Definition of Formats for Coding Phylogenetic Trees in R

Emmanuel Paradis

July 28, 2008

Contents

1 Introduction

This document explains how phylogenetic trees are coded and handled in the R package `ape`. Such data, like any R data, can be passed to C codes for computing-intensive tasks. How these data can be manipulated efficiently in R, and how they can be extended are also discussed. A companion to the present document is the application programmer interface (API) to `ape` C code [4].

Phylogenetic trees are complex data structures: coding them in computer programs requires special care. Felsenstein [2] outlines the coding used in most computer programs for phylogenetic analyses. Such data structures cannot be used easily in an interactive way because they are based on the recursive use of pointers. In addition, they cannot be extended or modified without recompiling the whole program. In R a different approach is used based on simple data structures like matrices and vectors. The class `"hclust"`, used by the function `hclust` of the package `stats`, codes hierarchical clusters with a two-column matrix indicating the pairings of observations. To code phylogenetic trees in R, it was necessary to extend and modify this class substantially.

The class `"phylo"` introduced in `ape` in August 2002 aims at this. Its initial definition in `ape` 0.1 has been useful in some applications. These and some feed-back from users have pointed out to some weaknesses and limitations of this initial definition. I detail below the new definition of the class `"phylo"` introduced in `ape` 1.9 released in November 2006.

2 Terminology and Notations

branch:	edge, vertex
node:	internal node
degree:	the number of edges that meet at a node
tip:	terminal node, leaf, node of degree 1
n:	number of tips
m:	number of nodes

3 Definition of the Class `"phylo"`

The class `"phylo"` is used to code "acyclical" phylogenetic trees. These trees have no reticulations, and all their internal nodes are of degree 3 or more, except the root (in the case of rooted trees) which is of degree 2 or more. An object of class `"phylo"` is a list with the following mandatory elements:

1. A numeric matrix named `edge` with two columns and as many rows as there are branches in the tree;

2. A character vector of length n named `tip.label` with the labels of the tips;

3. An integer value named `Nnode` giving the number of (internal) nodes;

4. An attribute `class` equal to `"phylo"`.

In the matrix `edge`, each branch is coded by the nodes it connects: tips are coded $1, \ldots, n$, and internal nodes are coded $n+1, \ldots, n+m$ ($n+1$ is the root). Both series are numbered without gaps.

The matrix `edge` has the following properties:

- The first column has only values greater than n (thus, values less than or equal to n appear only in the second column).

- All nodes appear in the first column at least twice.

- The number of occurrences of a node in the first column is related to the nature of the node: twice if it is dichotomous (i.e., of degree 3), three times if it is trichotomous (degree 4), and so on.
- All elements, except the root $n + 1$, appear once in the second column.

This representation is used for rooted and unrooted trees. For the latter, the position of the root is arbitrary.

The smallest tree of class "phylo" (a tree with a single branch) can be created in R with:

```
> tr <- list(edge = matrix(c(2, 1), 1, 2), tip.label = "a", Nnode = 1L)
> class(tr) <- "phylo"
> str(tr)
```

```
List of 3
 $ edge     : num [1, 1:2] 2 1
 $ tip.label: chr "a"
 $ Nnode    : int 1
 - attr(*, "class")= chr "phylo"
```

The following elements are optional:

1. A numeric vector named `edge.length` with the branch lengths: this has as many values of the number of rows of `edge`;
2. A character vector of length m named `node.label` with the labels of the nodes;
3. A single numeric value named `root.edge` giving the length of the branch at the root.

There is a correspondence between these elements and the structure of the mandatory ones (e.g., the length of the ith branch `edge[i, ]` is given by `edge.length[i]`).

There are several substantial changes compared to the definition used until version 1.8-5 of ape [3]: the most important one is that `edge` was previously of mode character with a different coding (tips had positive numbers "1", ..., "n", and nodes negative numbers "-1", ..., "-m"). Compatibily is provided by some functions in ape 1.9, and is still supported in ape 2.2-1 but it is planned to be removed soon.

Note that this new definition uses the S3 classes; for an implementation using S4 classes, see the package phylobase.[1]

3.1 Memory Requirements

The table below gives the sizes as given by `object.size()` with R 2.7.1 of three phylogenies distributed with ape and often used in examples, and two random trees generated with `rtree`.

[1] r-forge.r-project.org/projects/phylobase/

tree	n	bytes
bird.orders	23	2360
bird.families	135	11,000
chiroptera	917	64,168
random tree	1000	68,560
"	10,000	680,560

For big trees, the length of the tip labels will be critical for the overall quantity of memory used to store the object. `rtree` gives by default the labels `t1`, `t2`, ... If longer labels are used, this will obviously need more memory:

```
> tr <- rtree(1000)
> object.size(tr)

[1] 68560

> tr$tip.label <- paste("A_long_label_for_a_tree_with_1000_tips",
+     1:1000, sep = "-")
> object.size(tr)

[1] 108560
```

Interestingly, R manages repetitive elements in vector of mode character (like `tip.label`) in order to save memory, e.g.:

```
> tr$tip.label <- rep("A_long_label_for_a_tree_with_1000_tips",
+     1000)
> object.size(tr)

[1] 36632
```

This mechanism is also in action if only a subset of the character strings are repeated:

```
> tr$tip.label <- paste("A_long_label_for_a_tree_with_1000_tips",
+     1:1000, sep = "-")
> object.size(tr)

[1] 108560

> tr$tip.label[1:500] <- "Another_quite_long_label_for_a_tree"
> object.size(tr)

[1] 72632
```

3.2 Storage Mode

The matrix `edge` can be stored either as integers or as doubles; the latter requiring twice more memory than the former (the same applies to `Nnode` but the difference between both storage modes is only four bytes in all cases). `ape` used to have no requirement for the storage mode of this element, but since version 2.1 most functions in `ape` returns or manipulates trees with `edge` stored as integers. The gain in efficiency is substantial for the following reasons:

- The memory gain for a list of trees (which may be tens of thousands, e.g., an output from a Bayesian phylogenetic program) is potentially considerable.
- When passing the edge matrix to a C code, having it already as integers saves a significant amount of computing time.
- Overall, integers are faster to manipulate than doubles.

The distinction on storage mode has no consequence for end-users, and should be accomodated quite easily by developers. R extensively checks data types, so mixing integers and doubles is generally not a problem. However, in the case of ape some data are passed directly to some C code, and in some cases some care must be taken that the correct data type is used. Considerable attention has been paid to insure that no fatal error may occur in these situations; nevertheless, some functions may still need to be checked in this respect.

3.3 Guidelines for Creating the Matrix edge

In the matrix edge, each row represents a branch: the node in the first column is the origin of the branch, and the node or tip in the second column is its end. Note that for unrooted trees, this order is arbitrary (except for the terminal branches) because the position of the root is also arbitrary. This representation makes possible several generalizations, such as multichotomies (here $n = 3, m = 1$) ...

```
> matrix(c(4, 4, 4, 1:3), 3, 2)
```

```
     [,1] [,2]
[1,]    4    1
[2,]    4    2
[3,]    4    3
```

... or reticulations ($n = 4, m = 3$):

```
> cbind(c(5, 6, 6, 6, 5, 7, 7), c(6, 1, 2, 7, 7, 3, 4))
```

```
     [,1] [,2]
[1,]    5    6
[2,]    6    1
[3,]    6    2
[4,]    6    7
[5,]    5    7
[6,]    7    3
[7,]    7    4
```

There is an arbitrary direction ($6 \rightarrow 7$), from the smallest number to the largest one. At the moment, it is still an open question whether these reticulations should be coded in the matrix edge, or should rather be stored in a distinct matrix to avoid confusion. As mentioned above, reticulations are not supported in the class "phylo".

There is no mandatory order for the rows of edge, but they may be arranged in a way that is efficient for computation and manipulation. For instance, consider the tree in Newick format:

```
"((,),(,));"
```

Then the two following matrices are similar for `edge`:

```
> cbind(c(5, 6, 6, 5, 7, 7), c(6, 1, 2, 7, 3, 4))

     [,1] [,2]
[1,]    5    6
[2,]    6    1
[3,]    6    2
[4,]    5    7
[5,]    7    3
[6,]    7    4
```

```
> cbind(c(5, 5, 6, 6, 7, 7), c(6, 7, 1, 2, 3, 4))

     [,1] [,2]
[1,]    5    6
[2,]    5    7
[3,]    6    1
[4,]    6    2
[5,]    7    3
[6,]    7    4
```

In the first representation the branches are grouped *cladewise*, whereas in the second one the internal branches come first. Any order of the rows are valid with respect to the above definition. However, the cladewise order has an interesting feature: it is straightforward to find all the branches descendant of a given node (see § 4.2).

There is another interesting order:

```
> cbind(c(6, 6, 7, 7, 5, 5), c(1, 2, 3, 4, 6, 7))

     [,1] [,2]
[1,]    6    1
[2,]    6    2
[3,]    7    3
[4,]    7    4
[5,]    5    6
[6,]    5    7
```

Here, the branches are arranged so that a "pruning" calculation can be done by reading down the rows of `edge`. Additionally, if some conventions are taken, this arrangement can lead to a unique representation for a given tree in the same way than the matchings proposed by Diaconis & Holmes [1].[2] I shall call the above order *pruningwise*.

[2] Matchings work only for rooted dichotomous trees.

4 Tree Manipulation

The table below shows how to perform a few basic operations on objects of class `"phylo"` in R.

Operation	High level	Low level
How many tips?	`Ntip(tr)`	`length(tr$tip.label)`
How many nodes?	`Nnode(tr)`	`tr$Nnode`
How many branches?	`Nedge(tr)`	`dim(tr$edge)[1]`
How to find node `x`?		`which(tr$edge == x)` *or* `which(tr$edge == x, TRUE)`

What is the ancestor of node `x`?		`i <- which(tr$edge[, 2] == x)`
		`tr$edge[i, 1]`

What are the terminal branches?		`n <- Ntip(tr)`
		`which(tr$edge[, 2] >= n)`

The high level functions (introduced in `ape` 1.10-1) are easier to remember but also check for data type as can be seen in the code of `Ntip`:

```
> Ntip

function (phy)
{
    if (class(phy) != "phylo")
        stop("object \"phy\" is not of class \"phylo\"")
    length(phy$tip.label)
}
```

The other operations may also be wrapped in a function, for instance:

```
getAncestor <- function(phy, x)
{
    if (x == phy$Nnode + 1)
        stop("node 'x' is the root")
    i <- which(phy$edge[, 2] == x)
    phy$edge[i, 1]
}
```

4.1 Preorder Tree Traversal

Preorder tree traversal means here: travelling through the tree from the root to the tips. If an object of class `"phylo"` is in cladewise order, then the first element of the first column of `edge` is, by definition, the root and numbered $n + 1$. This is true whether the tree is rooted or not (remind that the root is arbitrary in the latter case). The beginning and end of each clade connected to the root is found with:

```
start <- which(tr$edge[, 1] == length(tr$tip.label) + 1)
end <- c(start[-1] - 1, dim(tr$edge)[1])
```

The MRCA of these clades (i.e., the direct descendants of the root) can be found with:

```
tr$edge[start, 2]
```

As an example, we take the avian families tree:

```
> data(bird.families)
> n <- length(bird.families$tip.label)
> start <- which(bird.families$edge[, 1] == n + 1)
> end <- c(start[-1] - 1, dim(bird.families$edge)[1])
> start
```

```
[1]  1 28
```

```
> end
```

```
[1]  27 271
```

The two nodes connected to the root are:

```
> bird.families$edge[start, 2]
```

```
[1] 139 152
```

This can be checked graphically with:

```
plot(bird.families)
nodelabels()
```

This approach can then be applied repeatedly from the root to the tips. For instance, we find that the first node descendant of the root is `tr$edge[start[1], 2]`: we may thus replace '`n + 1`' above by this value, and '`dim(tr$edge)[1]`' by '`end[1]`':

```
startB <- which(tr$edge[, 1] == tr$edge[start[1], 2])
endB <- c(startB[-1] - 1, end[1])
```

Note the new names `startB` and `endB`; in practice, this may be simplified by using recursive calls to a function (see § 4.6).

If the object `tr` is in pruningwise order, then tree traversal may be done through successive search of node and tips numbers using `which` (as was done by most functions in `ape`). However, this is less efficient.

4.2 Finding a Clade

For an arbitrary node, say `nod`, the approach above may be adapted to the following algorithm:

1. Find the node ancestor of `nod`, store its number in `anc`, and store the number of the branch in `i`.
2. Find the next occurrence of `anc` in `tr$edge[, 1]`, store it in `j`.

The clade descending from `nod` is given by the rows $i+1$ to $j-1$ of `tr$edge`. This algorithm coded in R is:

```
i <- which(tr$edge[, 2] == nod)
anc <- tr$edge[i, 1]
tmp <- which(tr$edge[, 1] == anc)
j <- tmp[which(tmp == i) + 1]
tr$edge[(i+1):(j-1), ]
```

Note that it is straightforward to translate this code in C.

4.3 Postorder Tree Traversal

Postorder tree traversal means here: travelling through the tree from the tips to the root. If the object `tr` is in pruningwise order, then postorder tree traversal is straightforward by descending along the rows of `tr$edge`. Otherwise, successive searches must be done.

Gabriel Valiente pointed out to me that the pruningwise order in `ape` should be called *bottom-up* order rather than postorder [5].

4.4 Relating Nodes and Tips to Other Data

Because tips and nodes are numbered sequentially from 1 to $n+m$, it is straightforward to associate them with other data. For instance, if we have n values of body mass for the tips of the tree stored in the vector `body.mass`, then the value for the ith tip would be `body.mass[i]`, and the corresponding label is `tr$tip.label[i]`. If instead we have a data frame `DF` with several such variables as columns (e.g., body mass, fecundity, ...), we would relate the ith tip with the row `DF[i, ]`.

If the elements of `body.mass` or the rows of `DF` are not in the same order than the tip labels, then the names or rownames of the former may be used, but this is less efficient than numeric indexing. Depending on the context of the analysis, it may be better to reorder the data first:

```
body.mass <- body.mass[tr$tip.label]
DF <- DF[tr$tip.label, ]
```

and then use numeric indexing.

If some data are available for the tips and the nodes, they can be stored in a vector of length $n+m$ where the elements 1 to n will be the values for the tips, and those $n+1$ to m will be for the nodes (e.g., the ancestral values of body mass). If values are available only for nodes (e.g., bootstrap values) they can be stored in a vector of length m, and numeric indexing may be used by offsetting with $-n$, i.e., for the ith node `x[i - n]` (for the root $i = n+1$, this would be `x[1]`).

4.5 Tracking Nodes

The problem of tracking a node through successive tree manipulation is not easy because nodes are numbered sequentially. For instance, if two trees are binded, the node numbers of one of them must be changed. The solution to this problem is to use the element `node.label`. Node labels may be created easily with:

```
tr$node.label <- paste("node", 1:tr$Nnode, sep = "")
```

If a second tree, say `trb`, is involved here:

```
trb$node.label <- paste("trb_node", 1:trb$Nnode, sep = "")
```

Both trees may be binded now.

```
trc <- bind.tree(tr, trb)
```

To find the node number of say `"trb_node1"` in the new tree:

```
which(trc$node.label == trb_node1)
```

4.6 Example

As an example of using the new definition of the class `"phylo"` and the guidelines above, I rewrote the function `rtree`. The logic of the algorithm is the same in both versions: splitting repeatedly a pool of n tips until splitting cannot be done. But the implementations in R are different. In the old version, the branches were defined successively and the matrix `edge` was filled along its rows; the latter were reordered at the end. In the new version, the rows are filled with respect to the size of the clades using a recursive function, so that no reordering is needed at the end. The following table compares the timings of the two versions.[3]

n	old	new	gain
10	0.026	0.0006	43
100	0.290	0.003	100
1000	4.017	0.030	134

This shows that considerable improvement can be achieved even using R codes only. Furthermore, the algorithm is now more appropriate to be coded in C, so that further improvement seems possible. Note that the operation done by `rtree` is actually a preorder tree traversal: thus, this operation can be done in a very short time with R code.

5 Uniqueness of Representation

At the moment, this remains unsolved bacause even adopting one of the rules described above for the order of the branches may lead to several representations of the same tree. Additional rules are needed. Some open questions:

- Is there an order that makes tree manipulation optimal? (see above for some partial answers)
- Do we really need a standard, unique order?

Keep in mind that the rules should work for rooted and unrooted, dichotomous and multichotomous trees (and possibly reticulograms but this is set aside for the moment).

[3] Timings averaged over 100 consecutive repetitions with a `for` loop, processor at 1.86 GHz, 2 Gb RAM, Linux Knoppix 4.0, R 2.3.0, `ape` 1.8-3.

6 Passing Trees from R to C

Because the class `"phylo"` uses only vectors,[4] it is easy to pass these data to C using the R function `.C`. For instance, the matrix `edge` may be passed with:[5]

```
.C("nameofCfunction", as.integer(tr$edge),
   dim(tr$edge)[1], PACKAGE = "nameofpackage")
```

which will be received in C with:

```
void nameofCfunction(int * edge, int * N)
```

where `edge` is a pointer to a C array of size $2N$. The two nodes of the ith branch will be accessed with `edge[i - 1]` and `edge[i - 1 + N]`. An alternative is to pass separately the two columns of `edge`:

```
.C("nameofCfunction", as.integer(tr$edge[, 1]),
   as.integer(tr$edge[, 2]), dim(tr$edge)[1],
   PACKAGE = "nameofpackage")
```

with in the C program:

```
void nameofCfunction(int * edge1, int * edge2, int * N)
```

Now the two nodes will be accessed with `edge1[i - 1]` and `edge2[i - 1]`. A complete tree structure may be passed with `.C`:

```
.C("nameofCfunction", as.integer(tr$edge), as.integer(tr$Nnode),
   dim(tr$edge)[1], as.double(tr$edge.length),
   as.character(tr$tip.label), as.character(tr$node.label),
   PACKAGE = "nameofpackage")
```

received in C with:

```
void nameofCfunction(int * edge, int * nnode,  int * N,
                     double * edge_length, char ** tip_label,
                     char ** node_label)
```

Note that other elements may be added in the arguments as long as they are R vectors. The advantage of using `.C` is that data manipulation in C is similar to any program in this language: the only constraint is that the function receiving the data from R must have pointers and return void.

Using `.Call` or `.External` is more flexible:

```
.Call("nameofCfunction", tr, PACKAGE = "nameofpackage")
.External("nameofCfunction", tr, PACKAGE = "nameofpackage")
```

where `tr` may be any R data. But the data manipulation in C is more complex since it deals with SEXP (*S expression*) structures:

```
SEXP nameofCfunction(SEXP tr)
```

This is advantageous if the number and/or size of elements is unknown in advance. An example can be found in the sources of `ape` (see src/bipartition.c and R/dist.topo.R, or src/tree_build.c and R/read.nexus.R).

[4] Matrices in R are actually vectors.

[5] Coercion to integers is not absolutely necessary if we are sure that the data are integers which is the case with the values returned by `dim` and `length`.

7 Others

It is also possible to add further attributes to an object of class `"phylo"` such as the number of tips (n), … also `rooted` (TRUE or FALSE), `order` (`"cladewise"` or `"pruningwise"`),[6] and so on. This may use `attr` or `$`.

In fact, any element may be appended in an object of class `"phylo"` like any R object. Adding some information may be useful to store the results of previous computations (as done with the class `"phylog"` in `ade4`).

8 Definition of the Class `"multiPhylo"`

An object of class `"multiPhylo"` is a list of one or several trees each of class `"phylo"`. It has two possible configurations:

1. All individual trees follow the definition in § 3, and the resulting list has the class `"multiPhylo"`.
2. The individual trees do not have an element `tip.label`, but otherwise follow the definition in § 3, and the resulting list has the class `"multiPhylo"` and an attribute `TipLabel`.

The second configuration implies that the ith tip in all trees have the same label. This is useful when reading a list of trees read from a NEXUS file with a TRANSLATE block: here the labels in the Newick are substituted by tokens 1, 2, …, n. `ape` takes directly these tokens and fills the matrix `edge` with them, and the object of class `"multiPhylo"` returned has the attribute `TipLabel` taken from the translation table of the NEXUS file.

Other functions, such as `rmtree` or `read.tree`, return a list with the first configuration. It is possible to switch to the second configuration with the internal function `.compressTipLabel`: it will eventually renumber the elements of `edge` in each tree and delete the element `tip.label`. An error occurs if the tip labels do not match among trees, or if some are duplicated within a tree. The size of the object in memory is approximately halved:

```
> TR <- rmtree(100, 100)
> object.size(TR)
```

```
[1] 736552
```

```
> object.size(.compressTipLabel(TR))
```

```
[1] 373832
```

There are extraction and subsetting methods for the class `"multiPhylo"` using `$`, `[`, and `[[` which set correctly the class and the elements of the returned object:

```
> get("[[.multiPhylo")
```

[6] This one is used experimentally by some functions in `ape`.

```
function (x, i)
{
    class(x) <- NULL
    phy <- x[[i]]
    if (!is.null(attr(x, "TipLabel")))
        phy$tip.label <- attr(x, "TipLabel")
    phy
}
```

Manipulation of objects of class `"multiPhylo"` thus follows R's standard operations on lists. Note that such an object may contain only one tree in which case it will eventually be extracted with `TR[[1]]`.

Lists of trees can be passed to C code just like trees of class `"phylo"` (§ 6), but here it is better to use the `.Call` or `.External` interface since the number of trees may be variable (for an example see the C function `prop_part` in src/bipartition.c).

9 References

[1] Diaconis P. W. & Holmes S. P. 1998. Matchings and phylogenetic trees. *Proceedings of the National Academy of Sciences USA* **95**: 14600–14602.

[2] Felsenstein J. 2004. *Inferring Phylogenies.* Sinauer Associates, Sunderland, MA.

[3] Paradis E. 2006. *Analysis of Phylogenetics and Evolution with R.* Springer, New York.

[4] Paradis E., Desper R., Gascuel O., Lefort V. & Cuong H. S. 2008. The API to `ape` C code. `http://ape.mpl.ird.fr/misc/API_ape_C_code-26March2008.pdf`.

[5] Valiente G. 2002. *Algorithms on Trees and Graphs.* Springer, New York.