Package ‘castor’

September 27, 2017

Type Package

Title Efficient Phylogenetics on Large Trees

Version 1.2.2

Date 2017-09-30

Author Stilianos Louca

Maintainer Stilianos Louca <`louca@zoology.ubc.ca`>

Description Efficient tree manipulation functions including pruning, rerooting, calculation of most-recent common ancestors, calculating distances from the tree root and calculating pairwise distance matrices. Calculation of phylogenetic signal and mean trait depth (trait conservatism). Ancestral state reconstruction and hidden character prediction of discrete characters, using Maximum Likelihood and Maximum Parsimony methods. Simulating and fitting models of trait evolution, and generating random trees using birth-death models.

License GPL (>= 2)

Depends Rcpp (>= 0.12.10)

Imports parallel, naturalsort, stats

LinkingTo Rcpp

NeedsCompilation yes

Repository CRAN

Date/Publication 2017-09-27 21:49:16 UTC

R topics documented:

`castor-package` *Efficient computations on large phylogenetic trees.*

Description

This package provides efficient tree manipulation functions including pruning, rerooting, calculation of most-recent common ancestors, calculating distances from the tree root and calculating pairwise distance matrices. Calculation of phylogenetic signal and mean trait depth (trait conservatism). Efficient ancestral state reconstruction and hidden character prediction of discrete characters on phylogenetic trees, using Maximum Likelihood and Maximum Parsimony methods. Simulating models of trait evolution, and generating random trees.

Details

The most important data unit is a phylogenetic tree of class "phylo", with the tree topology encoded in the member variable `tree.edge`. See the ape package manual for details on the "phylo" format. The castor package was designed to be efficient for large phylogenetic trees (>10,000 tips), and scales well to trees with millions of tips. Most functions have asymptotically linear time complexity O(N) in the number of edges N. This efficiency is achived via temporary auxiliary data structures, use of dynamic programing, heavy use of C++, and integer-based indexing instead of name-based indexing of arrays. All functions support trees that include monofurcations (nodes with a single child) as well as multifurcations (nodes with more than 2 children). See the associated paper by Louca et al. for a comparison with other packages.

Throughout this manual, "Ntips" refers to the number of tips, "Nnodes" to the number of nodes and "Nedges" to the number of edges in a tree. In the context of discrete trait evolution/reconstruction, "Nstates" refers to the number of possible states of the trait. In the context of multivariate trait evolution, "Ntraits" refers to the number of traits.

Author(s)

Stilianos Louca

Maintainer: Stilianos Louca <louca@zoology.ubc.ca>

References

S. Louca, L. W. Parfrey and M. Doebeli (in review). Efficient computations on large phylogenetic trees.

`asr_empirical_probabilities`

Empirical ancestral state probabilities.

Description

Given a rooted phylogenetic tree and the states of a discrete trait for each tip, calculate the empirical state frequencies/probabilities for each node in the tree, i.e. the frequencies/probabilities of states across all tips descending from that node. This may be used as a very crude estimate of ancestral state probabilities.

Usage

```
asr_empirical_probabilities(tree, tip_states, Nstates=NULL,
                            probabilities=TRUE, check_input=TRUE)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`tip_states`	An integer vector of size Ntips, specifying the state of each tip in the tree as an integer from 1 to Nstates, where Nstates is the possible number of states (see below).
`Nstates`	Either `NULL`, or an integer specifying the number of possible states of the trait. If `NULL`, then it will be computed based on the maximum value encountered in `tip_states`
`probabilities`	Logical, specifying whether empirical frequencies should be normalized to represent probabilities. If `FALSE`, then the raw occurrence counts are returned.
`check_input`	Logical, specifying whether to perform some basic checks on the validity of the input data. If you are certain that your input data are valid, you can set this to `FALSE` to reduce computation.

Details

For this function, the trait's states must be represented by integers within 1,..,Nstates, where Nstates is the total number of possible states. If the states are originally in some other format (e.g., characters or factors), you should map them to a set of integers 1,..,Nstates. You can easily map any set of discrete states to integers using the function `map_to_state_space`.

The tree may include multi-furcations (i.e. nodes with more than 2 children) as well as mono-furcations (i.e. nodes with only one child). The function has asymptotic time complexity O(Nedges x Nstates).

Tips must be represented in `tip_states` in the same order as in `tree$tip.label`. The vector `tip_states` need not include names; if it does, however, they are checked for consistency (if `check_input==TRUE`).

Value

A list with the following elements:

`ancestral_likelihoods`	A 2D integer (if `probabilities==FALSE`) or numeric (if `probabilities==TRUE`) matrix, listing the frequency or probability of each state for each node. This matrix will have size Nnodes x Nstates, where Nstates was either explicitly provided as an argument or inferred from `tip_states`. The rows in this matrix will be in the order in which nodes are indexed in the tree, i.e. the [n,s]-th entry will be the frequency or probability of the s-th state for the n-th node. Note that the name was chosen for compatibility with other ASR functions.

Author(s)

Stilianos Louca

See Also

`asr_max_parsimony`, `asr_squared_change_parsimony` `asr_mk_model`, `map_to_state_space`

Examples

```
## Not run:
asr_empirical_probabilities(tree, tip_states=c(1,3,2,3), Nstates=3)

## End(Not run)
```

`asr_independent_contrasts`

Ancestral state reconstruction via phylogenetic independent contrasts.

Description

Reconstruct ancestral states for a continuous (numeric) trait using phylogenetic independent contrasts (PIC; Felsenstein, 1985).

Usage

```
asr_independent_contrasts(tree, tip_states,  weighted=TRUE, check_input=TRUE)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`tip_states`	A numeric vector of size Ntips, specifying the known state of each tip in the tree.

`weighted`	Logical, specifying whether to weight tips and nodes by the inverse length of their incoming edge, as in the original method by Felsenstein (1985). If `FALSE`, edge lengths are treated as if they were 1.
`check_input`	Logical, specifying whether to perform some basic checks on the validity of the input data. If you are certain that your input data are valid, you can set this to `FALSE` to reduce computation.

Details

The function traverses the tree in postorder (tips–>root) and estimates the state of each node as a convex combination of the estimated states of its chilren. These estimates are the intermediate "X" variables introduced by Felsenstein (1985) in his phylogenetic independent contrasts method. For the root, this yields the same globally parsimonious state as the squared-changes parsimony algorithm implemented in `asr_squared_change_parsimony` (Maddison 1991). For any other node, PIC only yields locally parsimonious reconstructions, i.e. reconstructed states only depend on the subtree descending from the node (see discussion by Maddison 1991).

If `weighted==TRUE`, then this function yields the same ancestral state reconstructions as

`ape::ace(phy=tree, x=tip_states, type="continuous", method="pic", model="BM", CI=FALSE)`

in the `ape` package (v. 0.5-64).

If `tree$edge.length` is missing, each edge in the tree is assumed to have length 1. This is the same as setting `weighted=FALSE`. The tree may include multi-furcations (i.e. nodes with more than 2 children) as well as mono-furcations (i.e. nodes with only one child). Edges with length 0 will be adjusted internally to some tiny length if needed (if `weighted==TRUE`).

Tips must be represented in `tip_states` in the same order as in `tree$tip.label`. The vector `tip_states` need not include item names; if it does, however, they are checked for consistency (if `check_input==TRUE`). All tip states must be non-NA; otherwise, consider using one of the functions for hidden-state-prediction (e.g., `hsp_independent_contrasts`).

The function has asymptotic time complexity O(Nedges).

Value

A list with the following elements:

`ancestral_states`	A numeric vector of size Nnodes, listing the reconstructed state of each node. The entries in this vector will be in the order in which nodes are indexed in the tree.
`total_sum_of_squared_changes`	The total sum of squared changes in tree, minimized by the (optionally weighted) squared-change parsimony algorithm. This is equation 7 in (Maddison, 1991).

Author(s)

Stilianos Louca

References

J. Felsenstein (1985). Phylogenies and the Comparative Method. The American Naturalist. 125:1-15.

W. P. Maddison (1991). Squared-change parsimony reconstructions of ancestral states for continuous-valued characters on a phylogenetic tree. Systematic Zoology. 40:304-314.

See Also

`asr_squared_change_parsimony`, `asr_max_parsimony`, `asr_mk_model`

Examples

```
# generate random tree
Ntips = 100
tree = generate_random_tree(list(birth_rate_intercept=1),max_tips=Ntips)$tree

# simulate a continuous trait
tip_states = simulate_ou_model(tree, stationary_mean=0, spread=1, decay_rate=0.001)$tip_states

# reconstruct node states via weighted PIC
node_states = asr_independent_contrasts(tree, tip_states, weighted=TRUE)$ancestral_states
```

`asr_max_parsimony` *Maximum-parsimony ancestral state reconstruction.*

Description

Reconstruct ancestral states for a discrete trait using maximum parsimony. Transition costs can vary between transitions, and can optionally be weighted by edge length.

Usage

```
asr_max_parsimony(tree, tip_states, Nstates=NULL,
                  transition_costs="all_equal",
                  edge_exponent=0, weight_by_scenarios=TRUE,
                  check_input=TRUE)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`tip_states`	An integer vector of size Ntips, specifying the state of each tip in the tree as an integer from 1 to Nstates, where Nstates is the possible number of states (see below).
`Nstates`	Either `NULL`, or an integer specifying the number of possible states of the trait. If `NULL`, then Nstates will be computed based on the maximum value encountered in `tip_states`

`transition_costs`
: Either "all_equal","sequential", "proportional", "exponential", or a quadratic non-negatively valued matrix of size Nstates x Nstates, specifying the transition costs between all possible states (which can include 0 as well as `Inf`). The [r,c]-th entry of the matrix is the cost of transitioning from state r to state c. The option "all_equal" specifies that all transitions are permitted and are equally costly. "sequential" means that only transitions between adjacent states are permitted and are all equally costly. "proportional" means that all transitions are permitted, but the cost increases proportional to the distance between states. "exponential" means that all transitions are permitted, but the cost increases exponentially with the distance between states. The options "sequential" and "proportional" only make sense if states exhibit an order relation (as reflected in their integer representation).

`edge_exponent`
: Non-negative real-valued number. Optional exponent for weighting transition costs by the inverse length of edge lengths. If 0, edge lengths do not influence the ancestral state reconstruction (this is the conventional max-parsimony). If >0, then at each edge the transition costs are multiplied by $1/L^e$, where L is the edge length and e is the edge exponent. This parameter is mostly experimental; modify at your own discretion.

`weight_by_scenarios`
: Logical, indicating whether to weight each optimal state of a node by the number of optimal maximum-parsimony scenarios in which the node is in that state. If FALSE, then all optimal states of a node are weighted equally (i.e. are assigned equal probabilities).

`check_input`
: Logical, specifying whether to perform some basic checks on the validity of the input data. If you are certain that your input data are valid, you can set this to `FALSE` to reduce computation.

Details

For this function, the trait's states must be represented by integers within 1,..,Nstates, where Nstates is the total number of possible states. If the states are originally in some other format (e.g. characters or factors), you should map them to a set of integers 1,..,Nstates. The order of states (if relevant) should be reflected in their integer representation. For example, if your original states are "small", "medium" and "large" and `transition_costs=="sequential"`, it is advised to represent these states as integers 1,2,3. You can easily map any set of discrete states to integers using the function `map_to_state_space`.

This function utilizes Sankoff's (1975) dynamic programming algorithm for determining the smallest number (or least costly if transition costs are uneven) of state changes along edges needed to reproduce the observed tip states. The function has asymptotic time complexity O(Ntips+Nnodes x Nstates).

If `tree$edge.length` is missing, each edge in the tree is assumed to have length 1. If `edge_exponent` is 0, then edge lengths do not influence the result. If `edge_exponent!=0`, then all edges must have non-zero length. The tree may include multi-furcations (i.e. nodes with more than 2 children) as well as mono-furcations (i.e. nodes with only one child).

Tips must be represented in `tip_states` in the same order as in `tree$tip.label`. None of the input vectors or matrixes need include row or column names; if they do, however, they are checked for consistency (if `check_input==TRUE`).

This function is meant for reconstructing ancestral states in all nodes of a tree, when the state of each tip is known. If some of the tips have unknown state, consider either pruning the tree to keep only tips with known states, or using the function `hsp_max_parsimony`.

Value

A list with the following elements:

`ancestral_likelihoods`
: A 2D numeric matrix, listing the probability of each node being in each state. This matrix will have size Nnodes x Nstates, where Nstates was either explicitly provided as an argument or inferred from `tip_states`. The rows in this matrix will be in the order in which nodes are indexed in the tree, i.e. the [n,s]-th entry will be the probability of the s-th state for the n-th node. Note that the name was chosen for compatibility with other ASR functions.

Author(s)

Stilianos Louca

References

D. Sankoff (1975). Minimal mutation trees of sequences. SIAM Journal of Applied Mathematics. 28:35-42.

J. Felsenstein (2004). Inferring Phylogenies. Sinauer Associates, Sunderland, Massachusetts.

See Also

`hsp_max_parsimony`, `asr_squared_change_parsimony` `asr_mk_model`, `hsp_mk_model`, `map_to_state_space`

Examples

```
# generate random tree
Ntips = 10
tree = generate_random_tree(list(birth_rate_intercept=1),max_tips=Ntips)$tree

# simulate a discrete trait
Nstates = 5
Q = get_random_mk_transition_matrix(Nstates, rate_model="ER")
tip_states = simulate_mk_model(tree, Q)$tip_states

# reconstruct node states via MPR
results = asr_max_parsimony(tree, tip_states, Nstates)
node_states = max.col(results$ancestral_likelihoods)

# print reconstructed node states
print(node_states)
```

asr_mk_model *Ancestral state reconstruction with Mk models and rerooting*

Description

Ancestral state reconstruction of a discrete trait using a fixed-rates continuous-time Markov model (a.k.a. "Mk model"). This function can estimate the (instantaneous) transition matrix using maximum likelihood, or take a specified transition matrix. The function can optionally calculate marginal ancestral state estimates for each node in the tree, using the rerooting method by Yang et al. (1995).

Usage

```
asr_mk_model( tree,
              tip_states,
              Nstates = NULL,
              tip_priors = NULL,
              rate_model = "ER",
              transition_matrix = NULL,
              include_ancestral_likelihoods = TRUE,
              root_prior = "empirical",
              Ntrials = 1,
              optim_algorithm = "nlminb",
              optim_max_iterations = 200,
              optim_rel_tol = 1e-8,
              store_exponentials = TRUE,
              check_input =TRUE,
              Nthreads = 1)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`tip_states`	An integer vector of size Ntips, specifying the state of each tip in the tree in terms of an integer from 1 to Nstates, where Ntips is the number of tips and Nstates is the number of possible states (see below). Can also be `NULL`. If `tip_states==NULL`, then `tip_priors` must not be `NULL` (see below).
`Nstates`	Either NULL, or an integer specifying the number of possible states of the trait. If `Nstates==NULL`, then it will be computed based on the maximum value encountered in `tip_states` or based on the number of columns in `tip_priors` (whichever is non-`NULL`).
`tip_priors`	A 2D numeric matrix of size Ntips x Nstates, where Nstates is the possible number of states for the character modelled. Can also be `NULL`. Each row of this matrix must be a probability vector, i.e. it must only contain non-negative entries and must sum up to 1. The [i,s]-th entry should be the prior probability of tip i being in state s. If you know for certain that tip i is in some state s, you can set the corresponding entry to 1 and all other entries in that row to 0. If you

	know the exact states of all tips, you can also pass these via `tip_states` instead. If `tip_priors==NULL`, then `tip_states` must not be `NULL` (see above).
`rate_model`	Rate model to be used for fitting the transition rate matrix. Can be "ER" (all rates equal), "SYM" (transition rate i–>j is equal to transition rate j–>i), "ARD" (all rates can be different), "SUEDE" (only stepwise transitions i–>i+1 and i–>i-1 allowed, all 'up' transitions are equal, all 'down' transitions are equal) or "SRD" (only stepwise transitions i–>i+1 and i–>i-1 allowed, and each rate can be different). Can also be an index matrix that maps entries of the transition matrix to the corresponding independent rate parameter to be fitted. Diagonal entries should map to 0, since diagonal entries are not treated as independent rate parameters but are calculated from the remaining entries in the transition matrix. All other entries that map to 0 represent a transition rate of zero. The format of this index matrix is similar to the format used by the `ace` function in the `ape` package. `rate_model` is only relevant if `transition_matrix==NULL`.
`transition_matrix`	Either a numeric quadratic matrix of size Nstates x Nstates containing fixed transition rates, or `NULL`. The [r,c]-th entry in this matrix should store the transition rate from state r to state c. Each row in this matrix must have sum zero. If `NULL`, then the transition rates will be estimated using maximum likelihood, based on the `rate_model` specified.
`root_prior`	Prior probability distribution of the root's states. Can be "flat" (all states equal), "empirical" (empirical probability distribution of states across the tree's tips) or "stationary" (stationary probability distribution of the transition matrix). If "stationary" and `transition_matrix==NULL`, then a transition matrix is first fitted using a flat root prior, and then used to calculate the stationary distribution. `root_prior` can also be a non-negative numeric vector of size Nstates and with total sum equal to 1.
`include_ancestral_likelihoods`	Include the marginal ancestral state distribution for each node (conditional scaled likelihoods after rerooting) in the return values. Note that this may increase the computation time and memory needed, so you may set this to `FALSE` if you don't need marginal ancestral states.
`Ntrials`	Number of trials (starting points) for fitting the transition matrix. Only relevant if `transition_matrix=NULL`. A higher number may reduce the risk of landing in a local non-global optimum of the likelihood function, but will increase computation time during fitting.
`optim_algorithm`	Either "optim" or "nlminb", specifying which optimization algorithm to use for maximum-likelihood estimation of the transition matrix. Only relevant if `transition_matrix==NULL`.
`optim_max_iterations`	Maximum number of iterations (per fitting trial) allowed for optimizing the likelihood function.
`optim_rel_tol`	Relative tolerance (stop criterion) for optimizing the likelihood function.
`store_exponentials`	Logical, specifying whether to pre-calculate and store exponentials of the transition matrix during calculation of ancestral likelihoods. This may reduce compu-

tation time because each exponential is only calculated once, but requires more memory since all exponentials are stored.

Only relevant if `include_ancestral_likelihoods==TRUE`, otherwise exponentials are never stored.

`check_input`	Logical, specifying whether to perform some basic checks on the validity of the input data. If you are certain that your input data are valid, you can set this to `FALSE` to reduce computation.
`Nthreads`	Number of parallel threads to use for running multiple fitting trials simultaneously. This only makes sense if your computer has multiple cores/CPUs and if `Ntrials>1`, and is only relevant if `transition_matrix==NULL`. This option is ignored on Windows, because Windows does not support forking.

Details

For this function, the trait's states must be represented by integers within 1,..,Nstates, where Nstates is the total number of possible states. If the states are originally in some other format (e.g. characters or factors), you should map them to a set of integers 1,..,Nstates. The order of states (if relevant) should be reflected in their integer representation. For example, if your original states are "small", "medium" and "large" and `rate_model=="SUEDE"`, it is advised to represent these states as integers 1,2,3. You can easily map any set of discrete states to integers using the function `map_to_state_space`.

This function allows the specification of the precise tip states (if these are known) using the vector `tip_states`. Alternatively, if some tip states are only known in terms of a probability distribution, you can pass these probability distributions using the matrix `tip_priors`. Note that exactly one of the two arguments, `tip_states` or `tip_priors`, must be non-`NULL`.

Tips must be represented in `tip_states` or `tip_priors` in the same order as in `tree$tip.label`. None of the input vectors or matrixes need include row or column names; if they do, however, they are checked for consistency (if `check_input==TRUE`).

The rerooting method by Yang et al (2015) is used to reconstruct the marginal ancestral state probabilities for each node by treating the node as a root and calculating its conditional scaled likelihoods. Note that the re-rooting algorithm is strictly speaking only valid for reversible Mk models, that is, satisfying the criterion

$$\pi_i Q_{ij} = \pi_j Q_{ji}, \quad \forall i, j,$$

where Q is the transition rate matrix and π is the stationary distribution of the model. The rate models "ER", 'SYM", "SUEDE" and "SRD" are reversible. For example, for "SUEDE" or "SRD" choose $\pi_{i+1} = \pi_i Q_{i,i+1}/Q_{i+1,i}$. In contrast, "ARD" models are generally not reversible.

If `tree$edge.length` is missing, each edge in the tree is assumed to have length 1. The tree may include multi-furcations (i.e. nodes with more than 2 children) as well as mono-furcations (i.e. nodes with only one child). This function is similar to `rerootingMethod` in the `phytools` package (v0.5-64) and similar to `ape::ace` (v4.1) with options `method="ML", type="discrete"` and `marginal=FALSE`, but tends to be much faster than `rerootingMethod` and `ace` for large trees.

Value

A list with the following elements:

`transition_matrix`
: A numeric quadratic matrix of size Nstates x Nstates, containing the transition rates of the Markov model. The [r,c]-th entry is the transition rate from state r to state c. Will be the same as the input `transition_matrix`, if the latter was not `NULL`.

`loglikelihood`
: Log-likelihood of the Markov model. If `transition_matrix` was `NULL` in the input, then this will be the log-likelihood maximized during fitting.

`ancestral_likelihoods`
: Optional, only returned if `include_ancestral_likelihoods` was `TRUE`. A 2D numeric matrix, listing the probability of each node being in each state (marginal ancestral state distributions). This matrix will have size Nnodes x Nstates, where Nstates was either explicitly provided as an argument, or inferred from `tip_states` or `tip_priors` (whichever was non-`NULL`). The rows in this matrix will be in the order in which nodes are indexed in the tree, i.e. the [n,s]-th entry will be the probability of the s-th state for the n-th node. For example, `likelihoods[1,3]` will store the probability that the first node was in state 3.

Author(s)

Stilianos Louca

References

Z. Yang, S. Kumar and M. Nei (1995). A new method for inference of ancestral nucleotide and amino acid sequences. Genetics. 141:1641-1650.

See Also

`hsp_mk_model`, `asr_max_parsimony`, `asr_squared_change_parsimony`, `hsp_max_parsimony`, `map_to_state_space`

Examples

```
# generate random tree
Ntips = 1000
tree  = generate_random_tree(list(birth_rate_intercept=1),max_tips=Ntips)$tree

# create random transition matrix
Nstates = 5
Q = get_random_mk_transition_matrix(Nstates, rate_model="ER", max_rate=0.01)
cat(sprintf("Simulated ER transition rate=%g\n",Q[1,2]))

# simulate the trait's evolution
tip_states = simulate_mk_model(tree, Q)$tip_states

# reconstruct node states from simulated tip states
results = asr_mk_model(tree, tip_states, Nstates, rate_model="ER", Ntrials=2, Nthreads=2)
node_states = max.col(results$ancestral_likelihoods)

# print Mk model fitting summary
cat(sprintf("Mk model: log-likelihood=%g\n",results$loglikelihood))
cat(sprintf("Universal (ER) transition rate=%g\n",results$transition_matrix[1,2]))
```

```
# print reconstructed node states for last 20 nodes
print(tail(node_states,20))
```

---

asr_squared_change_parsimony
Squared-change parsimony ancestral state reconstruction.

Description

Reconstruct ancestral states for a continuous (numeric) trait using squared-change maximum parsimony (Maddison, 1991). Transition costs can optionally be weighted by the inverse edge lengths ("weighted squared-change parsimony" by Maddison).

Usage

```
asr_squared_change_parsimony(tree, tip_states,  weighted=TRUE, check_input=TRUE)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`tip_states`	A numeric vector of size Ntips, specifying the known state of each tip in the tree.
`weighted`	Logical, specifying whether to weight transition costs by the inverted edge lengths. This corresponds to the "weighted squared-change parsimony" reconstruction by Maddison (1991) for a Brownian motion model of trait evolution.
`check_input`	Logical, specifying whether to perform some basic checks on the validity of the input data. If you are certain that your input data are valid, you can set this to `FALSE` to reduce computation.

Details

The function traverses the tree in postorder (tips–>root) to calculate the quadratic parameters described by Maddison (1991) and obtain the globally parsimonious squared-change parsimony state for the root. The function then reroots at each node, updates all affected quadratic parameters in the tree and calculates the node's globally parsimonious squared-change parsimony state. The function has asymptotic time complexity O(Nedges).

If `tree$edge.length` is missing, each edge in the tree is assumed to have length 1. This is the same as setting `weighted=FALSE`. The tree may include multi-furcations (i.e. nodes with more than 2 children) as well as mono-furcations (i.e. nodes with only one child). Edges with length 0 will be adjusted internally to some tiny length if needed (if `weighted==TRUE`).

Tips must be represented in `tip_states` in the same order as in `tree$tip.label`. The vector `tip_states` need not include item names; if it does, however, they are checked for consistency (if `check_input==TRUE`).

If `weighted==FALSE`, then this function yields the same ancestral state reconstructions as `ape::ace(tip_states, tree, type="continuous", method="ML", model="BM", CI=FALSE)`

in the ape package (v. 0.5-64), assuming the tree as unit edge lengths. If `weighted==TRUE`, then this function yields the same ancestral state reconstructions as the maximum likelihood estimates under a Brownian motion model, as implemented by the `Rphylopars` package (v. 0.2.10):

`Rphylopars::anc.recon(tip_states, tree, vars=FALSE, CI=FALSE).`

Value

A list with the following elements:

`ancestral_states`	A numeric vector of size Nnodes, listing the reconstructed state of each node. The entries in this vector will be in the order in which nodes are indexed in the tree.
`total_sum_of_squared_changes`	The total sum of squared changes, minimized by the (optionally weighted) squared-change parsimony algorithm. This is equation 7 in (Maddison, 1991).

Author(s)

Stilianos Louca

References

W. P. Maddison (1991). Squared-change parsimony reconstructions of ancestral states for continuous-valued characters on a phylogenetic tree. Systematic Zoology. 40:304-314.

See Also

`asr_independent_contrasts` `asr_max_parsimony`, `asr_mk_model`

Examples

```
# generate random tree
Ntips = 100
tree = generate_random_tree(list(birth_rate_intercept=1),max_tips=Ntips)$tree

# simulate a continuous trait
tip_states = simulate_ou_model(tree, stationary_mean=0, spread=1, decay_rate=0.001)$tip_states

# reconstruct node states based on simulated tip states
node_states = asr_squared_change_parsimony(tree, tip_states, weighted=TRUE)$ancestral_states
```

collapse_tree_at_resolution

Collapse nodes of a tree at a phylogenetic resolution.

Description

Given a rooted tree and a phylogenetic resolution threshold, collapse all nodes whose distance to all descending tips does not exceed the threshold (or whose sum of descending edge lengths does not exceed the threshold), into new tips. This function can be used to obtain a "coarser" version of the tree, or to cluster closely related tips into a single tip.

Usage

```
collapse_tree_at_resolution(tree,
                            resolution             = 0,
                            by_edge_count          = FALSE,
                            shorten                = TRUE,
                            rename_collapsed_nodes = FALSE,
                            criterion              = 'max_tip_depth')
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`resolution`	Numeric, specifying the phylogenetic resolution at which to collapse the tree. This is the maximum distance a descending tip can have from a node, such that the node is collapsed into a new tip. If set to 0 (default), then only nodes whose descending tips are identical to the node will be collapsed.
`by_edge_count`	Logical. Instead of considering edge lengths, consider edge counts as phylogenetic distance between nodes and tips. This is the same as if all edges had length equal to 1.
`shorten`	Logical, indicating whether collapsed nodes should be turned into tips at the same location (thus potentially shortening the tree). If `FALSE`, then the incoming edge of each collapsed node is extended by some length L, where L is the distance of the node to its farthest descending tip (thus maintaining the height of the tree).
`rename_collapsed_nodes`	Logical, indicating whether collapsed nodes should be renamed using a representative tip name (the farthest descending tip).
`criterion`	Character, specifying the criterion to use for collapsing (i.e. how to interpret `resolution`). '`max_tip_depth`': Collapse nodes based on their maximum distance to any descending tip. '`sum_tip_paths`': Collapse nodes based on the sum of descending edges (each edge counted once). '`max_tip_pair_dist`': Collapse nodes based on the maximum distance between any pair of descending tips.

Details

The input tree may include multi-furcations (i.e. nodes with more than 2 children) as well as mono-furcations (i.e. nodes with only one child).

Tip labels and uncollapsed node labels of the collapsed tree are inheritted from the original tree. If `rename_collapsed_nodes==FALSE`, then labels of collapsed nodes will be the node labels from the original tree (in this case the original tree should include node labels). If `rename_collapsed_nodes==TRUE`, each collapsed node is given the label of its farthest descending tip. If `shorten==TRUE`, then edge lengths are the same as in the original tree. If `shorten==FALSE`, then edges leading into collapsed nodes may be longer than before.

Value

A list with the following elements:

`tree`	A new rooted tree of class "phylo", containing the collapsed tree.
`collapsed_nodes`	Integer vector, listing indices of collapsed nodes in the original tree (subset of 1,..,Nnodes).
`new2old_clade`	Integer vector of length equal to the number of tips+nodes in the collapsed tree, with values in 1,..,Ntips+Nnodes, mapping tip/node indices of the collapsed tree to tip/node indices in the original tree.
`new2old_edge`	Integer vector of length equal to the number of edges in the collapsed tree, with values in 1,..,Nedges, mapping edge indices of the collapsed tree to edge indices in the original tree.

Author(s)

Stilianos Louca

Examples

```
# generate a random tree
tree = generate_random_tree(list(birth_rate_intercept=1),max_tips=1000)$tree

# print number of nodes
cat(sprintf("Simulated tree has %d nodes\n",tree$Nnode))

# collapse any nodes with tip-distances < 20
collapsed = collapse_tree_at_resolution(tree, resolution=20)$tree

# print number of nodes
cat(sprintf("Collapsed tree has %d nodes\n",collapsed$Nnode))
```

count_clades_over_time

Count number of clades over time.

Description

Given a rooted phylogenetic tree whose edge lengths represent time intervals, calculate the number of clades at various time points (e.g., spanning from 0 to the maximum time of any tip). The root is interpreted as time 0, and the distance of any node or tip from the root is interpreted as time elapsed since the root. This function defines an equidistant sequence of time points, and counts how many edges "cross" each time point. Optionally, the slopes and relative slopes of the clade-counts-vs-time curve are also returned. The slopes and relative slopes are approximations for the species birth rate and the per-capita species birth rate (assuming no extinctions occurred).

Usage

```
count_clades_over_time(tree, Ntimes=NULL, times=NULL, include_slopes=FALSE)
```

Arguments

`tree`	A rooted tree of class "phylo", where edge lengths represent time intervals (or similar). The root is assumed to be the unique node with no incoming edge.
`Ntimes`	Integer, number of equidistant time points for which to calculade clade counts. Can also be `NULL`, in which case `times` must be provided.
`times`	Integer vector, listing time points (in ascending order) for which to calculate clade counts. Can also be `NULL`, in which case `Ntimes` must be provided.
`include_slopes`	Logical, specifying whether the slope and the relative slope of the returned clades-per-time-point curve should also be returned.

Details

If `tree$edge.length` is missing, then every edge in the tree is assumed to be of length 1. The tree may include multi-furcations as well as mono-furcations (i.e. nodes with only one child). The tree need not be ultrametric, although in general this function only makes sense for dated trees (e.g., where edge lengths are time intervals or similar).

Either `Ntimes` or `times` must be non-`NULL`, but not both.

Value

A list with the following elements:

`Ntimes`	Integer, indicating the number of returned time points. Equal to the provided `Ntimes` if applicable.
`times`	Numeric vector of size Ntimes, listing the considered time points in increasing order. If `times` was provided as an argument to the function, then this will be the same as provided.

`diversities`	Integer vector of size Ntimes, listing the number of clades for each time point.
`slopes`	Numeric vector of size Ntimes, listing the slopes (finite-difference approximation of 1st derivative) of the curve clade_counts vs time_point.
`relative_slopes`	Numeric vector of size Ntimes, listing the relative slopes of the curve clade_counts vs time_point, i.e. `slopes` divided by a sliding-window average of `clade_counts`.

Author(s)

Stilianos Louca

Examples

```
# generate a random tree
tree = generate_random_tree(list(birth_rate_intercept=1), max_tips=1000)$tree

# count clades over time
results = count_clades_over_time(tree, Ntimes=100)

# plot curve (number of clades vs time)
plot(results$times, results$diversities, type="l", xlab="time", ylab="# clades")
```

`count_tips_per_node` *Count descending tips.*

Description

Given a rooted phylogenetic tree, count the number of tips descending (directy or indirectly) from each node.

Usage

```
count_tips_per_node(tree)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.

Details

The asymptotic time complexity of this function is O(Nedges), where Nedges is the number of edges.

Value

An integer vector of size Nnodes, with the i-th entry being the number of tips descending (directly or indirectly) from the i-th node.

Author(s)

Stilianos Louca

See Also

`get_subtree_at_node`

Examples

```
# generate a tree using a simple speciation model
tree = generate_random_tree(list(birth_rate_intercept=1),max_tips=1000)$tree

# count number of tips descending from each node
tips_per_node = count_tips_per_node(tree);

# plot histogram of tips-per-node
barplot(table(tips_per_node[tips_per_node<10]), xlab="# tips", ylab="# nodes")
```

`exponentiate_matrix` *Exponentiate a matrix.*

Description

Calculate the exponential $\exp(T \cdot A)$ of some quadratic real-valued matrix A for one or more scalar scaling factors T.

Usage

```
exponentiate_matrix(A, scalings=1, max_absolute_error=1e-3,
                    min_polynomials=1, max_polynomials=1000)
```

Arguments

`A`	A real-valued quadratic matrix of size N x N.
`scalings`	Vector of real-valued scalar scaling factors T, for each of which the exponential $\exp(T \cdot A)$ should be calculated.
`max_absolute_error`	Maximum allowed absolute error for the returned approximations. A smaller allowed error implies a greater computational cost as more matrix polynomials need to be included (see below). The returned approximations may have a greater error if the parameter `max_polynomials` is set too low.
`min_polynomials`	Minimum number of polynomials to include in the approximations (see equation below), even if `max_absolute_error` may be satisfied with fewer polynomials. If you don't know how to choose this, in most cases the default is fine. Note that regardless of `min_polynomials` and `max_absolute_error`, the number of polynomials used will not exceed `max_polynomials`.

`max_polynomials`
: Maximum allowed number of polynomials to include in the approximations (see equation below). Meant to provide a safety limit for the amount of memory and the computation time required. For example, a value of 1000 means that up to 1000 matrices (powers of A) of size N x N may be computed and stored temporarily in memory. Note that if `max_polynomials` is too low, the requested accuracy (via `max_absolute_error`) may not be achieved. That said, for large trees more memory may be required to store the actual result rather than the intermediate polynomials, i.e. for purposes of saving RAM it doesn't make much sense to choose `max_polynomials` much smaller than the length of `scalings`.

Details

Discrete character evolution Markov models often involve repeated exponentiations of the same transition matrix along each edge of the tree (i.e. with the scaling T being the edge length). Matrix exponentiation can become a serious computational bottleneck for larger trees or large matrices (i.e. spanning multiple discrete states). This function pre-calculates polynomials $A^p/p!$ of the matrix, and then uses linear combinations of the same polynomials for each requested T:

$$\exp(T\cdot A)=\sum_{p=0}^{P}T^p\frac{A^p}{p!}+...$$

This function thus becomes very efficient when the number of scaling factors is large (e.g. >10,000). The number of polynomials included is determined based on the specified `max_absolute_error`, and based on the largest (by magnitude) scaling factor requested. The function utilizes the balancing algorithm proposed by James et al (2014, Algorithm 3) and the scaling & squaring method (Moler and Van Loan, 2003) to improve the conditioning of the matrix prior to exponentiation.

Value

A 3D numeric matrix of size N x N x S, where N is the number of rows & column of the input matrix A and S is the length of `scalings`. The [r,c,s]-th element of this matrix is the entry in the r-th row and c-th column of $\exp(scalings[s]\cdot A)$.

Author(s)

Stilianos Louca

References

R. James, J. Langou and B. R. Lowery (2014). On matrix balancing and eigenvector computation. arXiv:1401.5766

C. Moler and C. Van Loan (2003). Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. SIAM Review. 45:3-49.

Examples

```
# create a random 5 x 5 matrix
A = get_random_mk_transition_matrix(Nstates=5, rate_model="ER")
```

```
# calculate exponentials exp(0.1*A) and exp(10*A)
exponentials = exponentiate_matrix(A, scalings=c(0.1,10))

# print 1st exponential: exp(0.1*A)
print(exponentials[,,1])

# print 2nd exponential: exp(10*A)
print(exponentials[,,2])
```

`find_nearest_tips` *Find nearest tip to each tip & node of a tree.*

Description

Given a phylogenetic tree and a subset of potential target tips, for each tip and node in the clade find the nearest target tip. The set of target tips can also be taken as the whole set of tips in the tree.

Usage

```
find_nearest_tips(tree, only_descending_tips=FALSE,
                    target_tips=NULL, as_edge_counts=FALSE,
                    check_input=TRUE)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`only_descending_tips`	A logical indicating whether the nearest tip to a node or tip should be chosen from its descending tips only. If FALSE, then the whole set of possible target tips is considered.
`target_tips`	Optional integer vector or character vector listing the subset of target tips to restrict the search to. If an integer vector, this should list tip indices (values in 1,..,Ntips). If a character vector, it should list tip names (in this case `tree$tip.label` must exist). If `target_tips` is `NULL`, then all tips of the tree are considered as target tips.
`as_edge_counts`	Logical, specifying whether to count phylogenetic distance in terms of edge counts instead of cumulative edge lengths. This is the same as setting all edge lengths to 1.
`check_input`	Logical, whether to perform basic validations of the input data. If you know for certain that your input is valid, you can set this to `FALSE` to reduce computation time.

Details

Langille et al. (2013) introduced the Nearest Sequenced Taxon Index (NSTI) as a measure for how well a set of microbial operational taxonomic units (OTUs) is represented by a set of sequenced genomes of related organisms. Specifically, the NSTI of a microbial community is the average phylogenetic distance of any OTU in the community, to the closest relative with an available sequenced genome ("target tips"). In analogy to the NSTI, the function `find_nearest_tips` provides a means to find the nearest tip (from a subset of target tips) to each tip and node in a phylogenetic tree, together with the corresponding phylogenetic ("patristic") distance.

If `only_descending_tips` is `TRUE`, then only descending target tips are considered when searching for the closest target tip of a node/tip. In that case, if a node/tip has no descending target tip, its nearest target tip is set to NA. If `tree$edge.length` is missing or NULL, then each edge is assumed to have length 1. The tree may include multi-furcations as well as mono-furcations (i.e. nodes with only one child).

The asymptotic time complexity of this function is O(Nedges), where Nedges is the number of edges in the tree.

Value

A list with the following elements:

`nearest_tip_per_tip`
: An integer vector of size Ntips, listing the nearest target tip for each tip in the tree. Hence, `nearest_tip_per_tip[i]` is the index of the nearest tip (from the set of target tips), with respect to tip i (where i=1,..,Ntips). Some values may appear multiple times in this vector, if multiple tips share the same nearest target tip.

`nearest_tip_per_node`
: An integer vector of size Nnodes, listing the index of the nearest target tip for each node in the tree. Hence, `nearest_tip_per_node[i]` is the index of the nearest tip (from the set of target tips), with respect to node i (where i=1,..,Nnodes). Some values may appear multiple times in this vector, if multiple nodes share the same nearest target tip.

`nearest_distance_per_tip`
: Integer vector of size Ntips. Phylogenetic ("patristic") distance of each tip in the tree to its nearest target tip. If `only_descending_tips` was set to `TRUE`, then `nearest_distance_per_tip[i]` will be set to infinity for any tip i that is not a target tip.

`nearest_distance_per_node`
: Integer vector of size Nnodes. Phylogenetic ("patristic") distance of each node in the tree to its nearest target tip. If `only_descending_tips` was set to `TRUE`, then `nearest_distance_per_node[i]` will be set to infinity for any node i that has no descending target tips.

Author(s)

Stilianos Louca

References

M. G. I. Langille, J. Zaneveld, J. G. Caporaso et al (2013). Predictive functional profiling of microbial communities using 16S rRNA marker gene sequences. Nature Biotechnology. 31:814-821.

Examples

```
# generate a random tree
Ntips = 1000
tree = generate_random_tree(list(birth_rate_intercept=1),Ntips)$tree

# pick a random set of "target" tips
target_tips = sample.int(n=Ntips, size=as.integer(Ntips/10), replace=FALSE)

# find nearest target tip to each tip & node in the tree
results = find_nearest_tips(tree, target_tips=target_tips)

# plot histogram of distances to target tips (across all tips of the tree)
distances = results$nearest_distance_per_tip
hist(distances, breaks=10, xlab="nearest distance", ylab="number of tips", prob=FALSE);
```

`find_root` *Find the root of a tree.*

Description

Find the root of a phylogenetic tree. The root is defined as the unique node with no parent.

Usage

```
find_root(tree)
```

Arguments

`tree`	A tree of class "phylo".

Details

By convention, the root of a "phylo" tree is typically the first node (i.e. with index `length(tree$tip.label)+1`), however this is not always guaranteed. This function finds the root of a tree by searching for the node with no parent. If no root exists, NA is returned. If multiple roots exist (this should generally not happen), only one of them will be returned. The asymptotic time complexity of this function is O(Nedges), where Nedges is the number of edges in the tree.

Value

Index of the tree's root, as listed in `tree$edge`. An integer ranging from Ntips+1 to Ntips+Nnodes, where Ntips and Nnodes is the number of tips and nodes in the tree, respectively. Typically (but not always), the root will be Ntips+1.

Author(s)

Stilianos Louca

See Also

root_at_node

Examples

```
# generate a random tree
Ntips = 100
tree = generate_random_tree(list(birth_rate_intercept=1),Ntips)$tree

# reroot the tree at the 20-th node
new_root_node = 20
tree = root_at_node(tree, new_root_node, update_indices=FALSE)

# find new root index and compare with expectation
cat(sprintf("New root is %d, expected at %d\n",find_root(tree),new_root_node+Ntips))
```

---

fit_bm_model *Fit a Brownian motion model for multivariate trait evolution.*

Description

Given a rooted phylogenetic tree and states of one or more continuous (numeric) traits on the tree's tips, fit a multivariate Brownian motion model of correlated co-evolution of these traits. This estimates a single diffusivity matrix, which describes the variance-covariance structure of each trait's random walk. The model assumes a fixed diffusivity matrix on the entire tree.

Usage

```
fit_bm_model(tree, tip_states, check_input=TRUE)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`tip_states`	A numeric vector of size Ntips, or a 2D numeric matrix of size Ntips x Ntraits, specifying the numeric state of each trait at each tip in the tree.
`check_input`	Logical, specifying whether to perform some basic checks on the validity of the input data. If you are certain that your input data are valid, you can set this to `FALSE` to reduce computation.

Details

The BM model is defined by the stochastic differential equation

$$dX = \sigma \cdot dW$$

where W is a multidimensional Wiener process with Ndegrees independent components and σ is a matrix of size Ntraits x Ndegrees. Alternatively, the same model can be defined as a Fokker-Planck equation for the probability density ρ:

$$\frac{\partial \rho}{\partial t} = \sum_{i,j} D_{ij} \frac{\partial^2 \rho}{\partial x_i \partial x_j}.$$

The matrix D is referred to as the diffusivity matrix (or diffusion tensor), and $2D = \sigma \cdot \sigma^T$. Note that σ can be obtained from D by means of a Cholesky decomposition.

The function uses phylogenetic independent contrasts (Felsenstein, 1985) to retrieve independent increments of the multivariate random walk. The diffusivity matrix D is then fitted using maximum-likelihood on the intrinsic geometry of positive-definite matrices.

If `tree$edge.length` is missing, each edge in the tree is assumed to have length 1. The tree may include multifurcations (i.e. nodes with more than 2 children) as well as monofurcations (i.e. nodes with only one child). Note that multifurcations are internally expanded to bifurcations, prior to model fitting. The asymptotic time complexity of this function is O(Nedges*Nsimulations*Ntraits).

Value

A list with the following elements:

`diffusivity`	Either a single non-negative number (if `tip_states` was a vector) or a 2D quadratic non-negative-definite matrix (if `tip_states` was a 2D matrix). The fitted diffusivity matrix of the multivariate Brownian motion model.
`loglikelihood`	The log-likelihood of the fitted model, given the provided tip states data.

Author(s)

Stilianos Louca

References

J. Felsenstein (1985). Phylogenies and the Comparative Method. The American Naturalist. 125:1-15.

See Also

`simulate_bm_model`, `get_independent_contrasts`

Examples

```
# generate a random tree
tree = generate_random_tree(list(birth_rate_intercept=1), 10000)$tree


# Example 1: Scalar case
# - - - - - - - - - - - - - - -
# simulate scalar continuous trait on the tree
D = 1
tip_states = simulate_bm_model(tree, diffusivity=D)$tip_states

# estimate original diffusivity from the generated data
fit = fit_bm_model(tree, tip_states)
cat(sprintf("True D=%g, fitted D=%g\n",D,fit$diffusivity))


# Example 2: Multivariate case
# - - - - - - - - - - - - - - - -
# simulate vector-valued continuous trait on the tree
D = get_random_diffusivity_matrix(Ntraits=5)
tip_states = simulate_bm_model(tree, diffusivity=D)$tip_states

# estimate original diffusivity matrix from the generated data
fit = fit_bm_model(tree, tip_states)

# compare true and fitted diffusivity matrices
cat("True D:\n"); print(D)
cat("Fitted D:\n"); print(fit$diffusivity)
```

`generate_random_tree` *Generate a tree using a Poissonian speciation/extinction model.*

Description

Generate a random phylogenetic tree via simulation of a Poissonian speciation/extinction (birth/death) process. New species are added (born) by splitting of a randomly chosen extant tip. The birth and death rates of tips can each be constant or power-law functions of the number of extant tips. For example,

$$B = I + F \cdot N^E,$$

where B is the birth (or speciation) rate, I is the intercept, F is the power-law factor, N is the current number of extant tips and E is the power-law exponent. Optionally, the per-capita birth and death rates can be extended by adding a custom time series provided by the user.

Usage

```
generate_random_tree(parameters          = list(),
                     max_tips            = NULL,
                     max_time            = NULL,
                     max_time_eq          = NULL,
                     coalescent           = TRUE,
                     as_generations       = FALSE,
                     Nsplits              = 2,
                     added_rates_times    = NULL,
                     added_birth_rates_pc = NULL,
                     added_death_rates_pc = NULL,
                     added_periodic       = FALSE,
                     tip_basename         = "",
                     node_basename        = NULL,
                     include_birth_times  = FALSE,
                     include_death_times  = FALSE)
```

Arguments

`parameters`	A named list specifying the birth-death model parameters, with one or more of the following entries: `birth_rate_intercept`: Non-negative number. The intercept of the Poissonian rate at which new species (tips) are added. In units 1/time. By default this is 0. `birth_rate_factor`: Non-negative number. The power-law factor of the Poissonian rate at which new species (tips) are added. In units 1/time. By default this is 0. `birth_rate_exponent`: Numeric. The power-law exponent of the Poissonian rate at which new species (tips) are added. Unitless. By default this is 1. `death_rate_intercept`: Non-negative number. The intercept of the Poissonian rate at which extant species (tips) go extinct. In units 1/time. By default this is 0. `death_rate_factor`: Non-negative number. The power-law factor of the Poissonian rate at which extant species (tips) go extinct. In units 1/time. By default this is 0. `death_rate_exponent`: Numeric. The power-law exponent of the Poissonian rate at which extant species (tips) go extinct. Unitless. By default this is 1. `rarefaction`: Numeric between 0 and 1. Rarefaction to be applied at the end of the simulation (fraction of random tips kept in the tree). Note that if `coalescent==FALSE`, rarefaction may remove both extant as well as extinct clades. Set `rarefaction=1` to not perform any rarefaction.
`max_tips`	Maximum number of tips of the tree to be generated. If `coalescent=TRUE`, this refers to the number of extant tips. Otherwise, it refers to the number of extinct + extant tips. If `NULL` or <=0, the number of tips is unlimited (so be careful).
`max_time`	Maximum duration of the simulation. If `NULL` or <=0, this constraint is ignored.
`max_time_eq`	Maximum duration of the simulation, counting from the first point at which speciation/extinction equilibrium is reached, i.e. when (birth rate - death rate) changed sign for the first time. If `NULL` or <0, this constraint is ignored.
`coalescent`	Logical, specifying whether only the coalescent tree (i.e. the tree spanning the extant tips) should be returned. If `coalescent==FALSE` and the death rate is

non-zero, then the tree may include non-extant tips (i.e. tips whose distance from the root is less than the total time of evolution). In that case, the tree will not be ultrametric.

`as_generations` Logical, specifying whether edge lengths should correspond to generations. If FALSE, then edge lengths correspond to time.

`Nsplits` Integer greater than 1. Number of child-tips to generate at each diversification event. If set to 2, the generated tree will be bifurcating. If >2, the tree will be multifurcating.

`added_rates_times`
Numeric vector, listing time points (in ascending order) for the custom per-capita birth and/or death rates time series (see `added_birth_rates_pc` and `added_death_rates_pc` below). Can also be `NULL`, in which case the custom time series are ignored.

`added_birth_rates_pc`
Numeric vector of the same size as `added_rates_times`, listing per-capita birth rates to be added to the power law part. Can also be `NULL`, in which case this option is ignored and birth rates are purely described by the power law.

`added_death_rates_pc`
Numeric vector of the same size as `added_rates_times`, listing per-capita death rates to be added to the power law part. Can also be `NULL`, in which case this option is ignored and death rates are purely described by the power law.

`added_periodic` Logical, indicating whether `added_birth_rates_pc` and `added_death_rates_pc` should be extended periodically if needed (i.e. if not defined for the entire simulation time). If `FALSE`, added birth & death rates are extended with zeros.

`tip_basename` Character. Prefix to be used for tip labels (e.g. "tip."). If empty (""), then tip labels will be integers "1", "2" and so on.

`node_basename` Character. Prefix to be used for node labels (e.g. "node."). If `NULL`, no node labels will be included in the tree.

`include_birth_times`
Logical. If `TRUE`, then the times of speciation events (in order of occurrence) will also be returned.

`include_death_times`
Logical. If `TRUE`, then the times of extinction events (in order of occurrence) will also be returned.

Details

If `max_time==NULL`, then the returned tree will always contain `max_tips` tips. In particular, if at any moment during the simulation the tree only includes a single extant tip, the death rate is temporarily set to zero to prevent the complete extinction of the tree. If `max_tips==NULL`, then the simulation is ran as long as specified by `max_time`. If neither `max_time` nor `max_tips` is `NULL`, then the simulation halts as soon as the time exceeds `max_time` or the number of tips (extant tips if `coalescent` is `TRUE`) exceeds `max_tips`. If `max_tips!=NULL` and `Nsplits>2`, then the last diversification even may generate fewer than `Nsplits` children, in order to keep the total number of tips within the specified limit.

Both the per-capita birth and death rates can be made into completely arbitrary functions of time, by setting all power-law coefficients to zero and providing custom time series `added_birth_rates_pc` and `added_death_rates_pc`.

Value

A named list with the following elements:

`tree`	A rooted bifurcating (if `Nsplits==2`) or multifurcating (if `Nsplits>2`) tree of class "phylo", generated according to the specified birth/death model. If `coalescent==TRUE` or if all death rates are zero, and only if `as_generations==FALSE`, then the tree will be ultrametric. If `as_generations==TRUE` and `coalescent==FALSE`, all edges will have unit length.
`root_time`	Numeric, giving the time at which the tree's root was first split during the simulation. Note that if `coalescent==TRUE`, this may be later than the first speciation event during the simulation.
`final_time`	Numeric, giving the final time at the end of the simulation. Note that if `coalescent==TRUE`, then this may be greater than the total time span of the tree (since the root of the coalescent tree need not correspond to the first speciation event).
`equilibrium_time`	Numeric, giving the first time where the sign of (death rate - birth rate) changed from the beginning of the simulation, i.e. when speciation/extinction equilibrium was reached. May be infinite if the simulation stoped before reaching this point.
`Nbirths`	Total number of birth events (speciations) that occurred during tree growth. This may be lower than the total number of tips in the tree if death rates were non-zero and `coalescent==TRUE`, or if `Nsplits>2`.
`Ndeaths`	Total number of deaths (extinctions) that occurred during tree growth.
`birth_times`	Numeric vector, listing the times of speciation events during tree growth, in order of occurrence. Note that if `coalescent==TRUE`, then `speciation_times` may be greater than the phylogenetic distance to the coalescent root.
`death_times`	Numeric vector, listing the times of extinction events during tree growth, in order of occurrence. Note that if `coalescent==TRUE`, then `speciation_times` may be greater than the phylogenetic distance to the coalescent root.

Author(s)

Stilianos Louca

References

D. J. Aldous (2001). Stochastic models and descriptive statistics for phylogenetic trees, from Yule to today. Statistical Science. 16:23-34.

M. Steel and A. McKenzie (2001). Properties of phylogenetic trees generated by Yule-type speciation models. Mathematical Biosciences. 170:91-112.

Examples

```
# Simple speciation model
parameters = list(birth_rate_intercept=1)
tree = generate_random_tree(parameters,max_tips=100)$tree

# Exponential growth rate model
parameters = list(birth_rate_factor=1)
tree = generate_random_tree(parameters,max_tips=100)$tree
```

---

generate_tree_with_evolving_rates

Generate a random tree with birth/death rates evolving under Brownian motion.

Description

Generate a random phylogenetic tree via simulation of a Poissonian speciation/extinction (birth/death) process. New species are added (born) by splitting of a randomly chosen extant tip. Per-capita birth and death rates evolve under a Brownian motion model constrained to a finite interval (via reflection). Thus, the probability rate of a tip splitting or going extinct depends on the tip, with closely related tips having more similar per-capita birth and death rates.

Usage

```
generate_tree_with_evolving_rates(parameters          = list(),
                                  max_tips            = NULL,
                                  max_time            = NULL,
                                  max_time_eq         = NULL,
                                  coalescent          = TRUE,
                                  as_generations      = FALSE,
                                  Nsplits             = 2,
                                  tip_basename        = "",
                                  node_basename       = NULL,
                                  include_birth_times = FALSE,
                                  include_death_times = FALSE,
                                  include_rates       = FALSE)
```

Arguments

`parameters`	A named list specifying the birth-death model parameters, with one or more of the following entries: `birth_rate_diffusivity`: Non-negative number. Diffusivity constant for the Brownian motion model of the evolving per-capita birth rate. In units 1/time^3. See `simulate_bm_model` for an explanation of the diffusivity parameter. `min_birth_rate_pc`: Non-negative number. The minimum allowed per-capita birth rate of a clade. In units 1/time. By default this is 0.

max_birth_rate_pc: Non-negative number. The maximum allowed per-capita birth rate of a clade. In units 1/time. By default this is 1.

death_rate_diffusivity: Non-negative number. Diffusivity constant for the Brownian motion model of the evolving per-capita death rate. In units 1/time^3. See simulate_bm_model for an explanation of the diffusivity parameter.

min_death_rate_pc: Non-negative number. The minimum allowed per-capita death rate of a clade. In units 1/time. By default this is 0.

max_death_rate_pc: Non-negative number. The maximum allowed per-capita death rate of a clade. In units 1/time. By default this is 1.

rarefaction: Numeric between 0 and 1. Rarefaction to be applied at the end of the simulation (fraction of random tips kept in the tree). Note that if coalescent==FALSE, rarefaction may remove both extant as well as extinct clades. Set rarefaction=1 to not perform any rarefaction.

max_tips	Maximum number of tips of the tree to be generated. If coalescent=TRUE, this refers to the number of extant tips. Otherwise, it refers to the number of extinct + extant tips. If NULL or <=0, the number of tips is unlimited (so be careful).
max_time	Maximum duration of the simulation. If NULL or <=0, this constraint is ignored.
max_time_eq	Maximum duration of the simulation, counting from the first point at which speciation/extinction equilibrium is reached, i.e. when (birth rate - death rate) changed sign for the first time. If NULL or <0, this constraint is ignored.
coalescent	Logical, specifying whether only the coalescent tree (i.e. the tree spanning the extant tips) should be returned. If coalescent==FALSE and the death rate is non-zero, then the tree may include non-extant tips (i.e. tips whose distance from the root is less than the total time of evolution). In that case, the tree will not be ultrametric.
as_generations	Logical, specifying whether edge lengths should correspond to generations. If FALSE, then edge lengths correspond to time.
Nsplits	Integer greater than 1. Number of child-tips to generate at each diversification event. If set to 2, the generated tree will be bifurcating. If >2, the tree will be multifurcating.
tip_basename	Character. Prefix to be used for tip labels (e.g. "tip."). If empty (""), then tip labels will be integers "1", "2" and so on.
node_basename	Character. Prefix to be used for node labels (e.g. "node."). If NULL, no node labels will be included in the tree.
include_birth_times	Logical. If TRUE, then the times of speciation events (in order of occurrence) will also be returned.
include_death_times	Logical. If TRUE, then the times of extinction events (in order of occurrence) will also be returned.
include_rates	Logical. If TRUE, then the bper-capita birth & death rates of all tips and nodes will also be returned.

Details

If `max_time==NULL`, then the returned tree will always contain `max_tips` tips. In particular, if at any moment during the simulation the tree only includes a single extant tip, the death rate is temporarily set to zero to prevent the complete extinction of the tree. If `max_tips==NULL`, then the simulation is ran as long as specified by `max_time`. If neither `max_time` nor `max_tips` is `NULL`, then the simulation halts as soon as the time exceeds `max_time` or the number of tips (extant tips if `coalescent` is `TRUE`) exceeds `max_tips`. If `max_tips!=NULL` and `Nsplits>2`, then the last diversification even may generate fewer than `Nsplits` children, in order to keep the total number of tips within the specified limit.

The per-capita birth and death rates of the root are chosen randomly, independently and uniformly from their respective allowed intervals.

Value

A named list with the following elements:

`success`	Logical, indicating whether the simulation was successful. If `FALSE`, an additional element `error` (of type character) is included containing an explanation of the error; in that case the value of any of the other elements is undetermined.
`tree`	A rooted bifurcating (if `Nsplits==2`) or multifurcating (if `Nsplits>2`) tree of class "phylo", generated according to the specified birth/death model. If `coalescent==TRUE` or if all death rates are zero, and only if `as_generations==FALSE`, then the tree will be ultrametric. If `as_generations==TRUE` and `coalescent==FALSE`, all edges will have unit length.
`root_time`	Numeric, giving the time at which the tree's root was first split during the simulation. Note that if `coalescent==TRUE`, this may be later than the first speciation event during the simulation.
`final_time`	Numeric, giving the final time at the end of the simulation. If `coalescent==TRUE`, then this may be greater than the total time span of the tree (since the root of the coalescent tree need not correspond to the first speciation event).
`equilibrium_time`	Numeric, giving the first time where the sign of (death rate - birth rate) changed from the beginning of the simulation, i.e. when speciation/extinction equilibrium was reached. May be infinite if the simulation stoped before reaching this point.
`Nbirths`	Total number of birth events (speciations) that occurred during tree growth. This may be lower than the total number of tips in the tree if death rates were non-zero and `coalescent==TRUE`, or if `Nsplits>2`.
`Ndeaths`	Total number of deaths (extinctions) that occurred during tree growth.
`birth_times`	Numeric vector, listing the times of speciation events during tree growth, in order of occurrence. Note that if `coalescent==TRUE`, then `speciation_times` may be greater than the phylogenetic distance to the coalescent root.
`death_times`	Numeric vector, listing the times of extinction events during tree growth, in order of occurrence. Note that if `coalescent==TRUE`, then `speciation_times` may be greater than the phylogenetic distance to the coalescent root.

`birth_rates_pc`	Numeric vector, listing the per-capita birth rate of each tip and node in the tree. The length of an edge in the tree was thus drawn from an exponential distribution with rate equal to the per-capita birth rate of the child tip or node.
`death_rates_pc`	Numeric vector, listing the per-capita death rate of each tip and node in the tree.

Author(s)

Stilianos Louca

References

D. J. Aldous (2001). Stochastic models and descriptive statistics for phylogenetic trees, from Yule to today. Statistical Science. 16:23-34.

Examples

```
# Generate tree
parameters = list(birth_rate_diffusivity  = 1,
                  min_birth_rate_pc       = 1,
                  max_birth_rate_pc       = 2,
                  death_rate_diffusivity  = 0.5,
                  min_death_rate_pc       = 0,
                  max_death_rate_pc       = 1)
simulation = generate_tree_with_evolving_rates(parameters,max_tips=1000,include_rates=TRUE)
tree       = simulation$tree
Ntips      = length(tree$tip.label)

# plot per-capita birth & death rates of tips
plot( x=simulation$birth_rates_pc[1:Ntips],
      y=simulation$death_rates_pc[1:Ntips],
      type='p',
      xlab="pc birth rate",
      ylab="pc death rate",
      main="Per-capita birth & death rates across tips",
      las=1)
```

`get_all_distances_to_root`

Get distances of all tips and nodes to the root.

Description

Given a rooted phylogenetic tree, calculate the phylogenetic distance (cumulative branch length) of the root to each tip and node.

Usage

```
get_all_distances_to_root(tree, as_edge_count=FALSE)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`as_edge_count`	Logical, specifying whether distances should be counted in number of edges, rather than cumulative edge length. This is the same as if all edges had length 1.

Details

If `tree$edge.length` is missing, then every edge in the tree is assumed to be of length 1. The tree may include multi-furcations as well as mono-furcations (i.e. nodes with only one child). The asymptotic average time complexity of this function is O(Nedges), where Nedges is the number of edges in the tree.

Value

A numeric vector of size Ntips+Nnodes, with the i-th element being the distance (cumulative branch length) of the i-th tip or node to the root. Tips are indexed 1,..,Ntips and nodes are indexed (Ntips+1),..,(Ntips+Nnodes).

Author(s)

Stilianos Louca

See Also

`get_pairwise_distances`

Examples

```
# generate a random tree
Ntips = 1000
tree = generate_random_tree(list(birth_rate_intercept=1,
                                 death_rate_intercept=0.5),
                            max_tips=Ntips)$tree

# calculate distances to root
all_distances = get_all_distances_to_root(tree)

# extract distances of nodes to root
node_distances = all_distances[(Ntips+1):(Ntips+tree$Nnode)]

# plot histogram of distances (across all nodes)
hist(node_distances, xlab="distance to root", ylab="# nodes", prob=FALSE);
```

`get_independent_contrasts` *Phylogenetic independent contrasts for continuous traits.*

Description

Calculate phylogenetic independent contrasts (PICs) for one or more continuous traits on a phylogenetic tree, as described by Felsenstein (1985). The traitstates are assumed to be known for all tips of the tree. PICs are commonly used to calculate correlations between multiple traits, while accounting for shared evolutionary history at the tips.

Usage

```
get_independent_contrasts(tree, tip_states,  scaled=TRUE,
                          only_bifurcations=FALSE, check_input=TRUE)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`tip_states`	A numeric vector of size Ntips, or a 2D numeric matrix of size Ntips x Ntraits, specifying the numeric state of each trait at each tip in the tree.
`scaled`	Logical, specifying whether to divide (standardize) PICs by the square root of their expected variance, as recommended by Felsenstein (1985).
`only_bifurcations`	Logical, specifying whether to only calculate PICs for bifurcating nodes. If `FALSE`, then multifurcations are temporarily expanded to bifurcations, and an additional PIC is calculated for each created bifurcation. If `TRUE`, then multifurcations are not expanded and PICs will not be calculated for them.
`check_input`	Logical, specifying whether to perform some basic checks on the validity of the input data. If you are certain that your input data are valid, you can set this to `FALSE` to reduce computation.

Details

If the tree is bifurcating, then one PIC is returned for each node. If multifurcations are present and `only_bifurcations==FALSE`, these are internally expanded to bifurcations and an additional PIC is returned for each such bifurcation. PICs are never returned for monofurcating nodes. Hence, in general the number of returned PICs is the number of bifurcations in the tree, potentially after multifurcations have been expanded to bifurcations (if `only_bifurcations==FALSE`).

If `tree$edge.length` is missing, each edge in the tree is assumed to have length 1. The tree may include multifurcations (i.e. nodes with more than 2 children) as well as monofurcations (i.e. nodes with only one child). Edges with length 0 will be adjusted internally to some tiny length (chosen to be much smaller than the smallest non-zero length).

Tips must be represented in `tip_states` in the same order as in `tree$tip.label`. The vector `tip_states` need not include item names; if it does, however, they are checked for consistency (if `check_input==TRUE`).

The function has asymptotic time complexity O(Nedges x Ntraits). It is more efficient to calculate PICs of multiple traits with the same function call, than to calculate PICs for each trait separately. For a single trait, this function is equivalent to the function `ape::pic`, with the difference that it can handle multifurcating trees.

Value

A list with the following elements:

`PICs`	A numeric vector (if `tip_states` is a vector) or a numeric matrix (if `tip_states` is a matrix), listing the phylogenetic independent contrasts for each trait and for each bifurcating node (potentially after multifurcations have been expanded). If a matrix, then `PICs[:,T]` will list the PICs for the T-th trait. Note that the order of elements in this vector (or rows, if `PICs` is a matrix) is not necesssarily the order of nodes in the tree, and that `PICs` may contain fewer or more elements (or rows) than there were nodes in the input tree.
`variances`	Numeric vector of the same size as `PICs`. The estimated variances for the PICs, under a Brownian motion model of trait evolution. These roughly correspond to the cumulative edge lengths between sister nodes from which PICs were calculated; hence their units are the same as those of edge lengths. See Felsenstein (1985) for details.
`nodes`	Integer vector of the same size as `PICs`, listing the node indices for which PICs are returned. If `only_bifurcations==FALSE`, then this vector may contain `NA`s, corresponding to temporary nodes created during expansion of multifurcations. If `only_bifurcations==TRUE`, then this vector will only list nodes that were bifurcating in the input tree. In that case, `PICs[1]` will correspond to the node with name `tree$node.label[nodes[1]]`, whereas `PICs[2]` will correspond to the node with name `tree$node.label[nodes[2]]`, and so on.

Author(s)

Stilianos Louca

References

J. Felsenstein (1985). Phylogenies and the Comparative Method. The American Naturalist. 125:1-15.

See Also

`asr_independent_contrasts`

Examples

```
# generate random tree
Ntips = 100
tree = generate_random_tree(list(birth_rate_intercept=1),Ntips)$tree

# simulate a continuous trait
tip_states = simulate_bm_model(tree, diffusivity=0.1, include_nodes=FALSE)$tip_states;

# calculate PICs
results = get_independent_contrasts(tree, tip_states, scaled=TRUE, only_bifurcations=TRUE)

# assign PICs to the bifurcating nodes in the input tree
PIC_per_node = rep(NA, tree$Nnode)
valids = which(!is.na(results$nodes))
PIC_per_node[results$nodes[valids]] = results$PICs[valids]
```

---

get_mrca_of_set *Most recent common ancestor of a set of tips/nodes.*

Description

Given a rooted phylogenetic tree and a set of tips and/or nodes ("descendants"), calculate the most recent common ancestor (MRCA) of those descendants.

Usage

```
get_mrca_of_set(tree, descendants)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`descendants`	An integer vector or character vector, specifying the tips/nodes for which to find the MRCA. If an integer vector, it must list indices of tips (from 1 to Ntips) and/or nodes (from Ntips+1 to Ntips+Nnodes), where Ntips and Nnodes is the number of tips and nodes in the tree, respectively. If a character vector, it must list tip and/or node names. In this case `tree` must include `tip.label`, as well as `node.label` if nodes are included in `descendants`.

Details

The tree may include multi-furcations (i.e. nodes with more than 2 children) as well as mono-furcations (i.e. nodes with only one child). Duplicate entries in `descendants` are ignored.

Value

An integer in 1,..,(Ntips+Nnodes), representing the MRCA using the same index as in `tree$edge`. If the MRCA is a tip, then this index will be in 1,..,Ntips. If the MRCA is a node, then this index will be in (Ntips+1),..,(Ntips+Nnodes).

Author(s)

Stilianos Louca

See Also

`get_pairwise_mrcas`, `get_tips_for_mrcas`

Examples

```
# generate a random tree
Ntips = 1000
tree = generate_random_tree(list(birth_rate_intercept=1),Ntips)$tree

# pick 3 random tips or nodes
descendants = sample.int(n=(Ntips+tree$Nnode), size=3, replace=FALSE)

# calculate MRCA of picked descendants
mrca = get_mrca_of_set(tree, descendants)
```

`get_pairwise_distances` *Get distances between pairs of tips or nodes.*

Description

Calculate phylogenetic ("patristic") distances between tips or nodes in some list A and tips or nodes in a second list B of the same size.

Usage

```
get_pairwise_distances(tree, A, B, as_edge_counts=FALSE, check_input=TRUE)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`A`	An integer vector or character vector of size Npairs, specifying the first of the two members of each pair for which to calculate the distance. If an integer vector, it must list indices of tips (from 1 to Ntips) and/or nodes (from Ntips+1 to Ntips+Nnodes). If a character vector, it must list tip and/or node names.

`B`	An integer vector or character vector of size Npairs, specifying the second of the two members of each pair for which to calculate the distance. If an integer vector, it must list indices of tips (from 1 to Ntips) and/or nodes (from Ntips+1 to Ntips+Nnodes). If a character vector, it must list tip and/or node names.
`check_input`	Logical, whether to perform basic validations of the input data. If you know for certain that your input is valid, you can set this to `FALSE` to reduce computation time.
`as_edge_counts`	Logical, specifying whether distances should be calculated in terms of edge counts, rather than cumulative edge lengths. This is the same as if all edges had length 1.

Details

The "patristic distance" between two tips and/or nodes is the shortest cumulative branch length that must be traversed along the tree in order to reach one tip/node from the other. Given a list of tips and/or nodes A, and a 2nd list of tips and/or nodes B of the same size, this function will calculate patristic distance between each pair (A[i], B[i]), where i=1,2,..,Npairs.

If `tree$edge.length` is missing, then each edge is assumed to be of length 1; this is the same as setting `as_edge_counts=TRUE`. The tree may include multi-furcations as well as mono-furcations (i.e. nodes with only one child). The input tree must be rooted at some node for technical reasons (see function `root_at_node`), but the choice of the root node does not influence the result. If `A` and/or `B` is a character vector, then `tree$tip.label` must exist. If node names are included in `A` and/or `B`, then `tree$node.label` must also exist.

The asymptotic average time complexity of this function for a balanced binary tree is O(Ntips+Npairs*log2(Ntips)).

Value

A numeric vector of size Npairs, with the i-th element being the patristic distance between the tips/nodes A[i] and B[i].

Author(s)

Stilianos Louca

See Also

`get_all_distances_to_root`

Examples

```
# generate a random tree
Ntips = 1000
tree = generate_random_tree(list(birth_rate_intercept=1),Ntips)$tree

# pick 3 random pairs of tips or nodes
Npairs = 3
A = sample.int(n=(Ntips+tree$Nnode), size=Npairs, replace=FALSE)
B = sample.int(n=(Ntips+tree$Nnode), size=Npairs, replace=FALSE)
```

```
# calculate distances
distances = get_pairwise_distances(tree, A, B)

# reroot at some other node
tree = root_at_node(tree, new_root_node=20, update_indices=FALSE)
new_distances = get_pairwise_distances(tree, A, B)

# verify that distances remained unchanged
print(distances)
print(new_distances)
```

`get_pairwise_mrcas` *Get most recent common ancestors of tip/node pairs.*

Description

Given a rooted phylogenetic tree and one or more pairs of tips and/or nodes, for each pair of tips/nodes find the most recent common ancestor (MRCA). If one clade is descendant of the other clade, the latter will be returned as MRCA.

Usage

```
get_pairwise_mrcas(tree, A, B, check_input=TRUE)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`A`	An integer vector or character vector of size Npairs, specifying the first of the two members of each pair of tips/nodes for which to find the MRCA. If an integer vector, it must list indices of tips (from 1 to Ntips) and/or nodes (from Ntips+1 to Ntips+Nnodes). If a character vector, it must list tip and/or node names.
`B`	An integer vector or character vector of size Npairs, specifying the second of the two members of each pair of tips/nodes for which to find the MRCA. If an integer vector, it must list indices of tips (from 1 to Ntips) and/or nodes (from Ntips+1 to Ntips+Nnodes). If a character vector, it must list tip and/or node names.
`check_input`	Logical, whether to perform basic validations of the input data. If you know for certain that your input is valid, you can set this to `FALSE` to reduce computation time.

Details

The tree may include multi-furcations as well as mono-furcations (i.e. nodes with only one child). If `tree$edge.length` is missing, then each edge is assumed to be of length 1. Note that in some cases the MRCA of two tips may be a tip, namely when both tips are the same.

If `A` and/or `B` is a character vector, then `tree$tip.label` must exist. If node names are included in `A` and/or `B`, then `tree$node.label` must also exist.

The asymptotic average time complexity of this function is O(Nedges), where Nedges is the number of edges in the tree.

Value

An integer vector of size Npairs with values in 1,..,Ntips (tips) and/or in (Ntips+1),..,(Ntips+Nnodes) (nodes), with the i-th element being the index of the MRCA of tips/nodes `A[i]` and `B[i]`.

Author(s)

Stilianos Louca

See Also

`get_mrca_of_set`, `get_tips_for_mrcas`

Examples

```
# generate a random tree
Ntips = 100
tree = generate_random_tree(list(birth_rate_intercept=1),Ntips)$tree

# pick 3 random pairs of tips or nodes
Npairs = 3
A = sample.int(n=(Ntips+tree$Nnode), size=Npairs, replace=FALSE)
B = sample.int(n=(Ntips+tree$Nnode), size=Npairs, replace=FALSE)

# calculate MRCAs
MRCAs = get_pairwise_mrcas(tree, A, B)
```

---

get_random_diffusivity_matrix

Create a random diffusivity matrix for a Brownian motion model.

Description

Create a random diffusivity matrix for a Brownian motion model of multi-trait evolution. This may be useful for testing purposes. The diffusivity matrix is drawn from the Wishart distribution of symmetric, nonnegative-definite matrixes:

$$D = X^T \cdot X, \quad X[i,j] \sim N(0,V), \quad i = 1,..,n, j = 1,..,p,$$

where n is the degrees of freedom, p is the number of traits and V is a scalar scaling.

Usage

```
get_random_diffusivity_matrix(Ntraits, degrees=NULL, V=1)
```

Arguments

`Ntraits`	The number of traits modelled. Equal to the number of rows and the number of columns of the returned matrix.
`degrees`	Degrees of freedom for the Wishart distribution. Must be equal to or greater than `Ntraits`. Can also be `NULL`, which is the same as setting it equal to `Ntraits`.
`V`	Positive number. A scalar scaling for the Wishart distribution.

Value

A real-valued quadratic symmetric non-negative definite matrix of size Ntraits x Ntraits. Almost surely (in the probabilistic sense), this matrix will be positive definite.

Author(s)

Stilianos Louca

See Also

`get_random_mk_transition_matrix`, `simulate_bm_model`

Examples

```
# generate a 5x5 diffusivity matrix
D = get_random_diffusivity_matrix(Ntraits=5)

# check that it is indeed positive definite
if(all(eigen(D)$values>0)){
  cat("Indeed positive definite\n");
}else{
  cat("Not positive definite\n");
}
```

`get_random_mk_transition_matrix`

Create a random transition matrix for an Mk model.

Description

Create a random transition matrix for a fixed-rates continuous-time Markov model of discrete trait evolution ("Mk model"). This may be useful for testing purposes.

Usage

```
get_random_mk_transition_matrix(Nstates, rate_model, min_rate=0, max_rate=1)
```

Arguments

`Nstates`	The number of distinct states represented in the transition matrix (number of rows & columns).
`rate_model`	Rate model that the transition matrix must satisfy. Can be "ER" (all rates equal), "SYM" (transition rate i–>j is equal to transition rate j–>i), "ARD" (all rates can be different) or "SUEDE" (only stepwise transitions i–>i+1 and i–>i-1 allowed, all 'up' transitions are equal, all 'down' transitions are equal).
`min_rate`	A non-negative number, specifying the minimum rate in off-diagonal entries of the transition matrix.
`max_rate`	A non-negative number, specifying the maximum rate in off-diagonal entries of the transition matrix. Must not be smaller than `min_rate`.

Value

A real-valued quadratic matrix of size Nstates x Nstates, representing a transition matrix for an Mk model. Each row will sum to 1. The [r,c]-th entry represents the transition rate r–>c. The number of unique off-diagonal rates will depend on the `rate_model` chosen.

Author(s)

Stilianos Louca

See Also

`exponentiate_matrix`, `get_stationary_distribution`

Examples

```
# generate a 5x5 Markov transition rate matrix
Q = get_random_mk_transition_matrix(Nstates=5, rate_model="ARD")
```

---

get_stationary_distribution
Stationary distribution of Markov transition matrix.

Description

Calculate the stationary probability distribution vector p for a transition matrix Q of a continuous-time Markov chain. That is, calculate $p \in [0,1]^n$ such that `sum(p)==0` and $p^TQ = 0$.

Usage

```
get_stationary_distribution(Q)
```

Arguments

`Q`	A valid transition rate matrix of size Nstates x Nstates, i.e. a quadratic matrix in which every row sums up to zero.

Details

A stationary distribution of a discrete-state continuous-time Markov chain is a probability distribution across states that remains constant over time, i.e. $p^TQ = 0$. Note that in some cases (i.e. if Q is not irreducible), there may be multiple distinct stationary distributions. In that case,which one is returned by this function is unpredictable. Internally, p is estimated by stepwise minimization of the norm of p^TQ, starting with the vector p in which every entry equals 1/Nstates.

Value

A numeric vector of size Nstates and with non-negative entries, satisfying the conditions `p%*%Q==0` and `sum(p)==1.0`.

Author(s)

Stilianos Louca

See Also

`exponentiate_matrix`

Examples

```
# generate a random 5x5 Markov transition matrix
Q = get_random_mk_transition_matrix(Nstates=5, rate_model="ARD")

# calculate stationary probability distribution
p = get_stationary_distribution(Q)
print(p)

# test correctness (p*Q should be 0, apart from rounding errors)
cat(sprintf("max(abs(p*Q)) = %g\n",max(abs(p %*% Q))))
```

`get_subtree_at_node` *Extract a subtree descending from a specific node.*

Description

Given a tree and a focal node, extract the subtree descending from the focal node and place the focal node as the root of the extracted subtree.

Usage

```
get_subtree_at_node(tree, node)
```

Arguments

`tree`	A tree of class "phylo".
`node`	Character or integer specifying the name or index, respectively, of the focal node at which to extract the subtree. If an integer, it must be between 1 and `tree$Nnode`. If a character, it must be a valid entry in `tree$node.label`.

Details

The input tree need not be rooted, however "descendance" from the focal node is inferred based on the direction of edges in `tree$edge`. The input tree may include multi-furcations (i.e. nodes with more than 2 children) as well as mono-furcations (i.e. nodes with only one child).

Value

A list with the following elements:

`subtree`	A new tree of class "phylo", containing the subtree descending from the focal node. This tree will be rooted, with the new root being the focal node.
`new2old_tip`	Integer vector of length Ntips_kept (=number of tips in the extracted subtree) with values in 1,..,Ntips, mapping tip indices of the subtree to tip indices in the original tree. In particular, `tree$tip.label[new2old_tip]` will be equal to `subtree$tip.label`.
`new2old_node`	Integer vector of length Nnodes_kept (=number of nodes in the extracted subtree) with values in 1,..,Nnodes, mapping node indices of the subtree to node indices in the original tree. For example, `new2old_node[1]` is the index that the first node of the subtree had within the original tree. In particular, `tree$node.label[new2old_node]` will be equal to `subtree$node.label` (if node labels are available).
`new2old_edge`	Integer vector of length Nedges_kept (=number of edges in the extracted subtree), with values in 1,..,Nedges, mapping edge indices of the subtree to edge indices in the original tree. In particular, `tree$edge.length[new2old_edge]` will be equal to `subtree$edge.length` (if edge lengths are available).

Author(s)

Stilianos Louca

See Also

`get_subtree_with_tips`

Examples

```
# generate a random tree
Ntips = 1000
tree = generate_random_tree(list(birth_rate_intercept=1),Ntips)$tree

# extract subtree descending from a random node
```

```
node = sample.int(tree$Nnode,size=1)
subtree = get_subtree_at_node(tree, node)$subtree

# print summary of subtree
cat(sprintf("Subtree at %d-th node has %d tips\n",node,length(subtree$tip.label)))
```

---

get_subtree_with_tips *Extract a subtree spanning a specific subset of tips.*

Description

Given a rooted tree and a subset of tips, extract the subtree containing only those tips. The root of the tree is kept.

Usage

```
get_subtree_with_tips(tree,
                      only_tips               = NULL,
                      omit_tips               = NULL,
                      collapse_monofurcations = TRUE,
                      force_keep_root         = FALSE)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`only_tips`	Either a character vector listing tip names to keep, or an integer vector listing tip indices to keep (between 1 and Ntips). Can also be `NULL`. Tips listed in `only_tips` not found in the tree will be silently ignored.
`omit_tips`	Either a character vector listing tip names to omit, or an integer vector listing tip indices to omit (between 1 and Ntips). Can also be `NULL`. Tips listed in `omit_tips` not found in the tree will be silently ignored.
`collapse_monofurcations`	A logical specifying whether nodes with a single outgoing edge remaining should be collapsed (removed). Incoming and outgoing edge of such nodes will be concatenated into a single edge, connecting the parent (or earlier) and child (or later) of the node. In that case, the returned tree will have edge lengths that reflect the concatenated edges.
`force_keep_root`	Logical, specifying whether to keep the root even if `collapse_monofurcations==TRUE` and the root of the subtree is left with a single child. If `FALSE`, and `collapse_monofurcations==TRUE`, the root may be removed and one of its descendants may become root.

Details

If both `only_tips` and `omit_tips` are `NULL`, then all tips are kept and the tree remains unchanged. If both `only_tips` and `omit_tips` are non-`NULL`, then only tips listed in `only_tips` and not listed in `omit_tips` will be kept. If `only_tips` and/or `omit_tips` is a character vector listing tip names, then `tree$tip.label` must exist.

If the input tree does not include `edge.length`, each edge in the input tree is assumed to have length 1. The root of the tree (which is always kept) is assumed to be the unique node with no incoming edge. The input tree may include multi-furcations (i.e. nodes with more than 2 children) as well as mono-furcations (i.e. nodes with only one child).

The asymptotic time complexity of this function is O(Nnodes+Ntips), where Ntips is the number of tips and Nnodes the number of nodes in the input tree.

When `only_tips==NULL`, `omit_tips!=NULL`, `collapse_monofurcations==TRUE` and `force_keep_root==FALSE`, this function is analogous to the function `drop.tip` in the `ape` package with option `trim_internal=TRUE` (v. 0.5-64).

Value

A list with the following elements:

`subtree`	A new tree of class "phylo", containing only the tips specified by `tips_to_keep` and the nodes & edges connecting those tips to the root. The returned tree will include `edge.lengh` as a member variable, listing the lengths of the remaining (possibly concatenated) edges.
`root_shift`	Numeric, indicating the phylogenetic distance between the old and the new root. Will always be non-negative.
`new2old_tip`	Integer vector of length Ntips_kept (=number of tips in the extracted subtree) with values in 1,..,Ntips, mapping tip indices of the subtree to tip indices in the original tree. In particular, `tree$tip.label[new2old_tip]` will be equal to `subtree$tip.label`.
`new2old_node`	Integer vector of length Nnodes_kept (=number of nodes in the extracted subtree) with values in 1,..,Nnodes, mapping node indices of the subtree to node indices in the original tree. For example, `new2old_node[1]` is the index that the first node of the subtree had within the original tree. In particular, `tree$node.label[new2old_node]` will be equal to `subtree$node.label` (if node labels are available).

Author(s)

Stilianos Louca

See Also

`get_subtree_at_node`

Examples

```
# generate a random tree
Ntips = 1000
tree = generate_random_tree(list(birth_rate_intercept=1),Ntips)$tree

# choose a random subset of tips
tip_subset = sample.int(Ntips, size=as.integer(Ntips/10), replace=FALSE)

# extract subtree spanning the chosen tip subset
subtree = get_subtree_with_tips(tree, only_tips=tip_subset)$subtree

# print summary of subtree
cat(sprintf("Subtree has %d tips and %d nodes\n",length(subtree$tip.label),subtree$Nnode))
```

`get_tips_for_mrcas` *Find tips with specific most recent common ancestors.*

Description

Given a rooted phylogenetic tree and a list of nodes ("MRCA nodes"), for each MRCA node find a set of descending tips ("MRCA-defining tips") such that their most recent common ancestor (MRCA) is that node. This may be useful for cases where nodes need to be described as MRCAs of tip pairs for input to certain phylogenetics algorithms (e.g., for tree dating).

Usage

```
get_tips_for_mrcas(tree, mrca_nodes, check_input=TRUE)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`mrca_nodes`	Either an integer vector or a character vector, listing the nodes for each of which an MRCA-defining set of tips is to be found. If an integer vector, it should list node indices (i.e. from 1 to Nnodes). If a character vector, it should list node names; in that case `tree$node.label` must exist.
`check_input`	Logical, whether to perform basic validations of the input data. If you know for certain that your input is valid, you can set this to `FALSE` to reduce computation time.

Details

At most 2 MRCA-defining tips are assigned to each MRCA node. This function assumes that each of the `mrca_nodes` has at least two children or has a child that is a tip (otherwise the problem is not well-defined). The tree may include multi-furcations as well as mono-furcations (i.e. nodes with only one child).

The asymptotic time complexity of this function is O(Ntips+Nnodes) + O(Nmrcas), where Ntips is the number of tips, Nnodes is the number of nodes in the tree and Nmrcas is equal to `length(mrca_nodes)`.

Value

A list of the same size as `mrca_nodes`, whose n-th element is an integer vector of tip indices (i.e. with values in 1,..,Ntips) whose MRCA is the n-th node listed in `mrca_nodes`.

Author(s)

Stilianos Louca

See Also

`get_pairwise_mrcas`, `get_mrca_of_set`

Examples

```
# generate a random tree
Ntips = 1000
tree = generate_random_tree(list(birth_rate_intercept=1),Ntips)$tree

# pick random nodes
focal_nodes = sample.int(n=tree$Nnode, size=3, replace=FALSE)

# get tips for mrcas
tips_per_focal_node = get_tips_for_mrcas(tree, focal_nodes);

# check correctness (i.e. calculate actual MRCAs of tips)
for(n in 1:length(focal_nodes)){
  mrca = get_mrca_of_set(tree, tips_per_focal_node[[n]])
  cat(sprintf("Focal node = %d, should match mrca of tips = %d\n",focal_nodes[n],mrca-Ntips))
}
```

`get_trait_acf` *Phylogenetic autocorrelation function of a numeric trait.*

Description

Given a rooted phylogenetic tree and a numeric (typically continuous) trait with known value (state) on each tip, calculate the phylogenetic autocorrelation function (ACF) of the trait. The ACF is a function of phylogenetic distance x, where ACF(x) is the Pearson autocorrelation of the trait between two tips, provided that the tips have phylogenetic ("patristic") distance x. The function `get_trait_acf` also calculates the mean absolute difference and the mean relative difference of the trait between any two random tips at phylogenetic distance x (see details below).

Usage

```
get_trait_acf(tree, tip_states, Npairs=10000, Nbins=10)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`tip_states`	A numeric vector of size Ntips, specifying the value of the trait at each tip in the tree. Note that tip_states[i] (where i is an integer index) must correspond to the i-th tip in the tree.
`Npairs`	Total number of random tip pairs to draw. A greater number of tip pairs will improve the accuracy of the estimated ACF within each distance bin.
`Nbins`	Number of distance bins to consider within the range of phylogenetic distances encountered between tip pairs in the tree. A greater number of bins will increase the resolution of the ACF as a function of phylogenetic distance, but will decrease the number of tip pairs falling within each bin (which reduces the accuracy of the estimated ACF).

Details

The phylogenetic autocorrelation function (ACF) of a trait can give insight into the evolutionary processes shaping its distribution across clades. An ACF that decays slowly with increasing phylogenetic distance indicates a strong phylogenetic conservatism of the trait, whereas a rapidly decaying ACF indicates weak phylogenetic conservatism. Similarly, if the mean absolute difference in trait value between two random tips increases with phylogenetic distance, this indicates a phylogenetic autocorrelation of the trait (Zaneveld et al. 2014). Here, phylogenetic distance between tips refers to their patristic distance, i.e. the minimum cumulative edge length required to connect the two tips.

Since, the distances between all possible tip pairs do not cover a continuoum (as there is only a finite number of tips), this function randomly draws tip pairs from the tree, maps them onto a finite set of equally-sized distance bins and then estimates the ACF for the centroid of each distance bin based on tip pairs in that bin. In practice, as a next step one would usually plot the estimated ACF (returned vector `autocorrelations`) over the centroids of the distance bins (returned vector `distances`).

The tree may include multi-furcations (i.e. nodes with more than 2 children) as well as mono-furcations (i.e. nodes with only one child). If `tree$edge.length` is missing, then every edge is assumed to have length 1. The input tree must be rooted at some node for technical reasons (see function `root_at_node`), but the choice of the root node does not influence the result.

Value

A list with the following elements:

`distances`	Numeric vector of size Nbins, storing the centroid phylogenetic distance of each distance bin in increasing order. The first and last distance bin approximately span the full range of phylogenetic distances encountered between any two random tips in the tree.
`autocorrelations`	Numeric vector of size Nbins, storing the estimated Pearson autocorrelation of the trait for each distance bin.

`mean_abs_differences`
: Numeric vector of size Nbins, storing the mean absolute difference of the trait between two random tips in each distance bin.

`mean_rel_differences`
: Numeric vector of size Nbins, storing the mean relative difference of the trait between two random tips in each distance bin. The relative difference between two values X and Y is 0 if $X == Y$, and equal to

$$\frac{|X-Y|}{0.5\cdot(|X|+|Y|)}$$

otherwise.

`Npairs_per_distance`
: Integer vector of size Nbins, storing the number of random tip pairs associated with each distance bin.

Author(s)

Stilianos Louca

References

J. R. Zanefeld and R. L. V. Thurber (2014). Hidden state prediction: A modification of classic ancestral state reconstruction algorithms helps unravel complex symbioses. Frontiers in Microbiology. 5:431.

See Also

`get_trait_depth`

Examples

```
# generate a random tree
tree = generate_random_tree(list(birth_rate_intercept=1),max_tips=10000)$tree

# simulate continuous trait evolution on the tree
tip_states = simulate_bm_model(tree, diffusivity=1)$tip_states

# calculate autocorrelation function
ACF = get_trait_acf(tree, tip_states, Npairs=1e7, Nbins=20)

# plot ACF (autocorrelation vs phylogenetic distance)
plot(ACF$distances, ACF$autocorrelations, type="l", xlab="distance", ylab="ACF")
```

`get_trait_depth` *Calculate depth of phylogenetic conservatism for a binary trait.*

Description

Given a rooted phylogenetic tree and presences/absences of a binary trait for each tip, calculate the mean phylogenetic depth at which the trait is conserved across clades, in terms of the consenTRAIT metric introduced by Martiny et al (2013). This is the mean depth of clades that are positive in the trait (i.e. in which a sufficient fraction of tips exhibits the trait).

Usage

```
get_trait_depth(tree,
                tip_states,
                min_fraction        = 0.9,
                count_singletons    = TRUE,
                singleton_resolution= 0,
                weighted            = FALSE,
                Npermutations       = 0)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`tip_states`	A numeric vector of size Ntips indicating absence (value <=0) or presence (value >0) of a particular trait at each tip of the tree. Note that tip_states[i] (where i is an integer index) must correspond to the i-th tip in the tree.
`min_fraction`	Minimum fraction of tips in a clade exhibiting the trait, for the clade to be considered "positive" in the trait. In the original paper by Martiny et al (2013), this was 0.9.
`count_singletons`	Logical, specifying whether to include singletons in the statistics (tips positive in the trait, but not part of a larger positive clade). The phylogenetic depth of singletons is taken to be half the length of their incoming edge, as proposed by Martiny et al (2013). If FALSE, singletons are ignored.
`singleton_resolution`	Numeric, specifying the phylogenetic resolution at which to resolve singletons. Any clade found to be positive in a trait will be considered a singleton if the distance of the clade's root to all descending tips is below this threshold.
`weighted`	Whether to weight positive clades by their number of positive tips. If FALSE, each positive clades is weighted equally, as proposed by Martiny et al (2013).
`Npermutations`	Number of random permutations for estimating the statistical significance of the mean trait depth. If zero (default), the statistical significance is not calculated.

Details

This function calculates the "consenTRAIT" metric (or variants thereof) proposed by Martiny et al. (2013) for measuring the mean phylogenetic depth at which a binary trait (e.g. presence/absence of a particular metabolic function) is conserved across clades. A greater mean depth means that the trait tends to be conserved in deeper-rooting clades. In their original paper, Martiny et al. proposed to consider a trait as conserved in a clade (i.e. marking a clade as "positive" in the trait) if at least 90% of the clade's tips exhibit the trait (i.e. are "positive" in the trait). This fraction can be controlled using the `min_fraction` parameter. The depth of a clade is taken as the average distance of its tips to the clade's root.

The default parameters of this function reflect the original choices made by Martiny et al. (2013), however in some cases it may be sensible to adjust them. For example, if you suspect a high risk of false positives in the detection of a trait, it may be worth setting `count_singletons` to `FALSE` to avoid skewing the distribution of conservation depths towards shallower depths due to false positives.

The statistical significance of the calculated mean depth, i.e. the probability of encountering such a mean dept or higher by chance, can be estimated based on a null model in which each tip is randomly and independently re-assigned a presence or absence of the trait. In the null model, the probability that a tip exhibits the trait is set to the fraction of positive entries in `tip_states`.

The tree may include multi-furcations as well as mono-furcations (i.e. nodes with only one child). If `tree$edge.length` is missing, then every edge is assumed to have length 1.

Value

A list with the following elements:

`mean_depth`	Mean phylogenetic depth of clades that are positive in the trait.
`var_depth`	Variance of phylogenetic depths of clades that are positive in the trait.
`min_depth`	Minimum phylogenetic depth of clades that are positive in the trait.
`max_depth`	Maximum phylogenetic depth of clades that are positive in the trait.
`Npositives`	Number of clades that are positive in the trait.
`P`	Statistical significance (P-values) of mean_depth, under a null model of random trait presences/absences (see details above). This is the probability that under the null model, the mean_depth would be at least as high as observed in the data.
`mean_random_depth`	Mean random mean_depth, under a null model of random trait presences/absences (see details above).
`positive_clades`	Integer vector, listing indices of tips and nodes (from 1 to Ntips+Nnodes) that were found to be positive in the trait and counted towards the statistic.
`positives_per_clade`	Integer vector of size Ntips+Nnodes, listing the number of descending tips per clade (tip or node) that were positive in the trait.
`mean_depth_per_clade`	Numeric vector of size Ntips+Nnodes, listing the mean phylogenetic depth of each clade (tip or node), i.e. the average distance to all its descending tips.

Author(s)

Stilianos Louca

References

A. C. Martiny, K. Treseder and G. Pusch (2013). Phylogenetic trait conservatism of functional traits in microorganisms. ISME Journal. 7:830-838.

See Also

`get_trait_acf`

Examples

```
# generate a random tree
tree = generate_random_tree(list(birth_rate_intercept=1),max_tips=1000)$tree

# simulate binary trait evolution on the tree
Q = get_random_mk_transition_matrix(Nstates=2, rate_model="ARD", max_rate=0.1)
tip_states = simulate_mk_model(tree, Q)$tip_states

# change states from 1/2 to 0/1 (presence/absence)
tip_states = tip_states - 1

# calculate phylogenetic conservatism of trait
results = get_trait_depth(tree, tip_states, count_singletons=FALSE, weighted=TRUE)
cat(sprintf("Mean depth = %g, std = %g\n",results$mean_depth,sqrt(results$var_depth)))
```

`get_trait_stats_over_time` *Calculate mean & standard deviation of a numeric trait on a dated tree over time.*

Description

Given a rooted and dated phylogenetic tree, and a scalar numeric trait with known value on each node and tip of the tree, calculate the mean and the variance of the trait's states across the tree at discrete time points. For example, if the trait represents "body size", then this function calculates the mean body size of extant clades over time.

Usage

```
get_trait_stats_over_time(tree, states, Ntimes=NULL, times=NULL, check_input=TRUE)
```

Arguments

`tree`	A rooted tree of class "phylo", where edge lengths represent time intervals (or similar).
`states`	Numeric vector, specifying the trait's state at each tip and each node of the tree (in the order in which tips & nodes are indexed). May include `NA` or `NaN` if values are missing for some tips/nodes.
`Ntimes`	Integer, number of equidistant time points for which to calculade clade counts. Can also be `NULL`, in which case `times` must be provided.
`times`	Integer vector, listing time points (in ascending order) for which to calculate clade counts. Can also be `NULL`, in which case `Ntimes` must be provided.
`check_input`	Logical, specifying whether to perform some basic checks on the validity of the input data. If you are certain that your input data are valid, you can set this to `FALSE` to reduce computation.

Details

If `tree$edge.length` is missing, then every edge in the tree is assumed to be of length 1. The tree may include multi-furcations as well as mono-furcations (i.e. nodes with only one child). The tree need not be ultrametric (e.g. may include extinct tips), although in general this function only makes sense if edge lengths correspond to time (or similar).

Either `Ntimes` or `times` must be non-`NULL`, but not both. `states` need not include names; if it does, then these are checked to be in the same order as in the tree (if `check_input==TRUE`).

Value

A list with the following elements:

`Ntimes`	Integer, indicating the number of returned time points. Equal to the provided `Ntimes` if applicable.
`times`	Numeric vector of size Ntimes, listing the considered time points in increasing order. If `times` was provided as an argument to the function, then this will be the same as provided.
`clade_counts`	Integer vector of size Ntimes, listing the number of tips or nodes considered at each time point.
`means`	Numeric vector of size Ntimes, listing the arithmetic mean of trait states at each time point.
`stds`	Numeric vector of size Ntimes, listing the population standard deviation of trait states at each time point.

Author(s)

Stilianos Louca

Examples

```
# generate a random tree
tree = generate_random_tree(list(birth_rate_intercept=1), max_tips=1000)$tree

# simulate a numeric trait under Brownian-motion
trait = simulate_bm_model(tree, diffusivity=1)
states = c(trait$tip_states,trait$node_states)

# calculate trait stats over time
results = get_trait_stats_over_time(tree, states, Ntimes=100)

# plot trait stats over time (mean +/- std)
M = results$means
S = results$stds
matplot(x=results$times,
        y=matrix(c(M-S,M+S),ncol=2,byrow=FALSE),
        main = "Simulated BM trait over time",
        lty = 1, col="black",
        type="l", xlab="time", ylab="mean +/- std")
```

`get_tree_span` *Get min and max distance of any tip to the root.*

Description

Given a rooted phylogenetic tree, calculate the minimum and maximum phylogenetic distance (cumulative branch length) of any tip from the root.

Usage

```
get_tree_span(tree, as_edge_count=FALSE)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`as_edge_count`	Logical, specifying whether distances should be counted in number of edges, rather than cumulative edge length. This is the same as if all edges had length 1.

Details

If `tree$edge.length` is missing, then every edge in the tree is assumed to be of length 1. The tree may include multi-furcations as well as mono-furcations (i.e. nodes with only one child). The asymptotic average time complexity of this function is O(Nedges), where Nedges is the number of edges in the tree.

Value

A named list with the following elements:

`min_distance`	Minimum phylogenetic distance that any of the tips has to the root.
`max_distance`	Maximum phylogenetic distance that any of the tips has to the root.

Author(s)

Stilianos Louca

See Also

`get_pairwise_distances`

Examples

```
# generate a random tree
Ntips   = 1000
params  = list(birth_rate_intercept=1, death_rate_intercept=0.5)
tree    = generate_random_tree(params, max_tips=Ntips, coalescent=FALSE)$tree

# calculate min & max tip distances from root
tree_span = get_tree_span(tree)
cat(sprintf("Tip min dist = %g, max dist = %g\n",
            tree_span$min_distance,
            tree_span$max_distance))
```

`get_tree_traversal_root_to_tips`
Traverse tree from root to tips.

Description

Create data structures for traversing a tree from root to tips, and for efficient retrieval of a node's outgoing edges and children.

Usage

```
get_tree_traversal_root_to_tips(tree, include_tips)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`include_tips`	Include tips in the tarversal queue. If FALSE, then only nodes are included in the queue.

Details

Many dynamic programming algorithms for phylogenetics involve traversing the tree in a certain direction (root to tips or tips to root), and efficient (O(1) complexity) access to a node's direct children can significantly speed up those algorithms. This function is meant to provide data structures that allow traversing the tree's nodes (and optionally tips) in such an order that each node is traversed prior to its descendants (root–>tips) or such that each node is traversed after its descendants (tips–>root). This function is mainly meant for use in other algorithms, and is probably of little relevance to the average user.

The tree may include multi-furcations as well as mono-furcations (i.e. nodes with only one child).

The asymptotic time and memory complexity of this function is O(Ntips), where Ntips is the number of tips in the tree.

Value

A list with the following elements:

`queue`	An integer vector of size Nnodes (if `include_tips` was `FALSE`) or of size Nnodes+Ntips (if `include_tips` was `TRUE`), listing indices of nodes (and optionally tips) in the order root–>tips described above. In particular, `queue[1]` will be the index of the tree's root (typically Ntips+1).
`edges`	An integer vector of size Nedges (=`nrow(tree$edge)`), listing indices of edges (corresponding to `tree$edge`) such that outgoing edges of the same node are listed in consequtive order.
`node2first_edge`	An integer vector of size Nnodes listing the location of the first outgoing edge of each node in `edges`. That is, `edges[node2first_edge[n]]` points to the first outgoing edge of node n in `tree$edge`.
`node2last_edge`	An integer vector of size Nnodes listing the location of the last outgoing edge of each node in `edges`. That is, `edges[node2last_edge[n]]` points to the last outgoing edge of node n in `tree$edge`. The total number of outgoing edges of a node is thus given by `1+node2last_edge[n]-node2first_edge[n]`.

Author(s)

Stilianos Louca

See Also

`reorder_tree_edges`

Examples

```
## Not run:
get_tree_traversal_root_to_tips(tree, include_tips=TRUE)

## End(Not run)
```

hsp_empirical_probabilities

Hidden state prediction via empirical probabilities.

Description

Reconstruct ancestral discrete states of nodes and predict unknown (hidden) states of tips on a tree based on empirical state probabilities across tips. This is a very crude HSP method, and other more sophisticated methods should be preferred (e.g. `hsp_mk_model`).

Usage

```
hsp_empirical_probabilities(tree, tip_states,
                            Nstates=NULL, check_input=TRUE)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`tip_states`	An integer vector of size Ntips, specifying the state of each tip in the tree as an integer from 1 to Nstates, where Nstates is the possible number of states (see below). `tip_states` can include `NA` to indicate an unknown tip state that is to be predicted.
`Nstates`	Either `NULL`, or an integer specifying the number of possible states of the trait. If `NULL`, then it will be computed based on the maximum non-`NA` value encountered in `tip_states`
`check_input`	Logical, specifying whether to perform some basic checks on the validity of the input data. If you are certain that your input data are valid, you can set this to `FALSE` to reduce computation.

Details

For this function, the trait's states must be represented by integers within 1,..,Nstates, where Nstates is the total number of possible states. If the states are originally in some other format (e.g. characters or factors), you should map them to a set of integers 1,..,Nstates. You can easily map any set of discrete states to integers using the function `map_to_state_space`.

Any `NA` entries in `tip_states` are interpreted as unknown states. Prior to ancestral state reconstruction, the tree is temporarily prunned, keeping only tips with known state. The function then calculates the empirical state probabilities for each node in the pruned tree, based on the states across tips descending from each node. The state probabilities of tips with unknown state are set to those of the most recent ancestor with reconstructed states, as described by Zanefeld and Thurber (2014).

The tree may include multi-furcations (i.e. nodes with more than 2 children) as well as mono-furcations (i.e. nodes with only one child). This function has asymptotic time complexity O(Nedges x Nstates).

Tips must be represented in `tip_states` in the same order as in `tree$tip.label`. The vector `tip_states` need not include names; if it does, however, they are checked for consistency (if `check_input==TRUE`).

This function is meant for reconstructing ancestral states in all nodes of a tree as well as predicting the states of tips with an a priory unknown state. If the state of all tips is known and only ancestral state reconstruction is needed, consider using functions such as `asr_empirical_probabilities` for improved efficiency.

Value

A list with the following elements:

`likelihoods`	A 2D numeric matrix, listing the probability of each tip and node being in each state. This matrix will have (Ntips+Nnodes) rows and Nstates columns, where Nstates was either explicitly provided as an argument or inferred based on the number of unique values in `tip_states` (if `Nstates` was passed as NULL). In the latter case, the column names of this matrix will be the unique values found in `tip_states`. The rows in this matrix will be in the order in which tips and nodes are indexed in the tree, i.e. the rows 1,..,Ntips store the probabilities for tips, while rows (Ntips+1),..,(Ntips+Nnodes) store the probabilities for nodes. Each row in this matrix will sum up to 1. Note that the return value is named this way for compatibility with other HSP functions.

Author(s)

Stilianos Louca

References

J. R. Zanefeld and R. L. V. Thurber (2014). Hidden state prediction: A modification of classic ancestral state reconstruction algorithms helps unravel complex symbioses. Frontiers in Microbiology. 5:431.

See Also

`hsp_max_parsimony`, `hsp_mk_model`, `map_to_state_space`

Examples

```
# generate random tree
Ntips = 100
tree = generate_random_tree(list(birth_rate_intercept=1),max_tips=Ntips)$tree

# simulate a discrete trait
Nstates = 5
Q = get_random_mk_transition_matrix(Nstates, rate_model="ER", max_rate=0.1)
tip_states = simulate_mk_model(tree, Q)$tip_states

# print states of first 20 tips
print(tip_states[1:20])
```

```
# set half of the tips to unknown state
tip_states[sample.int(Ntips,size=as.integer(Ntips/2),replace=FALSE)] = NA

# reconstruct all tip states via MPR
likelihoods = hsp_empirical_probabilities(tree, tip_states, Nstates)$likelihoods
estimated_tip_states = max.col(likelihoods[1:Ntips,])

# print estimated states of first 20 tips
print(estimated_tip_states[1:20])
```

---

hsp_independent_contrasts

Hidden state prediction via phylogenetic independent contrasts.

Description

Reconstruct ancestral states of a continuous (numeric) trait for nodes and predict unknown (hidden) states for tips on a tree using phylogenetic independent contrasts.

Usage

```
hsp_independent_contrasts(tree, tip_states, weighted=TRUE, check_input=TRUE)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`tip_states`	A numeric vector of size Ntips, specifying the state of each tip in the tree. `tip_states` can include `NA` to indicate an unknown tip state that is to be predicted.
`weighted`	Logical, specifying whether to weight transition costs by the inverted edge lengths during ancestral state reconstruction. This corresponds to the "weighted squared-change parsimony" reconstruction by Maddison (1991) for a Brownian motion model of trait evolution.
`check_input`	Logical, specifying whether to perform some basic checks on the validity of the input data. If you are certain that your input data are valid, you can set this to `FALSE` to reduce computation.

Details

Any `NA` entries in `tip_states` are interpreted as unknown (hidden) states to be estimated. Prior to ancestral state reconstruction, the tree is temporarily prunned, keeping only tips with known state. The function then uses a postorder traversal algorithm to calculate the intermediate "X" variables (a state estimate for each node) introduced by Felsenstein (1985) in his phylogenetic independent contrasts method. Note that these are only local estimates, i.e. for each node the estimate is only based on the tip states in the subtree descending from that node (see discussion in Garland and

Ives, 2000). The states of tips with hidden state are set to those of the most recent ancestor with reconstructed state, as described by Zanefeld and Thurber (2014).

This function has asymptotic time complexity O(Nedges). If `tree$edge.length` is missing, each edge in the tree is assumed to have length 1. This is the same as setting `weighted=FALSE`. The tree may include multi-furcations (i.e. nodes with more than 2 children) as well as mono-furcations (i.e. nodes with only one child).

Tips must be represented in `tip_states` in the same order as in `tree$tip.label`. The vector `tip_states` need not include item names; if it does, however, they are checked for consistency (if `check_input==TRUE`).

This function is meant for reconstructing ancestral states in all nodes of a tree as well as predicting the states of tips with an a priory unknown state. If the state of all tips is known and only ancestral state reconstruction is needed, consider using the function `asr_independent_contrasts` for improved efficiency.

Value

A list with the following elements:

`states`	A numeric vector of size Ntips+Nnodes, listing the reconstructed state of each tip and node. The entries in this vector will be in the order in which tips and nodes are indexed in `tree$edge`.
`total_sum_of_squared_changes`	The total sum of squared changes in tree, minimized by the (optionally weighted) squared-change parsimony algorithm. This is equation 7 in (Maddison, 1991). Note that for the root, phylogenetic independent contrasts is equivalent to Maddison's squared-change parsimony.

Author(s)

Stilianos Louca

References

J. Felsenstein (1985). Phylogenies and the comparative method. The American Naturalist. 125:1-15.

T. Jr. Garland and A. R. Ives (2000). Using the past to predict the present: Confidence intervals for regression equations in phylogenetic comparative methods. The American Naturalist. 155:346-364.

W. P. Maddison (1991). Squared-change parsimony reconstructions of ancestral states for continuous-valued characters on a phylogenetic tree. Systematic Zoology. 40:304-314.

J. R. Zanefeld and R. L. V. Thurber (2014). Hidden state prediction: A modification of classic ancestral state reconstruction algorithms helps unravel complex symbioses. Frontiers in Microbiology. 5:431.

See Also

`asr_squared_change_parsimony` `hsp_max_parsimony`, `hsp_mk_model`, `map_to_state_space`

Examples

```
# generate random tree
Ntips = 100
tree = generate_random_tree(list(birth_rate_intercept=1),max_tips=Ntips)$tree

# simulate a continuous trait
tip_states = simulate_ou_model(tree, stationary_mean=0, spread=1, decay_rate=0.001)$tip_states

# print tip states
print(as.vector(tip_states))

# set half of the tips to unknown state
tip_states[sample.int(Ntips,size=as.integer(Ntips/2),replace=FALSE)] = NA

# reconstruct all tip states via weighted PIC
estimated_states = hsp_independent_contrasts(tree, tip_states, weighted=TRUE)$states

# print estimated tip states
print(estimated_states[1:Ntips])
```

`hsp_max_parsimony` *Hidden state prediction via maximum parsimony.*

Description

Reconstruct ancestral discrete states of nodes and predict unknown (hidden) states of tips on a tree using maximum parsimony. Transition costs can vary between transitions, and can optionally be weighted by edge length.

Usage

```
hsp_max_parsimony(tree, tip_states, Nstates=NULL,
                  transition_costs="all_equal",
                  edge_exponent=0.0, weight_by_scenarios=TRUE,
                  check_input=TRUE)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`tip_states`	An integer vector of size Ntips, specifying the state of each tip in the tree as an integer from 1 to Nstates, where Nstates is the possible number of states (see below). `tip_states` can include `NA` to indicate an unknown tip state that is to be predicted.
`Nstates`	Either `NULL`, or an integer specifying the number of possible states of the trait. If `NULL`, then it will be computed based on the maximum non-`NA` value encountered in `tip_states`

`transition_costs`	Same as for the function `asr_max_parsimony`.
`edge_exponent`	Same as for the function `asr_max_parsimony`.
`weight_by_scenarios`	Logical, indicating whether to weight each optimal state of a node by the number of optimal maximum-parsimony scenarios in which the node is in that state. If FALSE, then all possible states of a node are weighted equally (i.e. are assigned equal probabilities).
`check_input`	Logical, specifying whether to perform some basic checks on the validity of the input data. If you are certain that your input data are valid, you can set this to `FALSE` to reduce computation.

Details

For this function, the trait's states must be represented by integers within 1,..,Nstates, where Nstates is the total number of possible states. If the states are originally in some other format (e.g. characters or factors), you should map them to a set of integers 1,..,Nstates. The order of states (if relevant) should be reflected in their integer representation. For example, if your original states are "small", "medium" and "large" and `transition_costs=="sequential"`, it is advised to represent these states as integers 1,2,3. You can easily map any set of discrete states to integers using the function `map_to_state_space`.

Any `NA` entries in `tip_states` are interpreted as unknown states. Prior to ancestral state reconstruction, the tree is temporarily prunned, keeping only tips with known state. The function then applies Sankoff's (1975) dynamic programming algorithm for ancestral state reconstruction, which determines the smallest number (or least costly if transition costs are uneven) of state changes along edges needed to reproduce the known tip states. The state probabilities of tips with unknown state are set to those of the most recent ancestor with reconstructed states, as described by Zanefeld and Thurber (2014). This function has asymptotic time complexity O(Ntips+Nnodes x Nstates).

If `tree$edge.length` is missing, each edge in the tree is assumed to have length 1. If `edge_exponent` is 0, then edge lengths do not influence the result. If `edge_exponent!=0`, then all edges must have non-zero length. The tree may include multi-furcations (i.e. nodes with more than 2 children) as well as mono-furcations (i.e. nodes with only one child).

Tips must be represented in `tip_states` in the same order as in `tree$tip.label`. None of the input vectors or matrixes need include row or column names; if they do, however, they are checked for consistency (if `check_input==TRUE`).

This function is meant for reconstructing ancestral states in all nodes of a tree as well as predicting the states of tips with an a priory unknown state. If the state of all tips is known and only ancestral state reconstruction is needed, consider using the function `asr_max_parsimony` for improved efficiency.

Value

A list with the following elements:

`likelihoods`	A 2D numeric matrix, listing the probability of each tip and node being in each state. This matrix will have (Ntips+Nnodes) rows and Nstates columns, where Nstates was either explicitly provided as an argument or inferred based on the

number of unique values in `tip_states` (if `Nstates` was passed as NULL). In the latter case, the column names of this matrix will be the unique values found in `tip_states`. The rows in this matrix will be in the order in which tips and nodes are indexed in the tree, i.e. the rows 1,..,Ntips store the probabilities for tips, while rows (Ntips+1),..,(Ntips+Nnodes) store the probabilities for nodes. Each row in this matrix will sum up to 1. Note that the return value is named this way for compatibility with other HSP functions.

Author(s)

Stilianos Louca

References

D. Sankoff (1975). Minimal mutation trees of sequences. SIAM Journal of Applied Mathematics. 28:35-42.

J. Felsenstein (2004). Inferring Phylogenies. Sinauer Associates, Sunderland, Massachusetts.

J. R. Zanefeld and R. L. V. Thurber (2014). Hidden state prediction: A modification of classic ancestral state reconstruction algorithms helps unravel complex symbioses. Frontiers in Microbiology. 5:431.

See Also

`asr_max_parsimony`, `asr_mk_model`, `hsp_mk_model`, `map_to_state_space`

Examples

```
# generate random tree
Ntips = 10
tree = generate_random_tree(list(birth_rate_intercept=1),max_tips=Ntips)$tree

# simulate a discrete trait
Nstates = 5
Q = get_random_mk_transition_matrix(Nstates, rate_model="ER")
tip_states = simulate_mk_model(tree, Q)$tip_states

# print tip states
print(tip_states)

# set half of the tips to unknown state
tip_states[sample.int(Ntips,size=as.integer(Ntips/2),replace=FALSE)] = NA

# reconstruct all tip states via MPR
likelihoods = hsp_max_parsimony(tree, tip_states, Nstates)$likelihoods
estimated_tip_states = max.col(likelihoods[1:Ntips,])

# print estimated tip states
print(estimated_tip_states)
```

hsp_mk_model *Hidden state prediction with Mk models and rerooting*

Description

Reconstruct ancestral states of a discrete trait and predict unknown (hidden) states of tips using a fixed-rates continuous-time Markov model (a.k.a. "Mk model"). This function can estimate the transition matrix using maximum likelihood, or take a specified transition matrix. The function can optionally calculate marginal ancestral state estimates for each node in the tree, using the rerooting method by Yang et al. (1995). A subset of the tips may have completely unknown state probabilities; in this case the fitted Markov model is used to predict their state probabilities based on their most recent reconstructed ancestor, as described by Zanefeld and Thurber (2014).

Usage

```
hsp_mk_model( tree,
              tip_states,
              Nstates = NULL,
              tip_priors = NULL,
              rate_model = "ER",
              transition_matrix = NULL,
              root_prior = "empirical",
              Ntrials = 1,
              optim_algorithm = "nlminb",
              optim_max_iterations = 200,
              optim_rel_tol = 1e-8,
              store_exponentials = TRUE,
              check_input = TRUE,
              Nthreads = 1)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`tip_states`	An integer vector of size Ntips, specifying the state of each tip in the tree in terms of an integer from 1 to Nstates, where Nstates is the possible number of states (see below). Can also be `NULL`, in which case `tip_priors` must not be `NULL` (see below). `tip_states` can include `NA` to indicate an unknown (hidden) tip state that is to be predicted.
`Nstates`	Either NULL, or an integer specifying the number of possible states of the trait. If `Nstates==NULL`, then it will be computed based on the maximum non-`NA` value encountered in `tip_states` or based on the number of columns in `tip_priors` (whichever is non-`NULL`).
`tip_priors`	A 2D numeric matrix of size Ntips x Nstates, where Nstates is the possible number of states for the character modelled. Can also be `NULL`. Each row of this matrix must be a probability vector, i.e. it must only contain non-negative

entries and must sum up to 1. The [i,s]-th entry should be the prior probability of tip i being in state s. If you know for certain that tip i is in some state s, you can set the corresponding entry to 1 and all other entries in that row to 0. A row can include `NA` to indicate that neither the state nor the probability distribution of a state are known for that tip. If for all tips you either know the exact state or have no information at all, you can also use `tip_states` instead. If `tip_priors==NULL`, then `tip_states` must not be `NULL` (see above).

`rate_model`	Rate model to be used for fitting the transition rate matrix. Similar to the `rate_model` option in the function `asr_mk_model`. See the details of `asr_mk_model` on the assumptions of each `rate_model`.
`transition_matrix`	Either a numeric quadratic matrix of size Nstates x Nstates containing fixed transition rates, or `NULL`. The [r,c]-th entry in this matrix should store the transition (probability) rate from the state r to state c. Each row in this matrix must have sum zero. If `NULL`, then the transition rates will be estimated using maximum likelihood, based on the `rate_model` specified.
`root_prior`	Prior probability distribution of the root's states. Similar to the `root_prior` option in the function `asr_mk_model`.
`Ntrials`	Number of trials (starting points) for fitting the transition matrix. Only relevant if `transition_matrix=NULL`. A higher number may reduce the risk of landing in a local non-global optimum of the likelihood function, but will increase computation time during fitting.
`optim_algorithm`	Either "optim" or "nlminb", specifying which optimization algorithm to use for maximum-likelihood estimation of the transition matrix. Only relevant if `transition_matrix==NULL`.
`optim_max_iterations`	Maximum number of iterations (per fitting trial) allowed for optimizing the likelihood function.
`optim_rel_tol`	Relative tolerance (stop criterion) for optimizing the likelihood function.
`store_exponentials`	Logical, specifying whether to pre-calculate and store exponentials of the transition matrix during calculation of ancestral likelihoods. This may reduce computation time because each exponential is only calculated once, but will use up more memory since all exponentials are stored. Only relevant if `include_ancestral_likelihoods` is `TRUE`, otherwise exponentials are never stored.
`check_input`	Logical, specifying whether to perform some basic checks on the validity of the input data. If you are certain that your input data are valid, you can set this to `FALSE` to reduce computation.
`Nthreads`	Number of parallel threads to use for running multiple fitting trials simultaneously. This only makes sense if your computer has multiple cores/CPUs and `Ntrials>1`, and is only relevant if `transition_matrix==NULL`.

Details

For this function, the trait's states must be represented by integers within 1,..,Nstates, where Nstates is the total number of possible states. Note that Nstates can be chosen to be larger than the number of states observed in the tips of the present tree, to account for potential states not yet observed. If the trait's states are originally in some other format (e.g. characters or factors), you should map them to a set of integers 1,..,Nstates. The order of states (if applicable) should be reflected in their integer representation. For example, if your original states are "small", "medium" and "large" and `rate_model=="SUEDE"`, it is advised to represent these states as integers 1,2,3. You can easily map any set of discrete states to integers using the function `map_to_state_space`.

This function allows the specification of the precise tip states (if these are known) using the vector `tip_states`. Alternatively, if some tip states are only known in terms of a probability distribution, you can pass these probability distributions using the matrix `tip_priors`. Note that exactly one of the two arguments, `tip_states` or `tip_priors`, must be non-`NULL`. In either case, the presence of `NA` in `tip_states` or in a row of `tip_priors` is interpreted as an absence of any information for the corresponding tip (i.e. the tip has "hidden state"), and thus the tip will be excluded from the ancestral state reconstruction step.

Tips must be represented in `tip_states` or `tip_priors` in the same order as in `tree$tip.label`. None of the input vectors or matrixes need include row or column names; if they do, however, they are checked for consistency (if `check_input==TRUE`).

The rerooting method by Yang et al (2015) is used to reconstruct the marginal ancestral state probabilities for each node by treating the node as a root and calculating its conditional scaled likelihoods. The state probabilities of tips with hidden states are calculated from those of the most recent ancestor with reconstructed states, by multiplying with the exponentiated transition matrix along the connecting edges (Zanefeld and Thurber, 2014).

If `tree$edge.length` is missing, each edge in the tree is assumed to have length 1. The tree may include multi-furcations (i.e. nodes with more than 2 children) as well as mono-furcations (i.e. nodes with only one child).

Value

A list with the following elements:

`transition_matrix`	A numeric quadratic matrix of size Nstates x Nstates, containing the transition rates of the Markov model. The [r,c]-th entry is the transition rate from state r to state c. Will be the same as the input `transition_matrix`, if the latter was not `NULL`.
`loglikelihood`	Log-likelihood of the Markov model. If `transition_matrix` was `NULL` in the input, then this will be the log-likelihood maximized during fitting.
`likelihoods`	A 2D numeric matrix, listing the probability of each tip and node being in each state. This matrix will have (Ntips+Nnodes) rows and Nstates columns, where Nstates was either explicitly provided as an argument, or inferred from `tip_states` or `tip_priors` (whichever was non-`NULL`). The rows in this matrix will be in the order in which tips and nodes are indexed in the tree, i.e. rows 1,..,Ntips store the probabilities for tips, while rows (Ntips+1),..,(Ntips+Nnodes) store the probabilities for nodes. For example, `likelihoods[1,3]` will store the probability that tip 1 is in state 3. Each row in this matrix will sum up to 1.

Author(s)

Stilianos Louca

References

Z. Yang, S. Kumar and M. Nei (1995). A new method for inference of ancestral nucleotide and amino acid sequences. Genetics. 141:1641-1650.

J. R. Zanefeld and R. L. V. Thurber (2014). Hidden state prediction: A modification of classic ancestral state reconstruction algorithms helps unravel complex symbioses. Frontiers in Microbiology. 5:431.

See Also

hsp_max_parsimony, hsp_squared_change_parsimony, asr_mk_model, map_to_state_space

Examples

```
# generate random tree
Ntips = 1000
tree  = generate_random_tree(list(birth_rate_intercept=1),max_tips=Ntips)$tree

# simulate a discrete trait
Nstates = 5
Q = get_random_mk_transition_matrix(Nstates, rate_model="ER", max_rate=0.01)
tip_states = simulate_mk_model(tree, Q)$tip_states
cat(sprintf("Simulated ER transition rate=%g\n",Q[1,2]))

# print states for first 20 tips
print(tip_states[1:20])

# set half of the tips to unknown state
tip_states[sample.int(Ntips,size=as.integer(Ntips/2),replace=FALSE)] = NA

# reconstruct all tip states via Mk model max-likelihood
results = hsp_mk_model(tree, tip_states, Nstates, rate_model="ER", Ntrials=2, Nthreads=2)
estimated_tip_states = max.col(results$likelihoods[1:Ntips,])

# print Mk model fitting summary
cat(sprintf("Mk model: log-likelihood=%g\n",results$loglikelihood))
cat(sprintf("Universal (ER) transition rate=%g\n",results$transition_matrix[1,2]))

# print estimated states for first 20 tips
print(estimated_tip_states[1:20])
```

---

hsp_squared_change_parsimony

Hidden state prediction via squared-change parsimony.

Description

Reconstruct ancestral states of a continuous (numeric) trait for nodes and predict unknown (hidden) states for tips on a tree using squared-change (or weighted squared-change) parsimony (Maddison 1991).

Usage

```
hsp_squared_change_parsimony(tree, tip_states, weighted=TRUE, check_input=TRUE)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`tip_states`	A numeric vector of size Ntips, specifying the state of each tip in the tree. `tip_states` can include `NA` to indicate an unknown tip state that is to be predicted.
`weighted`	Logical, specifying whether to weight transition costs by the inverted edge lengths during ancestral state reconstruction. This corresponds to the "weighted squared-change parsimony" reconstruction by Maddison (1991) for a Brownian motion model of trait evolution.
`check_input`	Logical, specifying whether to perform some basic checks on the validity of the input data. If you are certain that your input data are valid, you can set this to `FALSE` to reduce computation.

Details

Any `NA` entries in `tip_states` are interpreted as unknown (hidden) states to be estimated. Prior to ancestral state reconstruction, the tree is temporarily prunned, keeping only tips with known state. The function then uses Maddison's squared-change parsimony algorithm to reconstruct the globally parsimonious state at each node (Maddison 1991). The states of tips with hidden state are set to those of the most recent ancestor with reconstructed state, as described by Zanefeld and Thurber (2014). This function has asymptotic time complexity O(Nedges). If `tree$edge.length` is missing, each edge in the tree is assumed to have length 1. This is the same as setting `weighted=FALSE`. The tree may include multi-furcations (i.e. nodes with more than 2 children) as well as mono-furcations (i.e. nodes with only one child).

Tips must be represented in `tip_states` in the same order as in `tree$tip.label`. The vector `tip_states` need not include item names; if it does, however, they are checked for consistency (if `check_input==TRUE`).

This function is meant for reconstructing ancestral states in all nodes of a tree as well as predicting the states of tips with an a priory unknown state. If the state of all tips is known and only ancestral state reconstruction is needed, consider using the function `asr_squared_change_parsimony` for improved efficiency.

Value

A list with the following elements:

`states`	A numeric vector of size Ntips+Nnodes, listing the reconstructed state of each tip and node. The entries in this vector will be in the order in which tips and nodes are indexed in `tree$edge`.
`total_sum_of_squared_changes`	The total sum of squared changes, minimized by the (optionally weighted) squared-change parsimony algorithm. This is equation 7 in (Maddison, 1991).

Author(s)

Stilianos Louca

References

W. P. Maddison (1991). Squared-change parsimony reconstructions of ancestral states for continuous-valued characters on a phylogenetic tree. Systematic Zoology. 40:304-314.

J. R. Zanefeld and R. L. V. Thurber (2014). Hidden state prediction: A modification of classic ancestral state reconstruction algorithms helps unravel complex symbioses. Frontiers in Microbiology. 5:431.

See Also

`asr_squared_change_parsimony` `hsp_max_parsimony`, `hsp_mk_model`, `map_to_state_space`

Examples

```
# generate random tree
Ntips = 100
tree = generate_random_tree(list(birth_rate_intercept=1),max_tips=Ntips)$tree

# simulate a continuous trait
tip_states = simulate_ou_model(tree, stationary_mean=0, spread=1, decay_rate=0.001)$tip_states

# print tip states
print(tip_states)

# set half of the tips to unknown state
tip_states[sample.int(Ntips,size=as.integer(Ntips/2),replace=FALSE)] = NA

# reconstruct all tip states via weighted SCP
estimated_states = hsp_squared_change_parsimony(tree, tip_states, weighted=TRUE)$states

# print estimated tip states
print(estimated_states[1:Ntips])
```

`is_monophyletic` *Determine if a set of tips is monophyletic.*

Description

Given a rooted phylogenetic tree and a set of focal tips, this function determines whether the tips form a monophyletic group.

Usage

```
is_monophyletic(tree, focal_tips, check_input=TRUE)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`focal_tips`	Either an integer vector or a character vector, listing the tips to be checked for monophyly. If an integer vector, it should list tip indices (i.e. from 1 to Ntips). If a character vector, it should list tip names; in that case `tree$tip.label` must exist.
`check_input`	Logical, whether to perform basic validations of the input data. If you know for certain that your input is valid, you can set this to `FALSE` to reduce computation time.

Details

This function first finds the most recent common ancestor (MRCA) of the focal tips, and then checks if all tips descending from that MRCA fall within the focal tip set.

Value

A logical, indicating whether the focal tips form a monophyletic set.

Author(s)

Stilianos Louca

See Also

`get_mrca_of_set`

Examples

```
# generate random tree
Ntips = 100
tree  = generate_random_tree(list(birth_rate_intercept=1),max_tips=Ntips)$tree

# pick a random subset of focal tips
focal_tips = which(sample.int(2,size=Ntips,replace=TRUE)==1)

# check if focal tips form a monophyletic group
is_monophyletic(tree, focal_tips)
```

`map_to_state_space` *Map states of a discrete trait to integers.*

Description

Given a list of states (e.g., for each tip in a tree), map the unique states to integers 1,..,Nstates, where Nstates is the number of possible states. This function can be used to translate states that are originally represented by characters or factors, into integer states as required by ancestral state reconstruction and hidden state prediction functions in this package.

Usage

```
map_to_state_space(raw_states, fill_gaps=FALSE,
                   sort_order="natural", include_state_values=FALSE)
```

Arguments

`raw_states`	A vector of values (states), each of which can be converted to a different character. This list can include the same value multiple times, for example if values represent the trait's states for tips in a tree.
`fill_gaps`	Logical. If `TRUE`, then states are converted to integers using `as.integer(as.character())`, and then all missing intermediate integer values are included as additional possible states. For example, if `raw_states` contained the values 2,4,6, then 3 and 5 are assumed to also be possible states.
`sort_order`	Character, specifying the order in which raw_states should be mapped to ascending integers. Either "natural" or "alphabetical". If "natural", numerical parts of characters are sorted numerically, e.g. as in "3"<"a2"<"a12"<"b1".
`include_state_values`	Logical, specifying whether to also return a numerical version of the unique states. For example, the states "3","a2","4.5" will be mapped to the numeric values 3, NA, 4.5.

Details

Several ancestral state reconstruction and hidden state prediction algorithms in the `castor` package (e.g., `asr_max_parsimony`) require that the focal trait's states are represented by integer indices within 1,..,Nstates. These indices are then associated, afor example, with column and row indices in the transition cost matrix (in the case of maximum parsimony reconstruction) or with column indices in the returned matrix containing marginal ancestral state probabilities (e.g., in `asr_mk_model`). The function `map_to_state_space` can be used to conveniently convert a set of discrete states into integers, for use with the aforementioned algorithms.

Value

A list with the following elements:

`Nstates`	Integer. Number of possible states for the trait, based on the unique values encountered in `raw_states` (after conversion to characters). This may be larger than the number of unique values in `raw_states`, if `fill_gaps` was set to `TRUE`.
`state_names`	Character vector of size Nstates, storing the original name (character version) of each state. For example, if `raw_states` was `c("b1","3","a12","a2","b1","a2")` and `sort_order=="natural"`, then `Nstates` will be 4 and `state_names` will be `c("3","a2","a12","b1")`.
`state_values`	Optional, only included if `include_state_values==TRUE`. A numeric vector of size `Nstates`, providing the numerical value for each unique state.
`mapped_states`	Integer vector of size equal to `length(raw_states)`, listing the integer representation of each value in `raw_states`.

`name2index`	An integer vector of size Nstates, with `names(name2index)` set to `state_names`. This vector can be used to map any new list of states (in character format) to their integer representation. In particular, `name2index[as.character(raw_states)]` is equal to `mapped_states`.

Author(s)

Stilianos Louca

Examples

```
# generate a sequence of random states
unique_states = c("b","c","a")
raw_states = unique_states[sample.int(3,size=10,replace=TRUE)]

# map to integer state space
mapping = map_to_state_space(raw_states)

cat(sprintf("Checking that original unique states is the same as the one inferred:\n"))
print(unique_states)
print(mapping$state_names)

cat(sprintf("Checking reversibility of mapping:\n"))
print(raw_states)
print(mapping$state_names[mapping$mapped_states])
```

`merge_short_edges` *Eliminate short edges in a tree by merging nodes into multifurcations.*

Description

Given a rooted phylogenetic tree and an edge length threshold, merge nodes/tips into multifurcations when their incoming edges are shorter than the threshold.

Usage

```
merge_short_edges(tree,
                  edge_length_epsilon = 0,
                  force_keep_tips     = TRUE,
                  new_tip_prefix      = "ex.node.tip.")
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`edge_length_epsilon`	Non-negative numeric, specifying the maximum edge length for an edge to be considered “short” and thus to be eliminated. Typically 0 or some small positive number.

`force_keep_tips`	Logical. If `TRUE`, then tips are always kept, even if their incoming edges are shorter than `edge_length_epsilon`. If `FALSE`, then tips with short incoming edges are removed from the tree; in that case some nodes may become tips.
`new_tip_prefix`	Character or `NULL`, specifying the prefix to use for new tip labels stemming from nodes. Only relevant if `force_keep_tips==FALSE`. If `NULL`, then labels of tips stemming from nodes will be the node labels from the original tree (in this case the original tree should include node labels).

Details

The input tree may include multi-furcations (i.e. nodes with more than 2 children) as well as mono-furcations (i.e. nodes with only one child). Whenever a short edge is eliminated, the edges originating from its child are elongated according to the short edge's length. The corresponding grand-children become children of the short edge's parent. Short edges are eliminated in a depth-first-search manner, i.e. traversing from the root to the tips.

Note that existing monofurcations are retained. If `force_keep_tips==FALSE`, then new monofurcations may also be introduced due to tips being removed.

This function is conceptually similar to the function `ape::di2multi`.

Value

A list with the following elements:

`tree`	A new rooted tree of class "phylo", containing the (potentially multifurcating) tree.
`new2old_clade`	Integer vector of length equal to the number of tips+nodes in the new tree, with values in 1,..,Ntips+Nnodes, mapping tip/node indices of the new tree to tip/node indices in the original tree.
`new2old_edge`	Integer vector of length equal to the number of edges in the new tree, with values in 1,..,Nedges, mapping edge indices of the new tree to edge indices in the original tree.
`Nedges_removed`	Integer. Number of edges that have been eliminated.

Author(s)

Stilianos Louca

See Also

`multifurcations_to_bifurcations`

Examples

```
# generate a random tree
Ntips = 100
tree = generate_random_tree(list(birth_rate_factor=1),max_tips=Ntips)$tree

# set some edge lengths to zero
```

```
tree$edge.length[sample.int(n=Ntips, size=10, replace=FALSE)] = 0

# print number of edges
cat(sprintf("Original tree has %d edges\n",nrow(tree$edge)))

# eliminate any edges of length zero
merged = merge_short_edges(tree, edge_length_epsilon=0)$tree

# print number of edges
cat(sprintf("New tree has %d edges\n",nrow(merged$edge)))
```

`multifurcations_to_bifurcations` *Expand multifurcations to bifurcations.*

Description

Eliminate multifurcations from a phylogenetic tree, by replacing each multifurcation with multiple bifurcations.

Usage

```
multifurcations_to_bifurcations(tree, dummy_edge_length=0,
                                new_node_basename="node.",
                                new_node_start_index=NULL)
```

Arguments

`tree`	A tree of class "phylo".
`dummy_edge_length`	Non-negative numeric. Length to be used for new (dummy) edges when breaking multifurcations into bifurcations. Typically this will be 0, but can also be a positive number if zero edge lengths are not desired in the returned tree.
`new_node_basename`	Character. Name prefix to be used for added nodes (e.g. "node." or "new.node."). Only relevant if the input tree included node labels.
`new_node_start_index`	Integer. First index for naming added nodes. Can also be `NULL`, in which case this is set to Nnodes+1, where Nnodes is the number of nodes in the input tree.

Details

For each multifurcating node (i.e. with more than 2 children), all children but one will be placed on new bifurcating nodes, connected to the original node through one or more dummy edges.

The input tree need not be rooted, however descendance from each node is inferred based on the direction of edges in `tree$edge`. The input tree may include multifurcations (i.e. nodes with more than 2 children) as well as monofurcations (i.e. nodes with only one child). Monofurcations are kept in the returned tree.

All tips and nodes in the input tree retain their original indices, however the returned tree may include additional nodes and edges. Edge indices may change.

If `tree$edge.length` is missing, then all edges in the input tree are assumed to have length 1. The returned tree will include `edge.length`, with all new edges having length equal to `dummy_edge_length`.

Value

A list with the following elements:

`tree`	A new tree of class "phylo", containing only bifurcations (and monofurcations, if these existed in the input tree).
`old2new_edge`	Integer vector of length Nedges, mapping edge indices in the old tree to edge indices in the new tree.
`Nnodes_added`	Integer. Number of nodes added to the new tree.

Author(s)

Stilianos Louca

Examples

```
# generate a random multifurcating tree
Ntips = 1000
tree = generate_random_tree(list(birth_rate_intercept=1), Ntips, Nsplits=5)$tree

# expand multifurcations to bifurcations
new_tree = multifurcations_to_bifurcations(tree)$tree

# print summary of old and new tree
cat(sprintf("Old tree has %d nodes\n",tree$Nnode))
cat(sprintf("New tree has %d nodes\n",new_tree$Nnode))
```

`pick_random_tips` *Pick random subsets of tips on a tree.*

Description

Given a rooted phylogenetic tree, this function picks random subsets of tips by traversing the tree from root to tips, choosing a random child at each node until reaching a tip. Multiple random independent subsets can be generated if needed.

Usage

```
pick_random_tips( tree,
                  size              = 1,
                  Nsubsets          = 1,
                  with_replacement  = TRUE,
                  drop_dims         = TRUE)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`size`	Integer. The size of each random subset of tips.
`Nsubsets`	Integer. Number of independent subsets to pick.
`with_replacement`	Logical. If `TRUE`, each tip can be picked multiple times within a subset (i.e. are "replaced" in the urn). If `FALSE`, tips are picked without replacement in each subset. In that case, `size` must not be greater than the number of tips in the tree.
`drop_dims`	Logical, specifying whether to return a vector (instead of a matrix) if `Nsubsets==1`.

Details

If `with_replacement==TRUE`, then each child of a node is equally probable to be traversed and each tip can be included multiple times in a subset. If `with_replacement==FALSE`, then only children with at least one descending tip not included in the subset remain available for traversal; each available child of a node has equal probability to be traversed. In any case, it is always possible for separate subsets to include the same tips.

This random sampling algorithm differs from a uniform sampling of tips at equal probabilities; instead, this algorithm ensures that sister clades have equal probabilities to be picked (if `with_replacement==TRUE` or if `size`«Ntips).

The time required by this function per random subset decreases with the number of subsets requested.

Value

A 2D integer matrix of size Nsubsets x size, with each row containing indices of randomly picked tips (i.e. in 1,..,Ntips) within a specific subset. If `drop_dims==TRUE` and `Nsubsets==1`, then a vector is returned instead of a matrix.

Author(s)

Stilianos Louca

Examples

```
# generate random tree
Ntips = 1000
tree  = generate_random_tree(list(birth_rate_intercept=1),max_tips=Ntips)$tree
```

```
# pick random tip subsets
Nsubsets = 100
size     = 50
subsets = pick_random_tips(tree, size, Nsubsets, with_replacement=FALSE)

# count the number of times each tip was picked in a subset ("popularity")
popularities = table(subsets)

# plot histogram of tip popularities
hist(popularities,breaks=20,xlab="popularity",ylab="# tips",main="tip popularities")
```

---

reorder_tree_edges *Reorder tree edges in preorder or postorder.*

Description

Given a rooted tree, this function reorders the rows in `tree$edge` so that they are listed in preorder (root–>tips) or postorder (tips–>root) traversal.

Usage

```
reorder_tree_edges(tree, root_to_tips=TRUE,
                   depth_first_search=TRUE,
                   index_only=FALSE)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`root_to_tips`	Logical, specifying whether to sort edges in preorder traversal (root–>tips), rather than in postorder traversal (tips–>roots).
`depth_first_search`	Logical, specifying whether the traversal (or the reversed traversal, if `root_to_tips` is `FALSE`) should be in depth-first-search format rather than breadth-first-search format.
`index_only`	Whether the function should only return a vector listing the reordered row indices of the edge matrix, rather than a modified tree.

Details

This function does not change the tree structure, nor does it affect tip/node indices and names. It merely changes the order in which edges are listed in the matrix `tree$edge`, so that edges are listed in preorder or postorder traversal. Preorder traversal guarantees that each edge is listed before any of its descending edges. Likewise, postorder guarantees that each edge is listed after any of its descending edges.

With options `root_to_tips=TRUE` and `depth_first_search=TRUE`, this function is analogous to the function `reorder` in the ape package with option `order="cladewise"`.

The tree can include multifurcations (nodes with more than 2 children) as well as monofurcations (nodes with 1 child). This function has asymptotic time complexity O(Nedges).

Value

If `index_only==FALSE`, a tree object of class "phylo", with the rows in edge reordered such that they are listed in direction root–>tips (if `root_to_tips==TRUE`) or tips–>root. The vector `tree$edge.length` will also be updated in correspondence. Tip and node indices and names remain unchanged.

If `index_only=TRUE`, an integer vector (X) of size Nedges, listing the reordered row indices of `tree$edge`, i.e. such that `tree$edge[X,]` would be the reordered edge matrix.

Author(s)

Stilianos Louca

See Also

`get_tree_traversal_root_to_tips`

Examples

```
## Not run:
postorder_tree = reorder_tree_edges(tree, root_to_tips=FALSE)

## End(Not run)
```

`root_at_node` *Root or re-root a tree at a specific node.*

Description

Given a tree (rooted or unrooted) and a specific node, this function changes the direction of edges (`tree$edge`) such that the designated node becomes the root (i.e. has no incoming edges and all other tips and nodes descend from it). The number of tips and the number of nodes remain unchanged.

Usage

```
root_at_node(tree, new_root_node, update_indices=TRUE)
```

Arguments

`tree`	A tree object of class "phylo". Can be unrooted or rooted.
`new_root_node`	Character or integer specifying the name or index, respectively, of the node to be turned into root. If an integer, it must be between 1 and `tree$Nnode`. If a character, it must be a valid entry in `tree$node.label`.
`update_indices`	Logical, specifying whether to update the node indices such that the new root is the first node in the list (as is common convention). This will modify `tree$node.label` (if it exists) and also the node indices listed in `tree$edge`.

Details

The input tree may include an arbitrary number of incoming and outgoing edges per node (but only one edge per tip), and the direction of these edges can be arbitrary. Of course, the undirected graph defined by all edges must still be a valid tree. The asymptotic time complexity of this function is O(Nedges).

If `update_indices==FALSE`, then node indices remain unchanged. If `update_indices==TRUE` (default), then node indices are modified such that the new root is the first node (i.e. with index Ntips+1 in `edge` and with index 1 in `node.label`). This is common convention, but it may be undesirable if, for example, you are looping through all nodes in the tree and are only temporarily designating them as root. Setting `update_indices=FALSE` also reduces the computation required for rerooting. Tip indices always remain unchanged.

Value

A tree object of class "phylo", with the `edge` element modified such that the node `new_root_node` is root. The elements `tip.label`, `edge.length` and `root.edge` (if they exist) are the same as for the input tree. If `update_indices==FALSE`, then the element `node.label` will also remain the same.

Author(s)

Stilianos Louca

See Also

`root_via_outgroup`, `find_root`

Examples

```
# generate a random tree
Ntips = 100
tree = generate_random_tree(list(birth_rate_intercept=1),max_tips=Ntips)$tree

# reroot the tree at the 20-th node
new_root_node = 20
tree = root_at_node(tree, new_root_node, update_indices=FALSE)

# find new root index and compare with expectation
cat(sprintf("New root is %d, expected at %d\n",find_root(tree),new_root_node+Ntips))
```

`root_via_outgroup` *Root or re-root a tree based on an outgroup.*

Description

Given a tree (rooted or unrooted) and a specific tip or node ("outgroup"), this function changes the direction of edges (`tree$edge`) such that the outgroup's parent node becomes the root. The number of tips and the number of nodes remain unchanged.

Usage

```
root_via_outgroup(tree, outgroup, update_indices=TRUE)
```

Arguments

`tree`	A tree object of class "phylo". Can be unrooted or rooted.
`outgroup`	Character or integer specifying the name or index, respectively, of the outgroup tip/node. If an integer, it must be between 1 and Ntips+Nnodes. If a character, it must be a valid entry in `tree$tip.label`.
`update_indices`	Logical, specifying whether to update the node indices such that the new root is the first node in the list (as is common convention). This will modify `tree$node.label` (if it exists) and also the node indices listed in `tree$edge`.

Details

The input tree may include an arbitrary number of incoming and outgoing edges per node (but only one edge per tip), and the direction of these edges can be arbitrary. Of course, the undirected graph defined by all edges must still be a valid tree. The asymptotic time complexity of this function is O(Nedges).

If `update_indices==FALSE`, then node indices remain unchanged. If `update_indices==TRUE` (default), then node indices are modified such that the new root is the first node (i.e. with index Ntips+1 in edge and with index 1 in `node.label`). This is common convention, but it may be undesirable in some cases. Setting `update_indices=FALSE` also reduces the computation required for rerooting. Tip indices always remain unchanged.

Value

A tree object of class "phylo", with the edge element modified such that the outgroup's parent node is root. The elements `tip.label`, `edge.length` and `root.edge` (if they exist) are the same as for the input tree. If `update_indices==FALSE`, then the element `node.label` will also remain the same.

Author(s)

Stilianos Louca

See Also

`root_at_node`, `find_root`

Examples

```
# generate a random tree
Ntips = 100
tree = generate_random_tree(list(birth_rate_intercept=1),max_tips=Ntips)$tree

# reroot the tree using the 1st tip as outgroup
outgroup = 1
tree = root_via_outgroup(tree, outgroup, update_indices=FALSE)
```

```
# find new root index
cat(sprintf("New root is %d\n",find_root(tree)))
```

---

simulate_bm_model *Simulate a Brownian motion model for multivariate trait co-evolution.*

Description

Given a rooted phylogenetic tree and a Brownian motion (BM) model for the co-evolution of one or more continuous (numeric) unbounded traits, simulate random outcomes of the model on all nodes and/or tips of the tree. The function traverses nodes from root to tips and randomly assigns a multivariate state to each node or tip based on its parent's previously assigned state and the specified model parameters. The generated states have joint distributions consistent with the multivariate BM model. Optionally, multiple independent simulations can be performed using the same model.

Usage

```
simulate_bm_model(tree, diffusivity=NULL, sigma=NULL,
                  include_tips=TRUE, include_nodes=TRUE,
                  root_states=NULL, Nsimulations=1, drop_dims=TRUE)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`diffusivity`	Either NULL, or a single number, or a 2D quadratic positive definite symmetric matrix of size Ntraits x Ntraits. Diffusivity matrix ("D") of the multivariate Brownian motion model (in units trait^2/edge_length). The convention is that if the root's state is fixed, then the covariance matrix of a node's state at distance L from the root will be $2LD$ (see mathematical details below).
`sigma`	Either NULL, or a single number, or a 2D matrix of size Ntraits x Ndegrees, where Ndegrees refers to the degrees of freedom of the model. Noise-amplitude coefficients of the multivariate Brownian motion model (in units trait/sqrt(edge_length)). This can be used as an alternative way to specify the Brownian motion model instead of through the diffusivity D. Note that $sigma \cdot \sigma^T = 2D$ (see mathematical details below).
`include_tips`	Include random states for the tips. If `FALSE`, no states will be returned for tips.
`include_nodes`	Include random states for the nodes. If `FALSE`, no states will be returned for nodes.
`root_states`	Numeric matrix of size NR x Ntraits (where NR can be arbitrary), specifying the state of the root for each simulation. If NR is smaller than `Nsimulations`, values in `root_states` are recycled in rotation. If `root_states` is `NULL` or empty, then the root state is set to 0 for all traits in all simulations.
`Nsimulations`	Number of random independent simulations to perform. For each node and/or tip, there will be `Nsimulations` random states generated.

`drop_dims`	Logical, specifying whether singleton dimensions should be dropped from `tip_states` and `node_states`, if `Nsimulations==1` and/or Ntraits==1. If `drop_dims==FALSE`, then `tip_states` and `tip_nodes` will always be 3D matrices.

Details

The BM model for Ntraits co-evolving traits is defined by the stochastic differential equation

$$dX = \sigma \cdot dW$$

where W is a multidimensional Wiener process with Ndegrees independent components and σ is a matrix of size Ntraits x Ndegrees. Alternatively, the same model can be defined as a Fokker-Planck equation for the probability density ρ:

$$\frac{\partial \rho}{\partial t} = \sum_{i,j} D_{ij} \frac{\partial^2 \rho}{\partial x_i \partial x_j}.$$

The matrix D is referred to as the diffusivity matrix (or diffusion tensor), and $2D = \sigma \cdot \sigma^T$. Either `diffusivity` ($D$) or `sigma` ($\sigma$) may be used to specify the BM model, but not both.

If `tree$edge.length` is missing, each edge in the tree is assumed to have length 1. The tree may include multi-furcations (i.e. nodes with more than 2 children) as well as mono-furcations (i.e. nodes with only one child). The asymptotic time complexity of this function is O(Nedges*Nsimulations*Ntraits).

Value

A list with the following elements:

`tip_states`	Either `NULL` (if `include_tips==FALSE`), or a 3D numeric matrix of size Nsimulations x Ntips x Ntraits. The [r,c,i]-th entry of this matrix will be the state of trait i at tip c generated by the r-th simulation. If `drop_dims==TRUE` and `Nsimulations==1` and Ntraits==1, then `tip_states` will be a vector.
`node_states`	Either `NULL` (if `include_nodes==FALSE`), or a 3D numeric matrix of size Nsimulations x Nnodes x Ntraits. The [r,c,i]-th entry of this matrix will be the state of trait i at node c generated by the r-th simulation. If `drop_dims==TRUE` and `Nsimulations==1` and Ntraits==1, then `node_states` will be a vector.

Author(s)

Stilianos Louca

See Also

`simulate_ou_model`, `simulate_rou_model`, `simulate_mk_model`, `fit_bm_model`

Examples

```
# generate a random tree
tree = generate_random_tree(list(birth_rate_intercept=1),max_tips=10000)$tree

# Example 1: Scalar case
# - - - - - - - - - - - - - - - -
# simulate scalar continuous trait evolution on the tree
tip_states = simulate_bm_model(tree, diffusivity=1)$tip_states

# plot histogram of simulated tip states
hist(tip_states, breaks=20, xlab="state", main="Trait probability distribution", prob=TRUE)

# Example 2: Multivariate case
# - - - - - - - - - - - - - - - -
# simulate co-evolution of 2 traits with 3 degrees of freedom
Ntraits  = 2
Ndegrees = 3
sigma    = matrix(stats::rnorm(n=Ntraits*Ndegrees, mean=0, sd=1), ncol=Ndegrees)
tip_states = simulate_bm_model(tree, sigma=sigma, drop_dims=TRUE)$tip_states

# generate scatterplot of traits across tips
plot(tip_states[,1],tip_states[,2],xlab="trait 1",ylab="trait 2",cex=0.5)
```

simulate_diversification_model

Simulate a speciation/extinction model.

Description

Simulate a speciation/extinction model for diversity over time, corresponding to a tree generation model. Speciation (birth) and extinction (death) rates can each be constant or power-law functions of the number of extant species. For example,

$$B = I + F \cdot N^E,$$

where B is the birth rate, I is the intercept, F is the power-law factor, N is the current number of extant species and E is the power-law exponent. Note that currently only the deterministic version of the model can be simulated; for the stochastic version (Poisson process) see generate_random_tree.

Usage

```
simulate_diversification_model( times,
                                parameters      = list(),
                                Nsplits         = 2,
                                start_time      = NULL,
                                start_diversity = 1,
                                coalescent      = FALSE,
                                reverse         = FALSE,
                                include_Nbirths = FALSE,
                                max_runtime     = NULL)
```

Arguments

`times`	Numeric vector, listing the times for which to calculate diversities, as predicted by the model. Values must be in ascending order.
`parameters`	A named list specifying the birth-death model parameters, with one or more of the following entries: `birth_rate_intercept`: Non-negative number. The intercept of the Poissonian rate at which new species (tips) are added. In units 1/time. `birth_rate_factor`: Non-negative number. The power-law factor of the Poissonian rate at which new species (tips) are added. In units 1/time. `birth_rate_exponent`: Numeric. The power-law exponent of the Poissonian rate at which new species (tips) are added. Unitless. `death_rate_intercept`: Non-negative number. The intercept of the Poissonian rate at which extant species (tips) go extinct. In units 1/time. `death_rate_factor`: Non-negative number. The power-law factor of the Poissonian rate at which extant species (tips) go extinct. In units 1/time. `death_rate_exponent`: Numeric. The power-law exponent of the Poissonian rate at which extant species (tips) go extinct. Unitless.
`Nsplits`	Integer greater than 1. Nnumber of splits to assume per diversification event. For bifurcating trees this should be set to 2. Mostly for experimental purposes.
`start_time`	Numeric. If `reverse==FALSE` (see below), then this is the start time of the tree (<=`times[1]`). If `reverse==TRUE`, then this is the final time of the tree (>=`max(times)`). Can also be `NULL`, in which case it is set to the first or last value in `times` (depending on `reverse`).
`start_diversity`	Numeric. True diversity at `start_time`. For example, if `reverse==TRUE`, then this should be the final diversity of the tree.
`coalescent`	Logical, specifying whether the diversity corresponding to a coalescent tree (i.e. the tree spanning only extant tips) should be calculated. If `coalescent==TRUE` and the death rate is non-zero, then the returned diversities will generally be lower than the number of extant species calculated by the model at each time point.
`reverse`	Logical. If `TRUE`, then the tree model is simulated in backward time direction. In that case, `start_diversity` is interpreted as the known diversity at the last time point, and all diversities at previous time points are calculated based on the model. If `FALSE`, then the model is simulated in forward-time.
`include_Nbirths`	Logical. If `TRUE`, then the cumulative birth (speciation) and death (extinction) events (for each time point) are included as returned values. This comes at a moderate computational overhead.
`max_runtime`	Numeric. Maximum runtime (in seconds) allowed for the simulation. If this time is surpassed, the simulation aborts.

Value

A named list with the following elements:

`success`	Logical, indicating whether the simulation was successful. If the simulation aborted due to runtime constraints (option `max_runtime`), `success` will be `FALSE`.
`diversities`	Numeric vector of the same size as `times`, listing the diversity (if `coalescent==FALSE`) or the subset of diversity seen in the coalescent tree (if `coalescent==TRUE`), for each time point in `times`.
`Nbirths`	Numeric vector of the same size as `times`, listing the cumulative number of speciation (birth) events up to each time point. Only relevant if `include_Nspeciations==TRUE`.
`Ndeaths`	Numeric vector of the same size as `times`, listing the cumulative number of extinction (death) events up to each time point. Only relevant if `include_Nspeciations==TRUE`.

Author(s)

Stilianos Louca

See Also

`generate_random_tree`,

`count_clades_over_time`

Examples

```
# Generate a tree
max_time = 100
parameters = list(birth_rate_intercept  = 1,
                  birth_rate_factor     = 0,
                  birth_rate_exponent   = 0,
                  death_rate_intercept  = 0,
                  death_rate_factor     = 0,
                  death_rate_exponent   = 0)
tree = generate_random_tree(parameters,max_time=max_time)$tree

# Calculate diversity-vs-time curve for the tree
times = seq(from=0,to=0.99*max_time,length.out=10)
tree_diversities = count_clades_over_time(tree, times=times)$diversities

# simulate diversity curve based on deterministic model
model_diversities = simulate_diversification_model(times,parameters)$diversities

# compare diversities in the tree to the simulated ones
plot(tree_diversities,model_diversities,xlab="tree diversities",ylab="simulated diversities")
abline(a=0,b=1,col="#A0A0A0") # show diagonal for reference
```

`simulate_mk_model` *Simulate an Mk model for discrete trait evolution.*

Description

Given a rooted phylogenetic tree, a fixed-rates continuous-time Markov model for the evolution of a discrete trait ("Mk model", described by a transition matrix) and a probability vector for the root, simulate random outcomes of the model on all nodes and/or tips of the tree. The function traverses nodes from root to tips and randomly assigns a state to each node or tip based on its parent's previously assigned state and the specified transition rates between states. The generated states have joint distributions consistent with the Markov model. Optionally, multiple independent simulations can be performed using the same model.

Usage

```
simulate_mk_model(tree, Q, root_probabilities="stationary",
                  include_tips=TRUE, include_nodes=TRUE,
                  Nsimulations=1, drop_dims=TRUE)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`Q`	A numeric matrix of size Nstates x Nstates, storing the transition rates between states. In particular, every row must sum up to zero.
`root_probabilities`	Probabilities of the different states at the root. Either a character vector with value "stationary" or "flat", or a numeric vector of length Nstates, where Nstates is the number of possible states of the trait. In the later case, `root_probabilities` must be a valid probability vector, i.e. with non-negative values summing up to 1. "stationary" sets the probabilities at the root to the stationary distribution of `Q` (see `get_stationary_distribution`), while "flat" means that each state is equally probable at the root.
`include_tips`	Include random states for the tips. If `FALSE`, no states will be returned for tips.
`include_nodes`	Include random states for the nodes. If `FALSE`, no states will be returned for nodes.
`Nsimulations`	Number of random independent simulations to perform. For each node and/or tip, there will be `Nsimulations` random states generated.
`drop_dims`	Logical, specifying whether the returned `tip_states` and `node_states` (see below) should be vectors, if `Nsimulations==1`. If `drop_dims==FALSE`, then `tip_states` and `tip_nodes` will always be 2D matrices.

Details

For this function, the trait's states must be represented by integers within 1,..,Nstates, where Nstates is the total number of possible states. If the states are originally in some other format (e.g. characters or factors), you should map them to a set of integers 1,..,Nstates. These integers should correspond to row & column indices in the transition matrix `Q`. You can easily map any set of discrete states to integers using the function `map_to_state_space`.

If `tree$edge.length` is missing, each edge in the tree is assumed to have length 1. The tree may include multi-furcations (i.e. nodes with more than 2 children) as well as mono-furcations (i.e.

nodes with only one child). The time required per simulation decreases with the total number of requested simulations.

Value

A list with the following elements:

`tip_states`	Either `NULL` (if `include_tips==FALSE`), or a 2D integer matrix of size Nsimulations x Ntips with values in 1,..,Nstates, where Ntips is the number of tips in the tree and Nstates is the number of possible states of the trait. The [r,c]-th entry of this matrix will be the state of tip c generated by the r-th simulation. If `drop_dims==TRUE` and `Nsimulations==1`, then `tip_states` will be a vector.
`node_states`	Either `NULL` (if `include_nodes==FALSE`), or a 2D integer matrix of size Nsimulations x Nnodes with values in 1,..,Nstates, where Nnodes is the number of nodes in the tree. The [r,c]-th entry of this matrix will be the state of node c generated by the r-th simulation. If `drop_dims==TRUE` and `Nsimulations==1`, then `node_states` will be a vector.

Author(s)

Stilianos Louca

See Also

`exponentiate_matrix`, `get_stationary_distribution`, `simulate_bm_model`, `simulate_ou_model`, `simulate_rou_model`

Examples

```
# generate a random tree
tree = generate_random_tree(list(birth_rate_intercept=1),max_tips=1000)$tree

# simulate discrete trait evolution on the tree (5 states)
Nstates = 5
Q = get_random_mk_transition_matrix(Nstates, rate_model="ARD", max_rate=0.1)
tip_states = simulate_mk_model(tree, Q)$tip_states

# plot histogram of simulated tip states
barplot(table(tip_states)/length(tip_states), xlab="state")
```

`simulate_ou_model` *Simulate an Ornstein-Uhlenbeck model for continuous trait evolution.*

Description

Given a rooted phylogenetic tree and an Ornstein-Uhlenbeck (OU) model for the evolution of a continuous (numeric) trait, simulate random outcomes of the model on all nodes and/or tips of the tree. The function traverses nodes from root to tips and randomly assigns a state to each node or tip based on its parent's previously assigned state and the specified model parameters. The generated states have joint distributions consistent with the OU model. Optionally, multiple independent simulations can be performed using the same model.

Usage

```
simulate_ou_model(tree, stationary_mean, spread, decay_rate,
                  include_tips=TRUE, include_nodes=TRUE,
                  Nsimulations=1, drop_dims=TRUE)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`stationary_mean`	Numeric. The mean (center) of the stationary distribution of the OU model.
`spread`	Numeric. The standard deviation of the stationary distribution of the OU model.
`decay_rate`	Numeric. Exponential decay rate (stabilization rate) of the OU model (in units 1/edge_length_units).
`include_tips`	Include random states for the tips. If `FALSE`, no states will be returned for tips.
`include_nodes`	Include random states for the nodes. If `FALSE`, no states will be returned for nodes.
`Nsimulations`	Number of random independent simulations to perform. For each node and/or tip, there will be `Nsimulations` random states generated.
`drop_dims`	Logical, specifying whether the returned `tip_states` and `node_states` (see below) should be vectors, if `Nsimulations==1`. If `drop_dims==FALSE`, then `tip_states` and `tip_nodes` will always be 2D matrices.

Details

For each simulation, the state of the root is picked randomly from the stationary distribution of the OU model, i.e. from a normal distribution with mean = `stationary_mean` and standard deviation = `spread`.

If `tree$edge.length` is missing, each edge in the tree is assumed to have length 1. The tree may include multi-furcations (i.e. nodes with more than 2 children) as well as mono-furcations (i.e. nodes with only one child). The asymptotic time complexity of this function is O(Nedges*Nsimulations), where Nedges is the number of edges in the tree.

Value

A list with the following elements:

`tip_states`	Either `NULL` (if `include_tips==FALSE`), or a 2D numeric matrix of size Nsimulations x Ntips, where Ntips is the number of tips in the tree. The [r,c]-th entry of this matrix will be the state of tip c generated by the r-th simulation. If `drop_dims==TRUE` and `Nsimulations==1`, then `tip_states` will be a vector.
`node_states`	Either `NULL` (if `include_nodes==FALSE`), or a 2D numeric matrix of size Nsimulations x Nnodes, where Nnodes is the number of nodes in the tree. The [r,c]-th entry of this matrix will be the state of node c generated by the r-th simulation. If `drop_dims==TRUE` and `Nsimulations==1`, then `node_states` will be a vector.

Author(s)

Stilianos Louca

See Also

`simulate_bm_model`, `simulate_mk_model`, `simulate_rou_model`

Examples

```
# generate a random tree
tree = generate_random_tree(list(birth_rate_intercept=1),max_tips=10000)$tree

# simulate evolution of a continuous trait
tip_states = simulate_ou_model(tree, stationary_mean=10, spread=1, decay_rate=0.1)$tip_states

# plot histogram of simulated tip states
hist(tip_states, breaks=20, xlab="state", main="Trait probability distribution", prob=TRUE)
```

`simulate_rou_model` *Simulate a reflected Ornstein-Uhlenbeck model for continuous trait evolution.*

Description

Given a rooted phylogenetic tree and a reflected Ornstein-Uhlenbeck (ROU) model for the evolution of a continuous (numeric) trait, simulate random outcomes of the model on all nodes and/or tips of the tree. The ROU process is similar to the Ornstein-Uhlenbeck process (see `simulate_ou_model`), with the difference that the ROU process cannot fall below a certain value (its "reflection point"), which (in this implementation) is also its deterministic equilibrium point (Hu et al. 2015). The function traverses nodes from root to tips and randomly assigns a state to each node or tip based on its parent's previously assigned state and the specified model parameters. The generated states have joint distributions consistent with the ROU model. Optionally, multiple independent simulations can be performed using the same model.

Usage

```
simulate_rou_model(tree, reflection_point, spread, decay_rate,
                   include_tips=TRUE, include_nodes=TRUE,
                   Nsimulations=1, drop_dims=TRUE)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`reflection_point`	Numeric. The reflection point of the ROU model. In castor, this also happens to be the deterministic equilibrium of the ROU process (i.e. if the decay rate were infinite). For example, if a trait can only be positive (but arbitrarily small), then `reflection_point` may be set to 0.
`spread`	Numeric. The stationary standard deviation of the corresponding unreflected OU process.
`decay_rate`	Numeric. Exponential decay rate (stabilization rate) of the ROU process (in units 1/edge_length_units).
`include_tips`	Include random states for the tips. If `FALSE`, no states will be returned for tips.
`include_nodes`	Include random states for the nodes. If `FALSE`, no states will be returned for nodes.
`Nsimulations`	Number of random independent simulations to perform. For each node and/or tip, there will be `Nsimulations` random states generated.
`drop_dims`	Logical, specifying whether the returned `tip_states` and `node_states` (see below) should be vectors, if `Nsimulations==1`. If `drop_dims==FALSE`, then `tip_states` and `tip_nodes` will always be 2D matrices.

Details

For each simulation, the state of the root is picked randomly from the stationary distribution of the ROU model, i.e. from a one-sided normal distribution with mode = `reflection_point` and standard deviation = `stationary_std`.

If `tree$edge.length` is missing, each edge in the tree is assumed to have length 1. The tree may include multi-furcations (i.e. nodes with more than 2 children) as well as mono-furcations (i.e. nodes with only one child). The asymptotic time complexity of this function is O(Nedges*Nsimulations), where Nedges is the number of edges in the tree.

Value

A list with the following elements:

`tip_states`	Either `NULL` (if `include_tips==FALSE`), or a 2D numeric matrix of size Nsimulations x Ntips, where Ntips is the number of tips in the tree. The [r,c]-th entry of this matrix will be the state of tip c generated by the r-th simulation. If `drop_dims==TRUE` and `Nsimulations==1`, then `tip_states` will be a vector.
`node_states`	Either `NULL` (if `include_nodes==FALSE`), or a 2D numeric matrix of size Nsimulations x Nnodes, where Nnodes is the number of nodes in the tree. The [r,c]-th entry of this matrix will be the state of node c generated by the r-th simulation. If `drop_dims==TRUE` and `Nsimulations==1`, then `node_states` will be a vector.

Author(s)

Stilianos Louca

References

Y. Hu, C. Lee, M. H. Lee, J. Song (2015). Parameter estimation for reflected Ornstein-Uhlenbeck processes with discrete observations. Statistical Inference for Stochastic Processes. 18:279-291.

See Also

`simulate_ou_model`, `simulate_bm_model`, `simulate_mk_model`

Examples

```
# generate a random tree
tree = generate_random_tree(list(birth_rate_intercept=1),max_tips=10000)$tree

# simulate evolution of a continuous trait whose value is always >=1
tip_states = simulate_rou_model(tree, reflection_point=1, spread=2, decay_rate=0.1)$tip_states

# plot histogram of simulated tip states
hist(tip_states, breaks=20, xlab="state", main="Trait probability distribution", prob=TRUE)
```

`trim_tree_at_height` *Trim a rooted tree down to a specific height.*

Description

Given a rooted phylogenetic tree and a maximum allowed distance from the root (“height”), remove tips and nodes and shorten the remaining terminal edges so that the tree’s height does not exceed the specified threshold. This corresponds to drawing the tree in rectangular layout and trimming everything beyond a specific phylogenetic distance from the root. Tips or nodes at the end of trimmed edges are kept, and the affected edges are shortened.

Usage

```
trim_tree_at_height(tree, height = Inf, by_edge_count = FALSE)
```

Arguments

`tree`	A rooted tree of class "phylo". The root is assumed to be the unique node with no incoming edge.
`height`	Numeric, specifying the phylogenetic distance from the root at which to trim.
`by_edge_count`	Logical. Instead of considering edge lengths, consider edge counts as phylogenetic distance. This is the same as if all edges had length equal to 1.

Details

The input tree may include multi-furcations (i.e. nodes with more than 2 children) as well as mono-furcations (i.e. nodes with only one child).

Tip labels and uncollapsed node labels of the collapsed tree are inheritted from the original tree. Labels of tips that used to be nodes (i.e. of which all descendants have been removed) will be the node labels from the original tree. If the input tree has no node names, it is advised to first add node names to avoid `NA` in the resulting tip names.

Value

A list with the following elements:

`tree`	A new rooted tree of class "phylo", containing the trimmed tree.
`Nedges_trimmed`	Integer. Number of edges trimmed (shortened).
`Nedges_removed`	Integer. Number of edges removed.
`new2old_clade`	Integer vector of length equal to the number of tips+nodes in the trimmed tree, with values in 1,..,Ntips+Nnodes, mapping tip/node indices of the trimmed tree to tip/node indices in the original tree. In particular, `c(tree$tip.label,tree$node.label)[new2old_clade]` will be equal to: `c(trimmed_tree$tip.label,trimmed_tree$node.label)`.
`new2old_edge`	Integer vector of length equal to the number of edges in the trimmed tree, with values in 1,..,Nedges, mapping edge indices of the trimmed tree to edge indices in the original tree. In particular, `tree$edge.length[new2old_edge]` will be equal to `trimmed_tree$edge.length` (if edge lengths are available).

Author(s)

Stilianos Louca

Examples

```
# generate a random tree, include node names
tree = generate_random_tree(list(birth_rate_intercept=1),
                            max_time=1000,
                            node_basename="node.")$tree

# print number of tips
cat(sprintf("Simulated tree has %d tips\n",length(tree$tip.label)))

# trim tree at height 500
trimmed = trim_tree_at_height(tree, height=500)$tree

# print number of tips in trimmed tree
cat(sprintf("Trimmed tree has %d tips\n",length(trimmed$tip.label)))
```

`write_tree` *Write a tree in Newick (parenthetic) format.*

Description

Write a phylogenetic tree to a file or a string, in Newick (parenthetic) format. If the tree is unrooted, it is first rooted internally at the first node.

Usage

```
write_tree(tree, file="", append=FALSE, digits=10)
```

Arguments

`tree`	A tree of class "phylo".
`file`	An optional path to a file, to which the tree should be written. The file may be overwritten without warning. If left empty (default), then a string is returned representing the tree.
`append`	Logical, specifying whether the tree should be appended at the end of the file, rather than replacing the entire file (if it exists).
`digits`	Number of significant digits for writing edge lengths.

Details

This function is comparable to (but typically much faster than) the `ape` function `write.tree`.

Value

If `file==""`, then a string is returned containing the Newick representation of the tree. Otherwise, the tree is directly written to the file and no value is returned.

Author(s)

Stilianos Louca

Examples

```
# generate a random tree
tree = generate_random_tree(list(birth_rate_intercept=1),max_tips=100)$tree

# obtain a string representation of the tree in Newick format
Newick_string = write_tree(tree)
```

Index