Definition of Formats for Coding Phylogenetic Trees in R

Emmanuel Paradis

22 December 2020

Contents

---

1 Introduction

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 or 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.

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 code for computing-intensive tasks. How these data can be manipulated efficiently in R, and how they can be extended are also briefly discussed. Section 3 gives some background on how this class has changed over time and can be skipped if you are interested only in the current implementation.

2 Terminology and Notations

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

3 History

The general structure of the class `"phylo"` has been stable since the first release of `ape`: the basic idea of a list with a matrix named `edge` coding the tree topology and additional vectors (`edge.length`, `tip.label`, ...) has not changed since `ape` 0.1. The important changes that happened over the years concern the details of the `edge` matrix. Initially, this matrix was of mode character: the tips were coded `"1"`, `"2"`, ..., `"n"` and the nodes were coded with `"-1"`, `"-2"`, ..., `"-m"`. This scheme is described in details in the first edition of *APER* [3]. It works well with small trees, but is rather inefficient for large ones (each character string in the matrix requires its own pointer resulting in large memory requirements for big trees).

In 2006, it was envisaged to change the mode of the `edge` matrix to numeric while keeping positive numbers for tips and negatives for nodes (see the version of this document dated June 14, 2006). However, this was never implemented in `ape`. Instead, the scheme detailed in this document was introduced in `ape` 1.9 released in November 2006. At this time, it was not decided whether to store the values in the `edge` matrix as doubles (each value requiring 8 bytes) or as integers (4 bytes).[1] Since `ape` 4.0 (November 2016), all functions in the package insure that the `edge` matrix has storage mode integer, although most functions work correctly if the storage mode is double (see below).

The class `"phylo"` was extended in `ape` 2.7-2 (June 2011) with the class `"evonet"` to support trees with reticulations (or evolutionary networks). Starting from `ape` 5.0 (October 2017), trees with nodes of degree 2 (singleton or "elbow" nodes) can be handled in the class `"phylo"`.

With respect to tree file input/output, reading Newick files was supported with the first version of `ape` using the function `read.tree`, and writing Newick files (`write.tree`) was introduced in version 0.2 (September 2002). Reading and writing NEXUS files (`read.nexus` and `write.nexus`) were introduced in `ape` 1.0 (February 2003). These four functions have been gradually improved over the years and are still the main functions for input/output of the class `"phylo"`. The functions `read.evonet` and `write.evonet` were introduced in `ape` 5.0 to read and write files in Newick extended format.

4 Definition of the Class `"phylo"`

The class `"phylo"` is used to code standard phylogenetic trees with no reticulations, and all internal nodes are of degree 2 or more. An object of class `"phylo"` is a list with the following mandatory elements:

1. A matrix of integers 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. A single integer value named `Nnode` giving the number of (internal) nodes;
4. An attribute `class` equal to `"phylo"`.

[1] Both data types are coded in R with the mode numeric.

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. This matrix 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 once.
- 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. A node can be only once if it is a singleton node.
- All nodes, 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:

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

Phylogenetic tree with 1 tips and 1 internal nodes.

Tip labels:
  a

Rooted; no branch lengths.


> str(tr)

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


> write.tree(tr)

[1] "(a);"
```

Next is an example with a singleton node ($n = 1$, $m = 2$):

```
> tr <- list(edge = matrix(c(2L, 3L, 3L, 1L), 2, 2),
+            tip.label = "a", Nnode = 2L)
> class(tr) <- "phylo"
> tr
Phylogenetic tree with 1 tips and 2 internal nodes.

Tip labels:
  a

Rooted; no branch lengths.

> str(tr)
```

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


> write.tree(tr)
[1] "((a));"
```

The following elements are optional:

1. A numeric vector named `edge.length` with the branch lengths: the length of this vector is equal to 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.
4. An attribute `"order"` which can take the values `"cladewise"`, `"pruningwise"`, or `"postorder"`.

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

4.1 Memory Requirements

The code below shows the memory used by three phylogenetic trees provided as data in `ape` and two random trees with $n = 1000$ and $n = 10{,}000$:

```
> data(bird.orders)
> data(bird.families)
> data(chiroptera)
> object.size(bird.orders)
3480 bytes


> object.size(bird.families)
15344 bytes


> object.size(chiroptera)
90280 bytes


> object.size(rtree(1000))
97304 bytes


> object.size(rtree(1e4))
961304 bytes
```

The matrix `edge` occupies a fixed quantity of memory which depends only on the number of edges: this number multiplied by two (the number of columns of `edge`) and multiplied by

four (the number of bytes used to store an integer) gives the quantity of memory needed to store `edge`. Since there are $2n - 2$ branches in a binary rooted tree (as generated by `rtree`, we can have a very good approximation with $(2n - 2) \times 8$:

```
> n <- 1e4
> tr <- rtree(n)
> (2 * n - 2) * 8
[1] 159984


> object.size(tr$edge)
160200 bytes
```

The branch lengths each occupy eight bytes since they are stored as doubles, so they are expected to use as much memory as the previous element:

```
> object.size(tr$edge.length)
160032 bytes
```

The largest part of the rest of the memory requirements is occupied by the tip labels:

```
> object.size(tr$tip.label)
640048 bytes
```

For big trees, the length of the tip labels is 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)
97304 bytes


> tr$tip.label <- paste("A_long_label_for_a_tree_with_1000_tips",
+                       1:1000, sep = "-")
> object.size(tr)
137304 bytes
```

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)
41400 bytes
```

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)
137304 bytes
```

```
> tr$tip.label[1:500] <- "Another_quite_long_label_for_a_tree"
> object.size(tr)
89400 bytes
```

4.2 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 (this is enforced more strictly since `ape` 4.0; see Sect. 3). 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.

```
> checkValidPhylo(tr)
Starting checking the validity of tr...
Found number of tips: n = 1000
Found number of nodes: m = 999
Done.


> storage.mode(tr$edge)
[1] "integer"


> storage.mode(tr$edge) <- "double"
> checkValidPhylo(tr)
Starting checking the validity of tr...
Found number of tips: n = 1000
Found number of nodes: m = 999
  MODERATE: the matrix 'edge' is not stored as integers
Done.
```

4.3 Limits on Tree Size

There are two potential limits on the size of a tree coded with the class `"phylo"`:

- In R, a matrix cannot have more than $2^{31} - 1$ rows (and columns as well but the `edge` matrix has only two columns) which puts an upper limit on the number of edges.

- This value is also the largest possible value for an integer in R (printed with `.Machine$integer.max`).

The second limit applies to the total number of internal and terminal nodes:

$$n + m \leq 2^{31} - 1$$

For any kind of tree, the number of edges is equal to $n + m - 1$, so both limits are very close although the second one will be reached first.

For fully binary trees, we have $m = n - 1$ for rooted trees, or $m = n - 2$ for unrooted trees; so, the number of edges are $2n - 2$ or $2n - 3$, respectively. In both cases, the largest possible value of n is therefore $2^{30} = 1{,}073{,}741{,}824$. The `edge` matrix of this tree would require a bit more than 17 GB of memory:

```
> (2 * (2^30) - 2) * 8
[1] 17179869168
```

4.4 Guidelines for Creating the Matrix `edge`

In practical situations, trees of class `"phylo"` are created in R by reading Newick or NEXUS files, by simulating random trees, or by recontructing trees from data (`nj()`, ...) The present section aims to introduce some ideas used in the following sections.

In the `edge` matrix, 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
```

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 Sect. 5.2).

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] This order is called *pruningwise* in ape.

5 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. They have been made generic in ape 4.0:

```
> Ntip

function (phy)
UseMethod("Ntip")
```

[2] Matchings work only for rooted dichotomous trees.

```
<bytecode: 0x55d1d9adfb50>
<environment: namespace:ape>


> Ntip.phylo
function (phy)
length(phy$tip.label)
<bytecode: 0x55d1daab9840>
<environment: namespace:ape>
```

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

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

5.1 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.

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.

5.2 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.

5.3 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 [4]. The postorder ordering has been introduced in `ape` 3.0-6 and is now used by most functions of the package.

5.4 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., posterior 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]`).

5.5 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 with:[3]

```
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)
```

5.6 Others

There are a number of functions in `ape` to manipulate trees: it would be too long to detail them here. More information can be found in `ape`'s manual or, equivalently, in the help pages of the package. Further information may also be found on `ape`'s web site:

`ape-package.ird.fr`

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)
```

[3] The function `makeNodeLabel` gives several possibilities to create similar labels.

[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`.

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. Furthermore, R objects are not duplicated with `.Call` whereas they are always duplicated in memory when using `.C`,[6] so the former is more efficient than the latter when handling a lot of data. Some examples 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).

7 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 Section 4, and the resulting list has the class `"multiPhylo"`.

[6] This was optional before R 3.2.0.

2. The individual trees do not have an element `tip.label`, but otherwise follow the definition in Section 4, 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 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)
1091480 bytes

> object.size(.compressTipLabel(TR))
446840 bytes
```

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

```
> ape:::`[[.multiPhylo`
function (x, i)
{
    class(x) <- NULL
    phy <- x[[i]]
    if (!is.null(attr(x, "TipLabel")))
        phy$tip.label <- attr(x, "TipLabel")
    phy
}
<bytecode: 0x55d1daa37b10>
<environment: namespace:ape>
```

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"` (Sect. 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).

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