa0016699
#              |    |         |    |    |
#              |    |          \---|    |     /-Rno0030248
#              |    |              |     \---|
#              |    |              |          \-Mms0024821
#          /---|    |              |
#         |    |    |               \-Mdo0014718
#         |    |    |
#         |    |    |     /-Dre0008389
#     /---|    |     \---|
#    |    |    |          \-Fru0004507
#    |    |    |
#    |    |     \-Cin0011238
#----|    |
#    |     \-Aga0007658
#    |
#    |--Dme0014628
#    |
#     \-Ddi0002240
#
# Gets the best evolutionary model tree present in the phylome 1 for a given geneid
winner_model, all_likelihoods, t  = p.get_best_tree(geneid, 1)
# As you can see, the best likelihood tree in this case was
# reconstructed using a JTT model rather than the WAG matrix.
```

5.3 Going phylogenomic scale

Just to show you how to explore a complete phylome:

Example 35

```
from ete2 import PhylomeDBConnector
# This connects to the main phylomeDB server (default parameters)
p = PhylomeDBConnector()
PHYLOME_ID = 1
# This is the species code/age dictionary used to correctly root the
# tree in the human phylome. You can define your own, or use the
# midpoint outgroup method
species2age = {’Aga’: 8, ’Ago’: 9, ’Ame’: 8, ’Ath’: 10, ’Bta’: 3, ’Cal’: 9, ’Cbr’: 8,\
      ’Cel’: 8, ’Cfa’: 3, ’Cgl’: 9, ’Cin’: 7, ’Cne’: 9, ’Cre’: 10, ’Ddi’: 10, \
      ’Dha’: 9, ’Dme’: 8, ’Dre’: 6, ’Ecu’: 9, ’Fru’: 6, ’Gga’: 4, ’Gth’: 10,\
      ’Gze’: 9, ’Hsa’: 1, ’Kla’: 9, ’Lma’: 10, ’Mdo’: 3, ’Mms’: 3, ’Mmu’: 2,\
      ’Ncr’: 9, ’Pfa’: 10, ’Pte’: 10, ’Ptr’: 2, ’Pyo’: 10, ’Rno’: 3, ’Sce’: 9,\
      ’Spb’: 9, ’Tni’: 6, ’Xtr’: 5, ’Yli’: 9   }
# Iterator over each sequence in the human proteme
for i, seqid in enumerate(p.get_seed_ids(PHYLOME_ID)):
    if i>2: break # Just process the first 2 ids
    winner_model, lks, t = p.get_best_tree(seqid, PHYLOME_ID)
    # If tree was sucsesfully reconstructed, runs the species overalp algorithm
    if t and seqid in t:
        outgroup = t.get_farthest_oldest_leaf(species2age)
 # Returned outgroup is used to root the tree
 t.set_outgroup(outgroup)
 # Finds the node representing the seed sequence.
 # We want the orthology relationships of such sequence.
 seed_node = t.search_nodes(name=seqid)[0]
 evol_events = seed_node.get_my_evol_events()
 for ev in evol_events:
     # Speciation event
            if ev.etype == "S":
  inparalogs = filter(lambda n: n.startswith("Hsa"), ev.in_seqs)
```

```
print 'ORTHOLOGY RELATIONSHIP:', ','.join(inparalogs), "<===>", ','.join(ev.out_seqs)
```