Package: Claddis (via r-universe)

September 3, 2024

Type Package

Title Measuring Morphological Diversity and Evolutionary Tempo

Version 0.7.0

Date 2024-09-01

Maintainer Graeme T. Lloyd <graemetlloyd@gmail.com>

Depends ape, phytools, strap, R (>= 3.5.0)

Imports clipr, geoscale, graphics, grDevices, methods, multicool, partitions, stats, utils

Suggests rgl, testthat

Description Measures morphological diversity from discrete character data and estimates evolutionary tempo on phylogenetic trees. Imports morphological data from #NEXUS (Maddison et al. (1997) <doi:10.1093/sysbio/46.4.590>) format with read_nexus_matrix(), and writes to both #NEXUS and TNT format (Goloboff et al. (2008) <doi:10.1111/j.1096-0031.2008.00217.x>). Main functions are test_rates(), which implements AIC and likelihood ratio tests for discrete character rates introduced across Lloyd et al. (2012) <doi:10.1111/j.1558-5646.2011.01460.x>, Brusatte et al. (2014) <doi:10.1016/j.cub.2014.08.034>, Close et al. (2015) <doi:10.1016/j.cub.2015.06.047>, and Lloyd (2016) <doi:10.1111/bij.12746>, and calculate_morphological_distances(), which implements multiple discrete character distance metrics from Gower (1971) <doi:10.2307/2528823>, Wills (1998) <doi:10.1006/bijl.1998.0255>, Lloyd (2016) <doi:10.1111/bij.12746>, and Hopkins and St John (2018) <doi:10.1098/rspb.2018.1784>. This also includes the GED correction from Lehmann et al. (2019) <doi:10.1111/pala.12430>. Multiple functions implement morphospace plots: plot_chronophylomorphospace() implements Sakamoto and Ruta (2012) <doi:10.1371/journal.pone.0039752>, plot_morphospace() implements Wills et al. (1994) <doi:10.1017/S009483730001263X>, plot_changes_on_tree() implements Wang and Lloyd (2016) <doi:10.1098/rspb.2016.0214>, and plot_morphospace_stack()

implements Foote (1993) <doi:10.1017/S0094837300015864>. Other functions include safe_taxonomic_reduction(), which implements Wilkinson (1995) <doi:10.1093/sysbio/44.4.501>, map_dollo_changes() implements the Dollo stochastic character mapping of Tarver et al. (2018) <doi:10.1093/gbe/evy096>, and estimate_ancestral_states() implements the ancestral state options of Lloyd (2018) <doi:10.1111/pala.12380>. calculate_tree_length() and reconstruct_ancestral_states() implements the generalised algorithms from Swofford and Maddison (1992; no doi).

Encoding UTF-8

License GPL (>=2)

LazyData yes

ByteCompile yes

RoxygenNote 7.3.1

Repository https://phylotastic.r-universe.dev

RemoteUrl https://github.com/graemetlloyd/Claddis

RemoteRef HEAD

RemoteSha 66d0d06037d96601a78dd53b07362fa95f536d48

Contents

Claddis-package 4
add_polymorphisms_to_costmatrix 5
add_uncertainties_to_costmatrix 13
align_matrix_block 16
assign_taxa_to_bins 18
bin_changes 20
bin_character_completeness 21
bin_edge_lengths 23
build_cladistic_matrix 24
calculate_g 26
calculate_gmax 32
calculate_kardashian_index 37
calculate_morphological_distances 39
calculate_MPD 43
calculate_tree_length 44
calculate_WMPD 49
check_cladisticMatrix 51
check_costMatrix 51
check_stateGraph 52
check_taxonGroups 53
check_timeBins 54
classify_costmatrix 55
compactify_cladistic_matrix 66

convert_adjacency_matrix_to_costmatrix . . . 67
convert_costmatrix_to_stategraph . . . 69
convert_stategraph_to_costmatrix . . . 71
convert_state_tree_to_adjacency_matrix . . . 74
count_cherries . . . 76
date_nodes . . . 77
day_2016 . . . 78
drop_time_tip . . . 78
estimate_ancestral_states . . . 79
estimate_squared_change_ancestors . . . 82
find_costmatrix_minimum_span . . . 85
find_descendant_edges . . . 87
find_linked_edges . . . 88
find_minimum_spanning_edges . . . 89
find_mrca . . . 90
find_shortest_costmatrix_path . . . 91
find_stategraph_minimum_span . . . 93
find_time_bin_midpoints . . . 95
find_unique_trees . . . 97
fix_costmatrix . . . 98
fix_root_time . . . 100
gauthier_1986 . . . 101
is.cladisticMatrix . . . 101
is.costMatrix . . . 102
is.stateGraph . . . 103
is.taxonGroups . . . 104
is.timeBins . . . 105
is_graph_connected . . . 106
locate_bracket_positions . . . 108
make_costmatrix . . . 109
make_labels . . . 117
map_dollo_changes . . . 118
match_tree_edges . . . 119
michaux_1989 . . . 120
ordinate_cladistic_matrix . . . 121
partition_time_bins . . . 124
permute_all_polymorphisms . . . 125
permute_all_treeshape_labellings . . . 126
permute_all_uncertainties . . . 127
permute_connected_graphs . . . 128
permute_costmatrices . . . 130
permute_graph_splits . . . 132
permute_restricted_compositions . . . 133
permute_tipstates . . . 134
permute_treeshapes . . . 140
plot_changes_on_tree . . . 141
plot_chronophylomorphospace . . . 143
plot_morphospace . . . 145

plot_morphospace_stack . . . 149
plot_multi_morphospace . . . 153
plot_rates_character . . . 155
plot_rates_time . . . 157
plot_rates_tree . . . 159
print.cladisticMatrix . . . 161
print.costMatrix . . . 162
print.stateGraph . . . 163
print.taxonGroups . . . 164
print.timeBins . . . 165
prune_cladistic_matrix . . . 166
read_nexus_matrix . . . 168
reconstruct_ancestral_states . . . 169
safe_taxonomic_reduction . . . 172
safe_taxonomic_reinsertion . . . 173
split_out_subgraphs . . . 176
test_rates . . . 177
trim_marginal_whitespace . . . 187
trim_matrix . . . 188
write_nexus_matrix . . . 189
write_tnt_matrix . . . 190

Index **192**

`Claddis-package` *Measuring Morphological Diversity and Evolutionary Tempo Measures morphological diversity from discrete character data and estimates evolutionary tempo on phylogenetic trees.*

Description

Measures morphological diversity from discrete character data and estimates evolutionary tempo on phylogenetic trees. Imports morphological data from #NEXUS (Maddison et al. (1997) doi:10.1093/sysbio/46.4.590) format with read_nexus_matrix(), and writes to both #NEXUS and TNT format (Goloboff et al. (2008) doi:10.1111/j.10960031.2008.00217.x). Main functions are test_rates(), which implements AIC and likelihood ratio tests for discrete character rates introduced across Lloyd et al. (2012) doi:10.1111/j.15585646.2011.01460.x, Brusatte et al. (2014) doi:10.1016/j.cub.2014.08.034, Close et al. (2015) doi:10.1016/j.cub.2015.06.047, and Lloyd (2016) doi:10.1111/bij.12746, and calculate_morphological_distances(), which implements multiple discrete character distance metrics from Gower (1971) doi:10.2307/2528823, Wills (1998) doi:10.1006/bijl.1998.0255, Lloyd (2016) doi:10.1111/bij.12746, and Hopkins and St John (2018) doi:10.1098/rspb.2018.1784. This also includes the GED correction from Lehmann et al. (2019) doi:10.1111/pala.12430. Multiple functions implement morphospace plots: plot_chronophylomorphospace() implements Sakamoto and Ruta (2012) doi:10.1371/journal.pone.0039752, plot_morphospace() implements Wills et al. (1994) doi:10.1017/S009483730001263X, plot_changes_on_tree() implements Wang and Lloyd (2016) doi:10.1098/rspb.2016.0214, and plot_morphospace_stack() implements Foote (1993) doi:10.1017/S0094837300015864. Other functions include safe_taxonomic_reduction(), which implements Wilkinson (1995) doi:10.1093/sysbio/44.4.501, map_dollo_changes() implements the Dollo stochastic

character mapping of Tarver et al. (2018) doi:10.1093/gbe/evy096, and estimate_ancestral_states() implements the ancestral state options of Lloyd (2018) doi:10.1111/pala.12380. calculate_tree_length() and reconstruct_ancestral_states() implements the generalised algorithms from Swofford and Maddison (1992; no doi).

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Lloyd, G. T., 2016. Estimating morphological diversity and tempo with discrete character-taxon matrices: implementation, challenges, progress, and future directions. *Biological Journal of the Linnean Society*, **118**, 131-151.

Examples

```
# Get morphological distances for Michaux (1989) data set:
distances <- calculate_morphological_distances(cladistic_matrix = michaux_1989)

# Show distances:
distances
```

---

add_polymorphisms_to_costmatrix

Adds polymorphisms to a costmatrix

Description

Given a costmatrix and choice of method, adds polymorphic transitions to a costmatrix.

Usage

```
add_polymorphisms_to_costmatrix(
  costmatrix,
  polymorphism_costs = "additive",
  polymorphism_geometry = "simplex",
  polymorphism_distance = "euclidean",
  message = TRUE
)
```

Arguments

`costmatrix` An object of class "costMatrix".

`polymorphism_costs` The method to use to assign costs to the transitions to and from polymorphic states. Must be one of `"additive"` (the default), `"geometric"`, `"maddison"`, or `"stratigraphic"`. See details.

`polymorphism_geometry`
: If using `polymorphism_costs = "geometric"`, the type of geometry to use. Must be one of `"hypercube"`, `"hypersphere"`, or `"simplex"` (the default). See details.

`polymorphism_distance`
: If using `polymorphism_costs = "geometric"`, the type of distance to use. Must be one of `"euclidean"` (the default), `"great_circle"`, or `"manhattan"`. See details.

`message`
: Logical indicating whether (`TRUE`, the default) or not (`FALSE`) to provide messages about the process of the function.

Details

Polymorphisms - the presence of two or more discrete character states in an operational taxonomic unit - represent perhaps the most complex problem for phylogenetic inference and ancestral state estimation. This is at least in part due to the varied nature their meaning can have (Swofford and Maddison 1992). For example, they may represent true variance within a species or composite variance in some higher taxon represented by a single OTU (something to be avoided if at all possible). Consequently, they cannot - and probably should not - be treated the same by all software in all situations.

One solution to the problem of polymorphisms is to pretend they do not exist. This is the approach used by common software such as TNT (Goloboff et al. 2008; Goloboff and Catalano 2016), which treat polymorphisms as uncertainties instead. I.e., they assume only one state is the "true" value and use parsimony to (implicitly) estimate what this state is. This approach can be adopted in Claddis too by simply selecting options that treat polymorphisms as uncertainties instead. However, misrepresenting true polymorphisms as uncertainties can leads to incorrect estimates of the amount of evolution (tree length under parsimony; Nixon and Davis 1991) and can disallow, potentially inappropriately, polymorphic values as ancestral state estimates (reconstructions). Claddis therefore offers options to treat polymorphisms as "real".

There is no single agreed method on dealing with "real" polymorphisms, although many have been proposed (Wiens 1999). Due to the constraints of how Claddis works the only options implemented here - aside from simple approaches like treating them as missing data - are those that work in the costmatrix setting. These are a mixture of novel approaches, extensions of other approaches, or taken directly from Maddison and Maddison (1987).

Generally speaking the costmatrix setting makes explicit the cost of individual transitions (e.g., from the single state 0 to the polymorphic state 1 *and* 2) and hence the only decision to make is how these costs should be assigned. Here three approaches are offered: 1) the generalised additive approach, 2) the geometric approach (for unordered characters), and 3) the Maddison approach (for ordered characters). These are described in more detail below.

The additive approach

One way of conceiving of polymorphic tip states is to conceptually "split" a terminal branch into multiple terminals, each with a single state from the polymorphism at its' tip. For example, a branch leading to the state "0&1" would become two branches, one leading to state 0 and one to state 1. Character costs can then be calculated for each branch separately and then, in order to "merge" these into a single branch, the sum of these costs can be taken. This approach was first suggested in Hoyal Cuthill and Lloyd (in prep) and is here termed the "additive" approach (`polymorphism_costs =`

"additive"). A major advantage to this approach is that it's rules translate across the broadest range of character types. However, there is no logical interpretation of polymorphic ancestral states. (I.e., splitting an ancestral node into two or more nodes is not possible in the same way.) Consequently, transitions "from" polymorphic states are assigned a cost of infinity, precluding them from consideration as ancestral values.

An example of the additive approach, applied to an unordered character with three states (0, 1, and 2) would thus appear as the following costmatrix:

	0	1	2	0&1	0&2	1&2	0&1&2
0	0	1	1	1	1	2	2
1	1	0	1	1	2	1	2
2	1	1	0	2	1	1	2
0&1	Inf	Inf	Inf	0	Inf	Inf	Inf
0&2	Inf	Inf	Inf	Inf	0	Inf	Inf
1&2	Inf	Inf	Inf	Inf	Inf	0	Inf
0&1&2	Inf	Inf	Inf	Inf	Inf	Inf	0

The geometric approach

An alternative approach based on an idea first suggested by Maddison and Maddison (1987) does allow polymorphisms to be treated as valid ancestral states. Here the conceptual idea is that transitions involve turning "on" or "off" particular states. Thus to go from state 1 to state 0&2 would involve turning "on" states 0 and 2 turning "off" state 1, giving a total cost of three. This approach is only suitable where costs between single states are equal and symmetric - i.e., the character is unordered.

An example of this approach, applied to an unordered character with three states (0, 1, and 2) would thus appear as the following costmatrix:

	0	1	2	0&1	0&2	1&2	0&1&2
0	0	2	2	1	1	3	2
1	2	0	2	1	3	1	2
2	2	2	0	3	1	1	2
0&1	1	1	3	0	2	2	1

```
|  0&2  |  1  |  3  |  1  |  2  |  0  |  2  |   1   |
-----------------------------------------------------
|  1&2  |  3  |  1  |  1  |  2  |  2  |  0  |   1   |
-----------------------------------------------------
| 0&1&2 |  2  |  2  |  2  |  1  |  1  |  1  |   0   |
-----------------------------------------------------
```

Note: to maintain a base cost of one the original single-state-to-single-state transitions are doubled from one to two. In order to avoid this affecting tree lengths the character weight is halved.

This Maddison and Maddison (1987) approach is extended here by drawing on an analogy, namely that this cost assignment is identical to considering a hypercube whose vertices lie at one of two (presence or absence) positions on orthogonal axes representing individual states. Thus the polymorphism 0&2 is at the coordinates (0 = present, 1 = absent, 2 = present). Here costs can be assigned by taking the minimum Manhattan distance between each pair of vertices. NB: here the origin (absence of all states) is unoccupied, but this does not affect the assignment of costs.

This analogy is why the approach is here termed the "geometric" approach (`polymorphism_costs = "geometric"`). Importantly, although the example above only represents the regular cube (three states = three dimensions) the approach translates to any number of states (i.e., any number of dimensions) without loss of generality, and hence this particular option is properly termed a hypercube (i.e., `polymorphism_geometry = "hypercube"` and `polymorphism_distance = "manhattan"`).

In of itself this does not change what was proposed by Maddison and Maddison. However, the geometric setting allows us to consider both different shapes (topologies) and different distance measures, as long as the same generality to higher dimensions holds. Two additional shapes are offered here. These are the hypersphere (the N-dimensional circle) and the simplex (the N-dimensional equilateral triangle). In addition, two distances are also offered. These are Euclidean (the straight line distance between two points) and Great circle (the shortest distance across the "surface" of a hypersphere) distances.

These distances are intended to match respective straights (hypercube and Manhattan, hypersphere and Great Circle, simplex and Euclidean), but any combination is permitted. Furthermore, and as above, all costs are rescaled such that the base transition cost is always one with the character weight modified accordingly.

The Maddison approach

Although the above approach is based on Maddison and Maddison (1987) the Maddison name (`polymorphism_costs = "maddison"`) is here reserved for an approach proposed by the same authors (Maddison and Maddison 2000) for use with ordered characters. The concept of this approach shares the idea of "switching" states on and off but here adds the idea of adjacency of switches. In other words, intermediate states (the essential element of an ordered character) are additional switches that must be turned "on" and "off" again in order to "reach" other states. Thus to go from state 0 to state 0 and 2 for the linear ordered character 0-1-2 we would have to turn "on" state to reach state "2", which is also turned on. Then we would turn off state "1", meaning total of three switches were flipped which would be the cost. Note: that here we are not turning "off" state 0 as it is present at both ends of the transition.

Extending our three state linear ordered example to all possible transitions we get the costmatrix:

```
---------------------------------------------
|  0  |  1  |  2  | 0&1 | 0&2 | 1&2 | 0&1&2 |
```

0	0	2	4	1	3	3	2
1	2	0	2	1	3	1	2
2	4	2	0	3	3	1	2
0&1	1	1	3	0	2	2	1
0&2	3	3	3	2	0	2	1
1&2	3	1	1	2	2	0	1
0&1&2	2	2	2	1	1	1	0

NB: Again the single-state-to-single-state cost is modified (doubled) meaning the character weight is halved.

The stratigraphic approach

The stratigraphic case seems like it ought to follow the same rule as irreversible characters (see below), but here "polymorphisms" have a different logical meaning. Specifically they encode the presence of a taxon in multiple time units. Thus, transitions *to* a stratigraphic polymorphism are based on the oldest state. By contrast, polymorphisms can never be ancestral states as there would be no basis to choose between (e.g.) state 0 and state 0 and 1.

The option `polymorphism_costs = "stratigraphic"` is only available where `costmatrix$type` is `"stratigraphic"` and, where `costmatrix$type` is `"stratigraphic"`, `polymorphism_costs = "stratigraphic"` is the only option.

Additional considerations

Aside from these three options there are some further considerations the user should make with respect to some more complex character types. These are discussed below under appropriate headers.

Dollo and irreversible polymorphisms

At first blush Dollo and irreversible (Camin-Sokal) characters appear to confound costmatrix construction. More specifically, for a Dollo character, the state 0 and 1 would suggest state 1 was acquired previously, but lost in some member(s) of the OTU. For an irreversible character the opposite would be true, and we would assume that state 1 was acquired independently by some member(s) of the OTU and state 0 was acquired previously. It is not possible to code either of these outcomes under the geometric or Maddison approaches. However, the additive approach perfectly captures this nuance. Thus tree length calculation and ancestral state estimation for Dollo and Camin-Sokal characters is possible under the additive approach and only the possibility of polymorphic ancestral states is excluded.

Transitions between uncertainties and polymorphisms

Because Claddis also permits users to include uncertainties in costmatrices transitions between these and polymorphisms must also be considered. Here this is automated by using the "minimum rule". In practice this means a transition from, for example, state 0 and 1 to state 1 or 2 will take the lowest cost of the two options (from state 0 and 1 to state 1 *or* from state 0 and 1 to state 2). In the case of

the additive approach these costs will always be infinite. In all cases transitions from uncertainties are assigned infinite cost as these are not considered valid ancestral states.

Polymorphism limits and gap-weighted characters

Even where polymorphisms may be appropriate (i.e., for some ordered and unordered characters) they still represent a major increase to costmatrix size that can cause memory limits to be hit. Specifically, and as shown in the hypercube example above, there will be as many possible states as N^2 - 1, where N is the number of single (i.e., non-polymorphic) states and the minus one indicates the exclusion of the unrepresentable origin value. Thus the size of costmatrix required grows very quickly:

```
------------------------------
| N states | Costmatrix size |
------------------------------
|     2    |      3 x 3      |
|     3    |      7 x 7      |
|     4    |     15 x 15     |
|     5    |     31 x 31     |
|     6    |     63 x 63     |
|     7    |    127 x 127    |
|     8    |    255 x 255    |
|     9    |    511 x 511    |
|    10    |   1023 x 1023   |
|    11    |   2047 x 2047   |
|    12    |   4095 x 4095   |
|    13    |   8191 x 8191   |
|    14    |  16383 x 16383  |
------------------------------
```

Because of this, the function will become extremely slow for these higher values and here is hard-capped at no more than fourteen states. Consequently any gap-weighted characters are also most likely inappropriate for polymorphism use (as well as being too memory intensive).

Value

An object of class "costMatrix".

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Goloboff, P. A. and Catalano, S. A., 2016. TNT version 1.5, including a full implementation of phylogenetic morphometrics/ *Cladistics*, **32**. 221-238

Goloboff, P., Farris, J. and Nixon, K., 2008. TNT, a free program for phylogenetic analysis. *Cladistics*, **24**, 774-786.

Hoyal Cuthill and Lloyd, in prep.

Maddison, W. P. and Maddison, D. R., 1987. MacClade 2.1, computer program and manual. Cambridge, Massachusetts.

Nixon, K. C. and Davis, J. I., 1991. Polymorphic taxa, missing values and cladistic analysis. *Cladistics*, **7**, 233-241.

Swofford, D. L. and Maddison, W. P., 1992. Parsimony, character-state reconstructions, and evolutionary inferences. In R. L. Mayden (ed.), *Systematics, Historical Ecology, and North American Freshwater Fishes*. Stanford University Press, Stanford. pp187-223.

Wiens, J. J., 1999. Polymorphism in systematics and comparative biology. *Annual Review of Ecology and Systematics*, **30**, 327-362.

Examples

```
# Generate an example three-state unordered character costmatrix:
unordered_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 2,
  character_type = "unordered"
)

# Generate an example three-state ordered character costmatrix:
ordered_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 2,
  character_type = "ordered"
)

# Generate an example three-state ordered character costmatrix with uncertainties already included:
ordered_uncertainty_costmatrix <- list(
  size = 7,
  n_states = 3,
  single_states = c("0", "1", "2"),
  type = "ordered",
  costmatrix = matrix(
    data = c(
      0, 1, 2, 0, 0, 1, 0,
      1, 0, 1, 0, 1, 0, 0,
      2, 1, 0, 1, 0, 0, 0,
      Inf, Inf, Inf, 0, Inf, Inf, Inf,
      Inf, Inf, Inf, Inf, 0, Inf, Inf,
      Inf, Inf, Inf, Inf, Inf, 0, Inf,
      Inf, Inf, Inf, Inf, Inf, Inf, 0
    ),
    nrow = 7,
    byrow = TRUE,
    dimnames = list(
      c(as.character(x = 0:2), "0/1", "0/2", "1/2", "0/1/2"),
      c(as.character(x = 0:2), "0/1", "0/2", "1/2", "0/1/2")
    )
  ),
  symmetry = "Symmetric",
  includes_polymorphisms = FALSE,
```

```
  polymorphism_costs = "additive",
  polymorphism_geometry = "simplex",
  polymorphism_distance = "euclidean",
  includes_uncertainties = TRUE,
  pruned = FALSE,
  dollo_penalty = 999,
  base_age = 1,
  weight = 1
)
class(ordered_uncertainty_costmatrix) <- "costMatrix"

# Generate an example five-state stratigraphic character costmatrix:
stratigraphic_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 4,
  character_type = "stratigraphy",
  state_ages = c(0, 1.3, 5.3, 8.0, 11.1)
)

# Add polymorphisms to unordered costmatrix using additive method:
unordered_costmatrix_additive_polymorphisms <- add_polymorphisms_to_costmatrix(
  costmatrix = unordered_costmatrix,
  polymorphism_costs = "additive"
)

# Show unordered costmatrix using additive method:
unordered_costmatrix_additive_polymorphisms$costmatrix

# Add polymorphisms to unordered costmatrix using geometric simplex method:
unordered_costmatrix_simplex_polymorphisms <- add_polymorphisms_to_costmatrix(
  costmatrix = unordered_costmatrix,
  polymorphism_costs = "geometric",
  polymorphism_geometry = "simplex",
  polymorphism_distance = "euclidean"
)

# Show unordered costmatrix using geometric simplex method:
unordered_costmatrix_simplex_polymorphisms$costmatrix

# Add polymorphisms to unordered costmatrix using geometric hypercube method:
unordered_costmatrix_hypercube_polymorphisms <- add_polymorphisms_to_costmatrix(
  costmatrix = unordered_costmatrix,
  polymorphism_costs = "geometric",
  polymorphism_geometry = "hypercube",
  polymorphism_distance = "manhattan"
)

# Show unordered costmatrix using geometric hypercube method:
unordered_costmatrix_hypercube_polymorphisms$costmatrix

# Add polymorphisms to ordered costmatrix using additive method:
ordered_costmatrix_additive_polymorphisms <- add_polymorphisms_to_costmatrix(
  costmatrix = ordered_costmatrix,
```

```
  polymorphism_costs = "additive"
)

# Show ordered costmatrix using additive method:
ordered_costmatrix_additive_polymorphisms$costmatrix

# Add polymorphisms to ordered costmatrix using maddison method:
ordered_costmatrix_maddison_polymorphisms <- add_polymorphisms_to_costmatrix(
  costmatrix = ordered_costmatrix,
  polymorphism_costs = "maddison"
)

# Show ordered costmatrix using Maddison method:
ordered_costmatrix_maddison_polymorphisms$costmatrix

# Add polymorphisms to ordered uncertainty costmatrix using additive method:
ordered_uncertainty_costmatrix_additive_polymorphisms <- add_polymorphisms_to_costmatrix(
  costmatrix = ordered_uncertainty_costmatrix,
  polymorphism_costs = "additive"
)

# Show costmatrix, with polymorphism-to-uncertainty transitions as infinities:
ordered_uncertainty_costmatrix_additive_polymorphisms$costmatrix

# Add polymorphisms to ordered uncertainty costmatrix using Maddison method:
ordered_uncertainty_costmatrix_maddison_polymorphisms <- add_polymorphisms_to_costmatrix(
  costmatrix = ordered_uncertainty_costmatrix,
  polymorphism_costs = "maddison"
)

# Show costmatrix, with polymorphism-to-uncertainty transitions
# interpolated using minimum cost rule:
ordered_uncertainty_costmatrix_maddison_polymorphisms$costmatrix

# Add polymorphisms to stratigraphic costmatrix using stratigraphic method:
stratigraphic_costmatrix_polymorphisms <- add_polymorphisms_to_costmatrix(
  costmatrix = stratigraphic_costmatrix,
  polymorphism_costs = "stratigraphic"
)

# Show stratigraphic costmatrix using stratigraphic method:
stratigraphic_costmatrix_polymorphisms$costmatrix
```

---

add_uncertainties_to_costmatrix

Adds uncertainties to a costmatrix

Description

Adds uncertainty transitions to a costmatrix.

Usage

```
add_uncertainties_to_costmatrix(costmatrix, message = TRUE)
```

Arguments

`costmatrix`	An object of class "costMatrix".
`message`	Logical indicating whether (`TRUE`, the default) or not (`FALSE`) to provide messages about the process of the function.

Details

Under a parsimony algorithm uncertainties indicate that the observed state cannot be restricted to a single value but some finite set of values. In #NEXUS format these are indicated with curly braces, , and within Claddis by a slash. For example, state 0 or 1 would be 0/1.

For most operations (calculating the length of a tree, estimating ancestral states) there is no need to formally add uncertainties to a costmatrix. Instead, the uncertainty can be established via setting up the tip states in the Swofford and Maddison (1992) approach. However, there are some usages within Claddis where it is useful to include them in the costmatrix itself, for example calculating minimum step counts for homoplasy metrics.

Here, uncertainties are added to a costmatrix in a similar way to polymorphisms. I.e., all possible uncertainty combinations are calculated and then the associated transition cost calculated and inserted. Note that here only one approach for calculating this cost is applied - the minimu rule of Hoyal Cuthill and Lloyd (in prep.).

Importantly costs of transitioning *from* uncertainties are assigned infinite cost. There aree two main reasons for this. Firstly, under parsimony and the minimum rule allowing uncertainties as ancestral states would lead to a tree length of zero and an ancestral state for every internal node that includes every possible state being favoured, regardless of the states at the tips. In other words, it would have no analytical value. Secondly, assigning uncertainty of ancestral states is best done by generating all (single or polymorphic) most parsimonious reconstructions and summarising these.

Note that in many practical cases an uncertainty involving all possible states - e.g., the uncertianty 0/1/2 for a three-state character - operates the same as a missing value (NA). However, it may still be useful in some circumstances. For example, to denote that the coding is definitively applicable (as opposed to inapplicable) or to properly record uncertainty rather than missing values for summary statistical purposes.

Finally, it should be noted that uncertainties are the only allowable breaking of the costmatrix rule that all off-diagonal transition costs must be positive. I.e., costs of zero from a single or polymorphic state to an uncertainty state are permitted.

Value

An object of class "costMatrix".

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Hoyal Cuthill and Lloyd, in prep.

Swofford, D. L. and Maddison, W. P., 1992. Parsimony, character-state reconstructions, and evolutionary inferences. In R. L. Mayden (ed.), *Systematics, Historical Ecology, and North American Freshwater Fishes*. Stanford University Press, Stanford. pp187-223.

Examples

```
# Generate an example three-state unordered character costmatrix:
# Generate an example three-state unordered character costmatrix:
unordered_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 2,
  character_type = "unordered"
)

# Generate an example three-state ordered character costmatrix:
ordered_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 2,
  character_type = "ordered"
)

# Generate an example three-state ordered character costmatrix with polymorphism included:
unordered_polymorphism_costmatrix <- list(
  size = 7,
  n_states = 3,
  single_states = c("0", "1", "2"),
  type = "unordered",
  costmatrix = matrix(
    data = c(
      0, 2, 2, 1, 1, 3, 2,
      2, 0, 2, 1, 3, 1, 2,
      2, 2, 0, 3, 1, 1, 2,
      1, 1, 3, 0, 2, 2, 1,
      1, 3, 1, 2, 0, 2, 1,
      3, 1, 1, 2, 2, 0, 1,
      2, 2, 2, 1, 1, 1, 0
    ),
    nrow = 7,
    byrow = TRUE,
    dimnames = list(
      c(as.character(x = 0:2), "0&1", "0&2", "1&2", "0&1&2"),
      c(as.character(x = 0:2), "0&1", "0&2", "1&2", "0&1&2")
    )
  ),
  symmetry = "Symmetric",
  includes_polymorphisms = TRUE,
  polymorphism_costs = "geometric",
  polymorphism_geometry = "hypercube",
  polymorphism_distance = "manhattan",
```

```
  includes_uncertainties = FALSE,
  pruned = FALSE,
  dollo_penalty = 999,
  base_age = 1,
  weight = 0.5
)
class(unordered_polymorphism_costmatrix) <- "costMatrix"

# Add uncertainties to unordered costmatrix:
unordered_costmatrix_uncertainties <- add_uncertainties_to_costmatrix(
  costmatrix = unordered_costmatrix
)

# Show unordered costmatrix with uncertainties included:
unordered_costmatrix_uncertainties$costmatrix

# Add uncertainties to ordered costmatrix:
ordered_costmatrix_uncertainties <- add_uncertainties_to_costmatrix(
  costmatrix = ordered_costmatrix
)

# Show ordered costmatrix with uncertainties included:
ordered_costmatrix_uncertainties$costmatrix

# Add uncertainties to unordered costmatrix with polymorphisms:
unordered_polymorphism_costmatrix_uncertainties <- add_uncertainties_to_costmatrix(
  costmatrix = unordered_polymorphism_costmatrix
)

# Show unordered polymorphism costmatrix with uncertainties included:
unordered_polymorphism_costmatrix_uncertainties$costmatrix
```

---

align_matrix_block *Aligns a phylogenetic matrix block*

Description

Given a block of taxa and characters aligns text so each character block begins at same point.

Usage

```
align_matrix_block(matrix_block)
```

Arguments

matrix_block	The matrix block as raw input text.

Details

The function serves to help build NEXUS files by neatly aligning raw text blocks of taxa and characters. Or in simple terms it takes input that looks like this:

```
Allosaurus  012100?1011
Abelisaurus  0100???0000
Tyrannosaurus  01012012010
Yi  10101?0????
```

And turns it into something that looks like this:

```
Allosaurus     012100?1011
Abelisaurus    0100???0000
Tyrannosaurus  01012012010
Yi             10101?0????
```

I use this in building the NEXUS files on my site, graemetlloyd.com.

Value

Nothing is returned, instead the aligned block is sent to the clipboard ready for pasting into a text editor.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Build example block from above:
x <- paste(c(
  "Allosaurus  012100?1011",
  "Abelisaurus  0100???0000",
  "Tyrannosaurus  01012012010",
  "Yi  10101?0????"
), collapse = "\n")

# Look at block pre-alignment:
x

# Align block and place on clipboard:
## Not run:
align_matrix_block(x)

## End(Not run)

# To test the response open a text editor and paste the
# contents of the clipboard.
```

assign_taxa_to_bins *Assign taxa to time bins*

Description

Given a set of first and last appearances assigns a set of taxa to a series of time bins.

Usage

```
assign_taxa_to_bins(taxon_ages, time_bins)
```

Arguments

taxon_ages	A matrix of taxon ages, with columns for first ("fad") and last ("lad") appearances and rownames correspodning to taxon names.
time_bins	An object of class timeBins.

Details

The various disparity plotting functions (plot_chronophylomorphospace, plot_morphospace_stack, plot_morphospace, plot_multi_morphospace) are designed to allow assignment of taxa to named groups so that these groups may be assigned different colours when plotting. One way taxa may be grouped is temporally, by assignment to a series of time bins.

There are many ways this may be automated and this function provides a very simple one: if the first and last appearance dates of a taxon overlap with a time bin then it can be assigned to that time bin. (In practice, taxa often have multiple occurrences with "ranges" that really represent uncertainty around their true age.)

Note that it is recommended that time bins be named without special characters beyond letters and underscores.

Value

An object of class taxonGroups.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

See Also

plot_chronophylomorphospace, plot_morphospace_stack, plot_morphospace, plot_multi_morphospace, ordinate_cladistic_matrix

Examples

```
# Build example time bins:
time_bins <- matrix(data = c(443.8, 358.9, 358.9, 298.9, 298.9, 251.9,
  251.9, 201.3, 201.3, 145.0, 145.0, 65.5, 65.5, 23.03), ncol = 2,
  byrow = TRUE, dimnames = list(c("Silurodevonian", "Carboniferous",
  "Permian", "Triassic", "Jurassic", "Cretaceous", "Paleogene"),
  c("fad", "lad")))

# Set class as timeBins:
class(time_bins) <- "timeBins"

# Build example taxon ages:
taxon_ages <- matrix(data = c(385.3, 374.5, 407, 374.5, 251, 228, 385.3,
  251, 251, 251, 391.8, 251, 251, 228, 385.3, 391.8, 391.8, 385.3, 311.7,
  359.2, 359.2, 416, 407, 407, 407, 407, 385.3, 397.5, 385.3, 161.2, 385.3,
  345.3, 318.1, 385.3, 228, 385.3, 385.3, 385.3, 385.3, 385.3, 385.3, 385.3,
  385.3, 385.3, 391.8, 407, 391.8, 374.5, 407, 70.6, 311.7, 407, 145.5, 251,
  65.5, 251, 112, 374.5, 374.5, 374.5, 385.3, 311.7, 249.7, 359.2, 391.8,
  374.5, 385.3, 83.5, 418.7, 251, 385.3, 391.8, 374.5, 345.3, 385.3, 385.3,
  407, 411.2, 397.5, 345.3, 374.5, 407, 216.5, 326.4, 411.2, 411.2, 374.5,
  359.2, 391.8, 359.2, 245, 216.5, 374.5, 245, 245, 245, 385.3, 245, 245,
  199.6, 374.5, 385.3, 385.3, 374.5, 306.5, 345.3, 345.3, 411.2, 397.5,
  397.5, 397.5, 397.5, 374.5, 391.8, 374.5, 145.5, 374.5, 326.4, 311.7,
  374.5, 199.6, 374.5, 374.5, 374.5, 374.5, 374.5, 374.5, 374.5, 374.5,
  374.5, 385.3, 397.5, 385.3, 359.2, 397.5, 65.5, 306.5, 397.5, 99.6, 245,
  23.03, 245, 99.6, 359.2, 359.2, 359.2, 374.5, 306.5, 247.4, 318.1, 385.3,
  359.2, 374.5, 70.6, 416, 250.4, 374.5, 385.3, 359.2, 326.4, 374.5, 374.5,
  397.5, 407, 391.8, 326.4, 359.2, 397.5, 203.6, 318.1, 407, 407),
  ncol = 2, dimnames = list(c("Adololopas_moyasmithae", "Adelargo_schultzei",
  "Amadeodipterus_kencampbelli", "Andreyevichthys_epitomus",
  "Aphelodus_anapes", "Archaeoceratodus_avus", "Archaeonectes_pertusus",
  "Arganodus_atlantis", "Ariguna_formosa", "Asiatoceratodus_sharovi",
  "Barwickia_downunda", "Beltanodus_ambilobensis", "Ceratodus_formosa",
  "Ceratodus_latissimus", "Chirodipterus_australis",
  "Chirodipterus_onawwayensis", "Chirodipterus_rhenanus",
  "Chirodipterus_wildungensis", "Conchopoma_gadiforme", "Ctenodus_romeri",
  "Delatitia_breviceps", "Diabolepis_speratus", "Dipnorhynch_cathlesae",
  "Dipnorhynchus_sussmilchi", "Dipnorhynchus_kiandrensis",
  "Dipnorhynchus_kurikae", "Dipterus_cf_valenciennesi",
  "Dipterus_valenciennesi", "Eoctenodus_microsoma",
  "Ferganoceratodus_jurassicus", "Fleurantia_denticulata",
  "Ganopristodus_splendens", "Gnathorhiza_serrata", "Gogodipterus_paddyensis",
  "Gosfordia_truncata", "Griphognathus_minutidens", "Griphognathus_sculpta",
  "Griphognathus_whitei", "Grossipterus_crassus", "Holodipterus_elderae",
  "Holodipterus_gogoensis", "Robinsondipterus_longi",
  "Asthenorhynchus_meemannae", "Holodipterus_santacrucensis",
  "Howidipterus_donnae", "Ichnomylax_kurnai", "Iowadipterus_halli",
  "Jarvikia_arctica", "Jessenia_concentrica", "Lepidosiren_paradoxa",
  "Megapleuron_zangerli", "Melanognathus_canadensis",
  "Metaceratodus_wollastoni", "Microceratodus_angolensis",
  "Mioceratodus_gregoryi", "Namatozodia_pitikanta", "Neoceratodus_forsteri",
  "Nielsenia_nordica", "Oervigia_nordica", "Orlovichthys_limnatis",
```

```
  "Palaeodaphus_insignis", "Palaeophichthys_parvulus", "Paraceratodus_germaini",
  "Parasagenodus_sibiricus", "Pentlandia_macroptera",
  "Phaneropleuron_andersoni", "Pillararhynchus_longi", "Protopterus_annectens",
  "Psarolepis_romeri", "Ptychoceratodus_serratus", "Rhinodipterus_secans",
  "Rhinodipterus_ulrichi", "Rhynchodipterus_elginensis", "Sagenodus_inaequalis",
  "Scaumenacia_curta", "Soederberghia_groenlandica",
  "Sorbitorhynchus_deleaskitus", "Speonesydrion_iani", "Stomiahykus_thlaodus",
  "Straitonia_waterstoni", "Sunwapta_grandiceps", "Tarachomylax_oepiki",
  "Tellerodus_sturi", "Tranodis_castrensis", "Uranolophus_wyomingensis",
  "Westollrhynchus_lehmanni"), c("fad", "lad")))

# Assign taxa to time bins:
assign_taxa_to_bins(taxon_ages = taxon_ages, time_bins = time_bins)
```

`bin_changes` *Counts the changes in a series of time bins*

Description

Given a vector of dates for a series of time bins and another for the times when a character change occurred will return the total number of changes in each bin.

Usage

```
bin_changes(change_times, time_bins)
```

Arguments

`change_times`	A vector of ages in millions of years at which character changes are hypothesised to have occurred.
`time_bins`	An object of class `timeBins`.

Details

Calculates the total number of evolutionary changes in a series of time bins. This is intended as an internal function for rate calculations, but could be used for other purposes (e.g., counting any point events in a series of time bins).

Value

A vector giving the number of changes for each time bin. Names indicate the maximum and minimum (bottom and top) values for each time bin.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Create a random dataset of 100 changes (between 100 and 0 Ma):
change_times <- stats::runif(n = 100, min = 0, max = 100)

# Create 10 equal-length time bins:
time_bins <- matrix(data = c(seq(from = 100, to = 10, length.out = 10),
  seq(from = 90, to = 0, length.out = 10)), ncol = 2,
  dimnames = list(LETTERS[1:10], c("fad", "lad")))

# Set class as timeBins:
class(time_bins) <- "timeBins"

# Get N changes for each bin:
bin_changes(change_times = change_times, time_bins = time_bins)
```

---

bin_character_completeness

Phylogenetic character completeness in time-bins

Description

Given a cladistic matrix, time-scaled tree, and set of time bin boundaries will return the proportional character completeness in each bin.

Usage

```
bin_character_completeness(
  cladistic_matrix,
  time_tree,
  time_bins,
  plot = FALSE,
  confidence.interval = 0.95
)
```

Arguments

`cladistic_matrix`
: A cladistic matrix in the form imported by read_nexus_matrix.

`time_tree`
: A time-scaled phylogenetic tree containing all the taxa in `cladistic_matrix`.

`time_bins`
: An object of class `timeBins`.

`plot`
: An optional choice to plot the results (default is `FALSE`).

`confidence.interval`
: The confidence interval to be used as a proportion (0 to 1). Default is 0.95 (i.e., 95%).

Details

Character completeness metrics have been used as an additional metric for comparing fossil record quality across time, space, and taxa. However, these only usually refer to point samples of fossils in bins, and not our ability to infer information along the branches of a phylogenetic tree.

This function returns the proportional phylogenetic character completeness for a set of time bins.

Value

A list summarising the mean, upper and lower confidence interval, and per character proportional character completeness in each time bin.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Create a random tree for the Day et al. 2016 data set:
day_2016tree <- ape::rtree(n = nrow(day_2016$matrix_1$matrix))
day_2016tree$tip.label <- rownames(x = day_2016$matrix_1$matrix)
day_2016tree$root.time <- max(diag(x = ape::vcv(phy = day_2016tree)))

# Build ten equal-length time bins spanning the tree:
time_bins <- matrix(data = c(seq(from = day_2016tree$root.time,
  to = day_2016tree$root.time - max(diag(x = ape::vcv(phy = day_2016tree))),
  length.out = 11)[1:10], seq(from = day_2016tree$root.time,
  to = day_2016tree$root.time - max(diag(x = ape::vcv(phy = day_2016tree))),
  length.out = 11)[2:11]), ncol = 2, dimnames = list(LETTERS[1:10], c("fad", "lad")))

# Set class as timeBins:
class(time_bins) <- "timeBins"

# Get proportional phylogenetic character completeness in ten equal-length
# time bins:
bin_character_completeness(
  cladistic_matrix = day_2016,
  time_tree = day_2016tree,
  time_bins = time_bins
)

# Same, but with a plot:
bin_character_completeness(
  cladistic_matrix = day_2016,
  time_tree = day_2016tree,
  time_bins = time_bins,
  plot = TRUE
)
```

bin_edge_lengths *Edge-lengths present in time-bins*

Description

Given a time-scaled tree and set of time bin boundaries will sum the edge-lengths present in each bin.

Usage

```
bin_edge_lengths(time_tree, time_bins, pruned_tree = NULL)
```

Arguments

`time_tree`	A time-scaled tree in phylo format with a `$root.time` value.
`time_bins`	An object of class `timeBins`.
`pruned_tree`	A time-scaled tree in phylo format with a `$root.time` value that is a subset of `time_tree`.

Details

Calculates the total edge duration of a time-scaled tree present in a series of time bins. This is intended as an internal function for rate calculations, but may be of use to someone.

The option of using a `pruned_tree` allows the user to correctly classify internal and terminal branches in a subtree of the larger tree. So for example, if taxa A and B are sisters then after pruning B the subtree branch leading to A is composed of an internal and a terminal branch on the complete tree.

Value

`binned_edge_lengths`
: A vector giving the summed values in millions of years for each time bin. Names indicate the maximum and minimum values for each time bin.

`binned_terminal_edge_lengths`
: As above, but counting terminal edges only.

`binned_internal_edge_lengths`
: As above, but counting internal edges only.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Create a random 10-taxon tree:
time_tree <- ape::rtree(n = 10)

# Add root age:
time_tree$root.time <- max(diag(ape::vcv(time_tree)))

# Create time bins:
time_bins <- matrix(data = c(seq(from = time_tree$root.time, to = 0,
  length.out = 11)[1:10], seq(from = time_tree$root.time, to = 0,
  length.out = 11)[2:11]), ncol = 2, dimnames = list(LETTERS[1:10],
  c("fad", "lad")))

# Set class:
class(time_bins) <- "timeBins"

# Get edge lengths for each bin:
bin_edge_lengths(time_tree = time_tree, time_bins = time_bins)
```

---

build_cladistic_matrix

Creates a morphological data file from a matrix

Description

Creates a morphological data file from a character-taxon matrix.

Usage

```
build_cladistic_matrix(
  character_taxon_matrix,
  header = "",
  character_weights = NULL,
  ordering = NULL,
  symbols = NULL,
  equalise.weights = FALSE,
  ignore_duplicate_taxa = FALSE
)
```

Arguments

`character_taxon_matrix`
: A Character-Taxon (columns-rows) matrix, with taxon names as rownames.

`header`
: A scalar indicating any header text (defaults to an empty string: "").

`character_weights`
: A vector specifying the weights used (if not specified defaults to 1).

`ordering`
: A vector indicating whether characters are ordered (`"ordered"`) or unordered (`"unordered"`) (if no specified defaults to ordered).

`symbols` The symbols to use if writing to a file (defaults to the numbers 0:9 then the letters A to V).

`equalise.weights` Optional that overrides the weights specified above make all characters truly equally weighted.

`ignore_duplicate_taxa` Logical indicating whether or not to ignore (allow; TRUE) duplicate taxa or not (FALSE; default).

Details

Claddis generally assumes that matrices will be imported into R from the #NEXUS format, but in some cases (e.g., when using simulated data) it might be desirable to build a matrix within R. This function allows the user to convert such a matrix into the format required by other Claddis functions as long as it only contains a single block.

NB: Currently the function cannot deal directly with costmatrices or continuous characters.

Value

`topper` Contains any header text or costmatrices and pertains to the entire file.

`matrix_N` One or more matrix blocks (numbered 1 to N) with associated information pertaining only to that matrix block. This includes the block name (if specificed, NA if not), the block datatype (one of "CONTINUOUS", "DNA", "NUCLEOTIDE", "PROTEIN", "RESTRICTION", "RNA", or "STANDARD"), the actual matrix (taxa as rows, names stored as rownames and characters as columns), the ordering type of each character (`"ordered"`, `"unordered"` etc.), the character weights, the minimum and maximum values (used by Claddis' distance functions), and the original characters (symbols, missing, and gap values) used for writing out the data.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

See Also

compactify_cladistic_matrix, prune_cladistic_matrix, read_nexus_matrix, safe_taxonomic_reduction, write_nexus_matrix, write_tnt_matrix

Examples

```
# Create random 10-by-50 matrix:
character_taxon_matrix <- matrix(sample(c("0", "1", "0&1", NA, ""),
  500,
  replace = TRUE
),
nrow = 10, dimnames =
  list(apply(matrix(sample(LETTERS, 40,
    replace = TRUE
```

```
  ), nrow = 10), 1, paste,
  collapse = ""
  ), c())
)

# Reformat for use elsewhere in Claddis:
build_cladistic_matrix(character_taxon_matrix)
```

`calculate_g` *Calculate the maximum tree length, g, under parsimony*

Description

Given a costmatrix and set of tip states returns the longest possible tree length under maximum parsimony.

Usage

```
calculate_g(
  costmatrix,
  tip_states,
  polymorphism_behaviour = "polymorphism",
  uncertainty_behaviour = "uncertainty"
)
```

Arguments

`costmatrix` An object of class `costMatrix`.

`tip_states` A character vector of tip states, with polymorphic states separated by `&`, uncertainties by `/`, missing values as `NA`, and inapplicables as empty strings `""`.

`polymorphism_behaviour` One of either "missing", "uncertainty", "polymorphism", or "random". See details.

`uncertainty_behaviour` One of either "missing", "uncertainty", "polymorphism", or "random". See details.

Details

The maximum cost a character could have on any tree under maximum parsimony, termed g, depends on both the individual state-to-state transition costs (captured by a costmatrix) and the sampled states (i.e., the `tip_states` input). In practice this is the maximum parsimony length on the star tree. This length cannot be exceeded by any other tree (Hoyal Cuthill and Lloyd, in review). Note: this is standard practice in phylogenetics software and is also how both PAUP* (Swofford 2003) and TNT (Goloboff et al. 2008; Goloboff and Catalano 2016) calculate maximum cost.

Special cases

A number of special cases apply to calculating g and are discussed further below.

Polymorphisms

Polymorphisms remain a complex problem in phylogenetics and here multiple options are provided to deal with them. These include: 1. `"missing"` - where they are simply replaced by a missing value (see below), 2. `"uncertainty"` - where they are treated as uncertainties instead (see below), 3. `"polymorphism"` - where they are treated as genuinely polymorphic, and 4. `"random"` - where one of the tip states is selected at random.

Options 1, 2, and 4 can be seen as undercounting the true amount of evolution that has occurred. However, how to *correctly* count this amount is unclear. If option 3 is chosen then polymorphic states must be present in `costmatrix` and users should refer to the add_polymorphisms_to_costmatrix function for details on available options.

Uncertainties

Uncertainties are much simpler to deal with than polymorphisms and a means to incorporate them into length counts was laid out in Swofford and Maddison (1992). Indeed, popular software such as PAUP* (Swofford 2003) and TNT (Goloboff et al. 2008; Goloboff and Catalano 2016) simply treat polymorphisms as uncertainties perhaps because of this. There is still a concern of undercounting evolutionary change for uncertainties in the maximum parsimony context as the cheapest possible state will in effect be used everytime, whereas future study that removes uncertainty may reveal s higher cost state to be the true value. As such the same options are offered for uncertainties as polymorphisms, including to treat them as polymorphisms although this should probably only be done where they were miscoded in the first place.

Again, if using uncertainties as uncertainties, these must be included in the costmatrix and this can be done by using the add_uncertainties_to_costmatrix function.

Missing values

In practice missing values (`NA`) may exist amongst `tip_states`. These are permitted, and in practice are mathematically and practically equivalent to a statement that a tip could be any state present in the costmatrix (i.e., a special case of an uncertainty where no state can be ruled out). However, it should be considered in interpretation that *g* will typically become smaller as the number of missing values increases.

Inapplicable values

Inapplicable values (`""`) may also exist amongst `tip_states`. These are conceptually different to missing values as there is no possibility that they can ever be (re)coded. Currently these are treated exactly the same as missing values, but again the user should apply caution in interpreting *g* in such cases as again it will be smaller than otherwise identical characters with fewer inapplicable values.

Value

A single value indicating the maximum length, *g*. Note: this is not modified by `costmatrix$weight`.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com> and Jen Hoyal Cuthill <j.hoyal-cuthill@essex.ac.uk>

References

Goloboff, P. A. and Catalano, S. A., 2016. TNT version 1.5, including a full implementation of phylogenetic morphometrics/ *Cladistics*, **32**. 221-238

Goloboff, P., Farris, J. and Nixon, K., 2008. TNT, a free program for phylogenetic analysis. *Cladistics*, **24**, 774-786.

Hoyal Cuthill, J. F. and Lloyd, G. T., in press. Measuring homoplasy I: comprehensive measures of maximum and minimum cost under parsimony across discrete cost matrix character types. *Cladistics*, bold, .

Swofford, D. L., 2003. *PAUP*. Phylogenetic Analysis Using Parsimony (*and Other Methods). Version 4*. Sinauer Associates, Sunderland, Massachusetts.

Swofford, D. L. and Maddison, W. P., 1992. Parsimony, character-state reconstructions, and evolutionary inferences. In R. L. Mayden (ed.), *Systematics, Historical Ecology, and North American Freshwater Fishes*. Stanford University Press, Stanford. pp187-223.

See Also

calculate_gmax

Examples

```
# Create a Type I character costmatrix:
constant_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 0,
  character_type = "unordered"
)

# Calculate g for the case of five state 0s:
calculate_g(
  costmatrix = constant_costmatrix,
  tip_states = c("0", "0", "0", "0", "0")
)

# Create a Type II character costmatrix:
binary_symmetric_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 1,
  character_type = "unordered"
)

# Calculate g for the case of two state 0s and three state 1s:
calculate_g(
  costmatrix = binary_symmetric_costmatrix,
  tip_states = c("0", "0", "1", "1", "1")
)

# Create a Type III character costmatrix:
unordered_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state= 2,
  character_type = "unordered"
)

# Calculate g for the case of two state 0s and three state 1s and two state 2s:
```

```
calculate_g(
  costmatrix = unordered_costmatrix,
  tip_states = c("0", "0", "1", "1", "1", "2", "2")
)

# Create a Type IV character costmatrix:
linear_ordered_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state= 2,
  character_type = "ordered"
)

# Calculate g for the case of two state 0s and three state 1s and two state 2s:
calculate_g(
  costmatrix = linear_ordered_costmatrix,
  tip_states = c("0", "0", "1", "1", "1", "2", "2")
)

# Create a Type V character costmatrix:
nonlinear_ordered_costmatrix <- convert_adjacency_matrix_to_costmatrix(
  adjacency_matrix = matrix(
    data = c(
      0, 1, 0, 0,
      1, 0, 1, 1,
      0, 1, 0, 0,
      0, 1, 0, 0
    ),
    nrow = 4,
    dimnames = list(0:3, 0:3)
  )
)

# Calculate g for the case of two state 0s, three state 1s, two state 2s and one state 3:
calculate_g(
  costmatrix = nonlinear_ordered_costmatrix,
  tip_states = c("0", "0", "1", "1", "1", "2", "2", "3")
)

# Create a Type VI character costmatrix:
binary_irreversible_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state= 1,
  character_type = "irreversible"
)

# Calculate g for the case of two state 0s and three state 1s:
calculate_g(
  costmatrix = binary_irreversible_costmatrix,
  tip_states = c("0", "0", "1", "1", "1")
)

# Create a Type VII character costmatrix:
multistate_irreversible_costmatrix <- make_costmatrix(
```

```
  min_state = 0,
  max_state= 2,
  character_type = "irreversible"
)

# Calculate g for the case of two state 0s and three state 1s and two state 2s:
calculate_g(
  costmatrix = multistate_irreversible_costmatrix,
  tip_states = c("0", "0", "1", "1", "1", "2", "2")
)

# Create a Type VIII character costmatrix:
binary_dollo_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state= 1,
  character_type = "dollo"
)

# Calculate g for the case of two state 0s and three state 1s:
calculate_g(
  costmatrix = binary_dollo_costmatrix,
  tip_states = c("0", "0", "1", "1", "1")
)

# Create a Type IX character costmatrix:
multistate_dollo_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state= 2,
  character_type = "dollo"
)

# Calculate g for the case of two state 0s and three state 1s and two state 2s:
calculate_g(
  costmatrix = multistate_dollo_costmatrix,
  tip_states = c("0", "0", "1", "1", "1", "2", "2")
)

# Create a Type X character costmatrix:
multistate_symmetric_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state= 5,
  character_type = "ordered"
)
multistate_symmetric_costmatrix$type <- "custom"
multistate_symmetric_costmatrix$costmatrix <- matrix(
  data = c(
    0, 1, 2, 3, 2, 3,
    1, 0, 3, 2, 1, 2,
    2, 3, 0, 3, 2, 1,
    3, 2, 3, 0, 1, 2,
    2, 1, 2, 1, 0, 1,
    3, 2, 1, 2, 1, 0
  ),
```

```
  nrow = multistate_symmetric_costmatrix$size,
  ncol = multistate_symmetric_costmatrix$size,
  byrow = TRUE,
  dimnames = list(
    multistate_symmetric_costmatrix$single_states,
    multistate_symmetric_costmatrix$single_states
  )
)

# Calculate g for the case of two state 0s, three state 1s, two state 2s,
# one state 3, three state 4s and two state 5s:
calculate_g(
  costmatrix = multistate_symmetric_costmatrix,
  tip_states = c("0", "0", "1", "1", "1", "2", "2", "3", "4", "4", "4", "5", "5")
)

# Create a Type XI character costmatrix:
binary_asymmetric_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state= 1,
  character_type = "ordered"
)
binary_asymmetric_costmatrix$type <- "custom"
binary_asymmetric_costmatrix$costmatrix <- matrix(
  data = c(
    0, 1,
    10, 0
  ),
  nrow = binary_asymmetric_costmatrix$size,
  ncol = binary_asymmetric_costmatrix$size,
  byrow = TRUE,
  dimnames = list(
    binary_asymmetric_costmatrix$single_states,
    binary_asymmetric_costmatrix$single_states
  )
)
binary_asymmetric_costmatrix$symmetry <- "Asymmetric"

# Calculate g for the case of two state 0s and three state 1s:
calculate_g(
  costmatrix = binary_asymmetric_costmatrix,
  tip_states = c("0", "0", "1", "1", "1")
)

# Create a Type XII character costmatrix:
multistate_asymmetric_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state= 2,
  character_type = "ordered"
)
multistate_asymmetric_costmatrix$type <- "custom"
multistate_asymmetric_costmatrix$costmatrix <- matrix(
  data = c(
```

```
    0, 1, 1,
    1, 0, 1,
    10, 10, 0
  ),
  nrow = multistate_asymmetric_costmatrix$size,
  ncol = multistate_asymmetric_costmatrix$size,
  byrow = TRUE,
  dimnames = list(
    multistate_asymmetric_costmatrix$single_states,
    multistate_asymmetric_costmatrix$single_states
  )
)
multistate_asymmetric_costmatrix$symmetry <- "Asymmetric"

# Calculate g for the case of two state 0s and three state 1s and two state 2s:
calculate_g(
  costmatrix = multistate_asymmetric_costmatrix,
  tip_states = c("0", "0", "1", "1", "1", "2", "2")
)
```

`calculate_gmax` *Calculate the maximum possible tree length, gmax, under parsimony*

Description

Given a costmatrix and number of tips, calculates the longest possible tree length under maximum parsimony.

Usage

```
calculate_gmax(costmatrix, n_taxa, allow_zeroes = FALSE)
```

Arguments

`costmatrix`	An object of class `costMatrix`.
`n_taxa`	The number of taxa (ree tips) to consider.
`allow_zeroes`	Logical indicating whether some states are allowed to be assigned zero tips. Defaults to `FALSE` (recommended). Note that if the number of states exceeds the number of tips then `allow_zeroes = TRUE` is forced.

Details

The concept of *gmax* was introduced by Hoyal Cuthill et al. (2010) and in the strictest sense represents the integer state frequency that maximizes *g* with the restriction that every state is sampled at least once. This function is intended to relax that slightly using the `allow_zeroes` option and indeed this is not possible in rare cases such as either high levels of missing data or gap-weighted (Thiele 1993) characters where it is possible for the number of states to exceed the (effective) number of tips. For a more detailed description of the problem and implemented solution(s) see Hoyal Cuthill and Lloyd (in press).

Value

A numeric indicating the maximum possible cost, *gmax*.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com> and Jen Hoyal Cuthill <j.hoyal-cuthill@essex.ac.uk>

References

Hoyal Cuthill, J. F. and Lloyd, G. T., in press. Measuring homoplasy I: comprehensive measures of maximum and minimum cost under parsimony across discrete cost matrix character types. *Cladistics*, bold, .

Hoyal Cuthill, J. F., Braddy, S. J. and Donoghue, P. C. J., 2010. A formula for maximum possible steps in multistate characters: isolating matrix parameter effects on measures of evolutionary convergence. *Cladistics*, **26**, 98-102.

Thiele, K.. 1993. The Holy Grail of the perfect character: the cladistic treatment of morphometric data. *Cladistics*, **9**, 275-304.

See Also

calculate_g

Examples

```
# Create a Type I character costmatrix:
constant_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 0,
  character_type = "unordered"
)

# Calculate gmax for ten tips:
calculate_gmax(
  costmatrix = constant_costmatrix,
  n_taxa = 10
)

# Create a Type II character costmatrix:
binary_symmetric_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 1,
  character_type = "unordered"
)

# Calculate gmax for ten tips:
calculate_gmax(
  costmatrix = binary_symmetric_costmatrix,
  n_taxa = 10
)
```

```
# Create a Type III character costmatrix:
unordered_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 2,
  character_type = "unordered"
)

# Calculate gmax for ten tips:
calculate_gmax(
  costmatrix = unordered_costmatrix,
  n_taxa = 10
)

# Create a Type IV character costmatrix:
linear_ordered_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 2,
  character_type = "ordered"
)

# Calculate gmax for ten tips:
calculate_gmax(
  costmatrix = linear_ordered_costmatrix,
  n_taxa = 10
)

# Create a Type V character costmatrix:
nonlinear_ordered_costmatrix <- convert_adjacency_matrix_to_costmatrix(
  adjacency_matrix = matrix(
    data = c(
      0, 1, 0, 0,
      1, 0, 1, 1,
      0, 1, 0, 0,
      0, 1, 0, 0
    ),
    nrow = 4,
    dimnames = list(0:3, 0:3)
  )
)

# Calculate gmax for ten tips:
calculate_gmax(
  costmatrix = nonlinear_ordered_costmatrix,
  n_taxa = 10
)


# Create a Type VI character costmatrix:
binary_irreversible_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 1,
  character_type = "irreversible"
)
```

```
# Calculate gmax for ten tips:
calculate_gmax(
  costmatrix = binary_irreversible_costmatrix,
  n_taxa = 10
)

# Create a Type VII character costmatrix:
multistate_irreversible_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 2,
  character_type = "irreversible"
)

# Calculate gmax for ten tips:
calculate_gmax(
  costmatrix = multistate_irreversible_costmatrix,
  n_taxa = 10
)

#' # Create a Type VIII character costmatrix:
binary_dollo_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 1,
  character_type = "dollo"
)

# Calculate gmax for ten tips:
calculate_gmax(
  costmatrix = binary_dollo_costmatrix,
  n_taxa = 10
)

# Create a Type IX character costmatrix:
multistate_dollo_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 2,
  character_type = "dollo"
)

# Calculate gmax for ten tips:
calculate_gmax(
  costmatrix = multistate_dollo_costmatrix,
  n_taxa = 10
)

# Create a Type X character costmatrix:
multistate_symmetric_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 5,
  character_type = "ordered"
)
multistate_symmetric_costmatrix$type <- "custom"
```

```
multistate_symmetric_costmatrix$costmatrix <- matrix(
  data = c(
    0, 1, 2, 3, 2, 3,
    1, 0, 3, 2, 1, 2,
    2, 3, 0, 3, 2, 1,
    3, 2, 3, 0, 1, 2,
    2, 1, 2, 1, 0, 1,
    3, 2, 1, 2, 1, 0
  ),
  nrow = multistate_symmetric_costmatrix$size,
  ncol = multistate_symmetric_costmatrix$size,
  byrow = TRUE,
  dimnames = list(
    multistate_symmetric_costmatrix$single_states,
    multistate_symmetric_costmatrix$single_states
  )
)

# Calculate gmax for ten tips:
calculate_gmax(
  costmatrix = multistate_symmetric_costmatrix,
  n_taxa = 10
)

# Create a Type XI character costmatrix:
binary_asymmetric_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 1,
  character_type = "ordered"
)
binary_asymmetric_costmatrix$type <- "custom"
binary_asymmetric_costmatrix$costmatrix <- matrix(
  data = c(
    0, 1,
    10, 0
  ),
  nrow = binary_asymmetric_costmatrix$size,
  ncol = binary_asymmetric_costmatrix$size,
  byrow = TRUE,
  dimnames = list(
    binary_asymmetric_costmatrix$single_states,
    binary_asymmetric_costmatrix$single_states
  )
)
binary_asymmetric_costmatrix$symmetry <- "Asymmetric"

# Calculate gmax for ten tips:
calculate_gmax(
  costmatrix = binary_asymmetric_costmatrix,
  n_taxa = 10
)

# Create a Type XII character costmatrix:
```

```
multistate_asymmetric_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 2,
  character_type = "ordered"
)
multistate_asymmetric_costmatrix$type <- "custom"
multistate_asymmetric_costmatrix$costmatrix <- matrix(
  data = c(
    0, 1, 1,
    1, 0, 1,
    10, 10, 0
  ),
  nrow = multistate_asymmetric_costmatrix$size,
  ncol = multistate_asymmetric_costmatrix$size,
  byrow = TRUE,
  dimnames = list(
    multistate_asymmetric_costmatrix$single_states,
    multistate_asymmetric_costmatrix$single_states
  )
)
multistate_asymmetric_costmatrix$symmetry <- "Asymmetric"

# Calculate gmax for ten tips:
calculate_gmax(
  costmatrix = multistate_asymmetric_costmatrix,
  n_taxa = 10
)
```

`calculate_kardashian_index`

Calculates a researcher's Kardashian Index

Description

Given counts of a researcher's Twitter followers and citations, returns their Kardashian Index.

Usage

```
calculate_kardashian_index(twitter_followers, total_citations)
```

Arguments

`twitter_followers`
: The number of twitter followers the researcher has.

`total_citations`
: The total number of citations across the researcher's publications (e.g., as garnered from a Google Scholar profile).

Details

This function implements the Kardashian Index of Hall (2014) and interested readers should consult that paper for more background.

Value

A scalar representing the ratio of expected Twitter followers (based on number of citations) to actual Twitter followers. Values greater than one indicate more Twitter followers than expected, those below one, fewer. According to Hall (2014), values above 5 are "Science Kardashians".

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Hall, N., 2014. The Kardashian index: a measure of discrepant social media profile for scientists. *Genome Biology*, **15**, 424.

Examples

```
# Calculate the Kardashian Index of Sam Giles (@GilesPalaeoLab)
# as of 10/5/21:
calculate_kardashian_index(
  twitter_followers = 6534,
  total_citations = 550
)

# Calculate the Kardashian Index of Christopher Jackson (@seis_matters)
# as of 10/5/21:
calculate_kardashian_index(
  twitter_followers = 26000,
  total_citations = 6265
)

# Calculate the Kardashian Index of Graeme T. Lloyd (@GraemeTLloyd)
# as of 10/5/21:
calculate_kardashian_index(
  twitter_followers = 2133,
  total_citations = 2780
)

# Calculate the Kardashian Index of Katie Mack (@AstroKatie)
# as of 10/5/21:
calculate_kardashian_index(
  twitter_followers = 394900,
  total_citations = 1131
)
```

calculate_morphological_distances

Get distance matrices from a cladistic matrix

Description

Takes a cladistic morphological dataset and converts it into a set of pairwise distances.

Usage

```
calculate_morphological_distances(
  cladistic_matrix,
  distance_metric = "mord",
  ged_type = "wills",
  distance_transformation = "arcsine_sqrt",
  polymorphism_behaviour = "min_difference",
  uncertainty_behaviour = "min_difference",
  inapplicable_behaviour = "missing",
  character_dependencies = NULL,
  alpha = 0.5
)
```

Arguments

`cladistic_matrix`
: A character-taxon matrix in the format imported by read_nexus_matrix.

`distance_metric`
: The distance metric to use. Must be one of `"gc"`, `"ged"`, `"red"`, or `"mord"` (the default).

`ged_type`
: The type of GED to use. Must be one of `"legacy"`, `"hybrid"`, or `"wills"` (the default). See details for an explanation.

`distance_transformation`
: The type of distance transformation to perform. Options are `"none"`, `"sqrt"`, or `"arcsine_sqrt"` (the default). (Note: this is only really appropriate for the proportional distances, i.e., "gc" and "mord".)

`polymorphism_behaviour`
: The distance behaviour for dealing with polymorphisms. Must be one of `"mean_difference"`, `"min_difference"` (the default), or `"random"`.

`uncertainty_behaviour`
: The distance behaviour for dealing with uncertainties. Must be one of `"mean_difference"`, `"min_difference"` (the default), or `"random"`.

`inapplicable_behaviour`
: The behaviour for dealing with inapplicables. Must be one of `"missing"` (default), or `"hsj"` (Hopkins and St John 2018; see details).

`character_dependencies`
: Only relevant if using `inapplicable_behaviour = "hsj"`. Must be a two-column matrix with colnames "dependent_character" and "independent_character" that specifies character hierarchies. See details.

`alpha`
: The alpha value (sensu Hopkins and St John 2018). Only relevant if using `inapplicable_behaviour = "hsj"`. See details.

Details

There are many options to consider when generating a distance matrix from morphological data, including the metric to use, how to treat inapplicable, polymorphic (e.g., 0&1), or uncertain (e.g., 0/1) states, and whether the output should be transformed (e.g., by taking the square root so that the distances are - or approximate - Euclidean distances). Some of these issues have been discussed previously in the literature (e.g., Lloyd 2016; Hopkins and St John 2018), but all likely require further study.

Claddis currently offers four different distance metrics: 1. Raw Euclidean Distance (`"red"`) - this is only really applicable if there are no missing data, 2. The Gower Coefficient (`"gc"`; Gower 1971) - this rescales distances by the number of characters that can be coded for both taxa in each pairwise comparison thus correcting for missing data, 3. The Maximum Observable Rescaled Distance (`"mord"`) - this was introduced by Lloyd (2016) as an extension of the `"gc"` designed to deal with the fact that multistate ordered characters can lead to `"gc"`s of greater than 1 and works by rescaling by the maximum possible distance that could be observed based on the number of characters codable in each pairwise comparison meaning all resulting distances are on a zero to one scale, and 4. The Generalised Euclidean Distance - this was introduced by Wills (1998) as a means of correcting for the fact that a `"red"` metric will become increasingly non-Euclidean as the amount of missing data increases and works by filling in missing distances (for characters that are coded as missing in at least one taxon in the pairwise comparison) by using the mean pairwise dissimilarity for that taxon pair as a substitute. In effect then, `"red"` makes no consideration of missing data, `"gc"` and `"mord"` normalise by the available data (and are identical if there are no ordered multistate characters), and `"ged"` fills in missing distances by extrapolating from the available data.

Note that Lloyd (2016) misidentified the substitute dissimilarity for the `"ged"` as the mean for the whole data set (Hopkins and St John 2018) and this was the way the GED implementation of Claddis operated up to version 0.2. This has now been amended (as of version 0.3) so that the function produces the `"ged"` in the form that Wills (1998) intended. However, this implementation can still be accessed as the `"legacy"` option for `ged_type`, with `"wills"` being the WIlls (1998) implementation. An advantage of this misinterpreted form of the GED is that it will always return a complete pairwise distance matrix, however it is not recommended (see Lloyd 2016). Instead a third option for `ged_type` - (`"hybrid"`) - offers the same outcome but only uses the mean distance from the entire matrix in the case where there are no codable characters in common in a pairwise comparison. This new hybrid option has not been used in a published study.

Typically the resulting distance matrix will be used in an ordination procedure such as principal coordinates (effectively classical multidimensional scaling where k, the number of axes, is maximised at N - 1, where N is the number of rows (i.e., taxa) in the matrix). As such the distance should be - or approximate - Euclidean and hence a square root transformation is typically applied (`distance_transformation` with the `"sqrt"` option). However, if applying pre-ordination (i.e., ordination-free) disparity metrics (e.g., weighted mean pairwise distance) you may wish to avoid any transformation (`"none"` option). In particular the MORD will only fall on a zero to one scale if this is the case. However, if transforming the MORD for ordination this zero to one property

may mean the arcsine square root (`"arcsine_sqrt"` option) is preferred. (Note that if using only unordered multistate or binary characters and the `"gc"` the zero to one scale will apply too.)

An unexplored option in distance matrix construction is how to deal with polymorphisms (Lloyd 2016). Up to version 0.2 of Claddis all polymorphisms were treated the same regardless of whether they were true polymorphisms (multiple states are observed in the taxon) or uncertainties (multiple, but not all states, are posited for the taxon). Since version 0.3, however, these two forms can be distinguished by using the different #NEXUS forms (Maddison et al. 1997), i.e., (01) for polymorphisms and {01} for uncertainties and within Claddis these are represented as 0&1 or 0/1, respectively. Thus, since 0.3 Claddis allows these two forms to be treated separately, and hence differently (with `polymorphism_behaviour` and `uncertainty_behaviour`). Again, up to version 0.2 of Claddis no options for polymorphism behaviour were offered, instead only a minimum distance was employed. I.e., the distance between a taxon coded 0&1 and a taxon coded 2 would be the smaller of the comparisons 0 with 2 or 1 with 2. Since version 0.3 this is encoded in the `"min_difference"` option. Currently two alternatives (`"mean_difference"` and `"random"`) are offered. The first takes the mean of each possible difference and the second simply samples one of the states at random. Note this latter option makes the function stochastic and so it should be rerun multiple times (for example, with a `for` loop or `apply` function). In general this issue (and these options) are not explored in the literature and so no recommendation can be made beyond that users should think carefully about what this choice may mean for their individual data set(s) and question(s).

A final consideration is how to deal with inapplicable characters. Up to version 0.2 Claddis treated inapplicable and missing characters the same (as NA values, i.e., missing data). However, since Claddis version 0.3 these can be imported separately, i.e., by using the "MISSING" and "GAP" states in #NEXUS format (Maddison et al. 1997), with the latter typically representing the inapplicable character. These appear as NA and empty strings (""), respectively, in Claddis format. Hopkins and St John (2018) showed how inapplicable characters - typically assumed to represent secondary characters - could be treated in generating distance matrices. These are usually hierarchical in form. E.g., a primary character might record the presence or absence of feathers and a secondary character whether those feathers are symmetric or asymmetric. The latter will generate inapplicable states for taxa without feathers and without correcting for this ranked distances can be incorrect (Hopkins and St John 2018). Unfortunately, however, the #NEXUS format (Maddison et al. 1997) does not really allow explicit linkage between primary and secondary characters and so this information must be provided separately to use the Hopkins and St John (2018) approach. This is done here with the `character_dependencies` option. This must be in the form of a two-column matrix with column headers of `"dependent_character"` and `"independent_character"`. The former being secondary characters and the latter the corresponding primary character. (Note that characters are to be numbered across the whole matrix from 1 to N and do not restart with each block of the matrix.) If using `inapplicable_behaviour = "hsj"` the user must also provide an `alpha` value between zero and one. When `alpha = 0` the secondary characters contribute nothing to the distance and when `alpha = 1` the primary character is not counted in the weight separately (see Hopkins and St John 2018). The default value (0.5) offers a compromise between these two extremes.

Here the implementation of this approach differs somewhat from the code available in the supplementary materials to Hopkins and St John (2018). Specifically, this approach is incorporated (and used) regardless of the overriding distance metric (i.e., the `distance_metric` option). Additionally, the Hopkins and St John function specifically allows an extra level of dependency (secondary and tertary characters) with these being applied recursively (tertiary first then secondary). Here, though, additional levels of dependency do not need to be defined by the user as this information is

already encoded in the `character_dependencies` option. Furthermore, because of this any level of dependency is possible (if unlikely), e.g., quarternary etc.

Value

`distance_metric`
: The distance metric used.

`distance_matrix`
: The pairwise distance matrix generated.

`comparable_character_matrix`
: The matrix of characters that can be compared for each pairwise distance.

`comparable_weights_matrix`
: The matrix of weights for each pairwise distance.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com> and Thomas Guillerme <guillert@tcd.ie>

References

Gower, J. C., 1971. A general coefficient of similarity and some of its properties. *Biometrika*, **27**, 857-871.

Hopkins, M. J. and St John, K., 2018. A new family of dissimilarity metrics for discrete character matrices that include inapplicable characters and its importance for disparity studies. *Proceedings of the Royal Society of London B*, **285**, 20181784.

Lloyd, G. T., 2016. Estimating morphological diversity and tempo with discrete character-taxon matrices: implementation, challenges, progress, and future directions. *Biological Journal of the Linnean Society*, **118**, 131-151.

Maddison, D. R., Swofford, D. L. and Maddison, W. P., 1997. NEXUS: an extensible file format for systematic information. *Systematic Biology*, **46**, 590-621.

Wills, M. A., 1998. Crustacean disparity through the Phanerozoic: comparing morphological and stratigraphic data. *Biological Journal of the Linnean Society*, **65**, 455-500.

Examples

```
# Get morphological distances for the Day et al. (2016) data set:
distances <- calculate_morphological_distances(cladistic_matrix = day_2016)

# Show distance metric:
distances$distance_metric

# Show distance matrix:
distances$distance_matrix

# Show number of characters that can be scored for
# each pairwise comparison:
distances$comparable_character_matrix

# To repeat using the Hopkins and St John approach
```

```
# we first need to define the character dependency
# (here there is only one, character 8 is a
# secondary where 7 is the primary character):
character_dependencies <- matrix(c(8, 7),
  ncol = 2,
  byrow = TRUE, dimnames = list(
    c(),
    c(
      "dependent_character",
      "independent_character"
    )
  )
)

# Get morphological distances for the Day et
# al. (2016) data set using HSJ approach:
distances <- calculate_morphological_distances(
  cladistic_matrix = day_2016,
  inapplicable_behaviour = "hsj",
  character_dependencies = character_dependencies,
  alpha = 0.5
)

# Show distance metric:
distances$distance_metric

# Show distance matrix:
distances$distance_matrix

# Show number of characters that can be scored for
# each pairwise comparison:
distances$comparable_character_matrix

# Show weighting of each calculable pairwise distance:
distances$comparable_weights_matrix
```

`calculate_MPD` *Calculate mean pairwise distances*

Description

Given distanceMatrices and taxonGroups objects calculates their mean pairwise distances.

Usage

```
calculate_MPD(distances, taxon_groups)
```

Arguments

distances	An object of class distanceMatrices.
taxon_groups	An object of class taxonGroups.

Details

Not all measures of disparity (morphological distance) require an ordination space. For example, the pariwise distances between taxa are themselves a disparity metric. This function takes the output from calculate_morphological_distances and a set of taxon groups and returns the mean pairwise distance for each of those groups.

Value

A labelled vector of weighted mean pairwise distances.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Get morphological distances for the Day et al. (2016) data set:
distances <- calculate_morphological_distances(
  cladistic_matrix = day_2016,
  distance_metric = "mord",
  distance_transformation = "none"
)

# Build simple taxonomic groups for Day et al. (2016) data set:
taxon_groups <- list(nonBurnetiamorpha = c("Biarmosuchus_tener", "Hipposaurus_boonstrai",
 "Bullacephalus_jacksoni", "Pachydectes_elsi", "Niuksenitia_sukhonensis", "Ictidorhinus_martinsi",
 "RC_20", "Herpetoskylax_hopsoni"), Burnetiamorpha = c("Lemurosaurus_pricei", "Lobalopex_mordax",
  "Lophorhinus_willodenensis", "Proburnetia_viatkensis", "Lende_chiweta",
  "Paraburnetia_sneeubergensis", "Burnetia_mirabilis", "BP_1_7098"))

# Set class as taxonGroups:
class(taxon_groups) <- "taxonGroups"

# Calculate mean pairwise distances:
calculate_MPD(distances, taxon_groups)
```

---

calculate_tree_length *Calculates the parsimony length of a set of phylogenetic tree(s)*

Description

Given a tree, or set of trees, and a cladistic matrix returns their parsimony length in number of steps.

Usage

```
calculate_tree_length(
  trees,
  cladistic_matrix,
  inapplicables_as_missing = FALSE,
  polymorphism_behaviour,
  uncertainty_behaviour,
  polymorphism_geometry,
  polymorphism_distance,
  state_ages,
  dollo_penalty
)
```

Arguments

`trees` A tree (`phylo` object) or set of trees (`multiPhylo` object).

`cladistic_matrix` A character-taxon matrix in the format imported by read_nexus_matrix. These should be discrete with rownames (ytaxon labels) matching the tip labels of `trees`.

`inapplicables_as_missing` Logical that decides whether or not to treat inapplicables as missing (TRUE) or not (FALSE, the default and recommended option).

`polymorphism_behaviour` One of either "missing", "uncertainty", "polymorphism", or "random". See details.

`uncertainty_behaviour` One of either "missing", "uncertainty", "polymorphism", or "random". See details.

`polymorphism_geometry` Argument passed to make_costmatrix.

`polymorphism_distance` Argument passed to make_costmatrix.

`state_ages` Argument passed to make_costmatrix.

`dollo_penalty` Argument passed to make_costmatrix.

Details

Under the maximum parsimony criterion, a phylogenetic hypothesis is considered optimal if it requires the fewest number of evolutionary "steps" or - to generalise to non-discrete values - minimum total cost. In order to evalulate this criterion we must therefore be able to calculate a tree's "length" (total cost assuming the lowest cost for every character used). Given a set of phylogenetic hypothes(es) and a cladistic matrix this function calculates the minimum length for each tree.

Input data format

This function operates on a phylogenetic tree, or trees (in `ape` format), and a cladistic matrix (in `cladisticMatrix` format). However, the algorithm used is based on the generalised costmatrix

approach of Swofford and Maddison (1992) and hence costmatrices need to be defined for each character (this is done internally by calling make_costmatrix), and some of the options are merely passed to this function.

Algorithm

Technically the Swofford and Maddison (1992) algorithm is designed for ancestral state reconstruction, but as its' first pass of the tree assigns lengths for each possible state at each node the minimum value of these options at the root is also the tree length for that character and hence by skipping the later steps this can be used as a tree length algorithm by simply summing the values across each character. The choice of the Swofford and Maddison algorithm, rather than the Wagner or Fitch algorithms (for ordered and unordered characters, respectively) is to generalize to the broadest range of character types, including asymmetric characters (Camin-Sokal, Dollo, stratigraphic), custom character types (specified using costmatrices or character state trees), as well as to any resolution of tree (i.e., including multifurcating trees - important for establishing maximum costs for homoplasy indices). The only restriction here is that the tree must be rooted such that time's arrow is explicitly present. This is essential, as the root defines the lengths across the whole tree, but also for asymmetric characters directionality must be explicit, as well as some downstream approaches (such as ACCTRAN and DELTRAN). The two obvious drawbacks to this algorithm are that it can be slower and that it is not appropriate for unrooted trees.

Costmatrices and costmatrix options

Costmatrices are described in detail in the make_costmatrix manual, as are the options that are passed from this function to that one. Thus, the user is directed there for a more in-depth discussion of options.

Inapplicable and missing characters

In practice these two character types are treated the same for length calculations - in effect these are "free" characters that do not constrain the tree length calculation in the same way that a coded character would (because a coded character's transition cost must be accounted for; Swofford and Maddison 1992). Note that there *are* reasons to take differences into account in phylogenetic inference itself (see papers by Brazeau et al. 2019 and Goloboff et al. in press). The option to treat them differently here is therefore only important in terms of downstream analyses, such as ancestral state reconstruction (see reconstruct_ancestral_states for details).

Polymorphisms and uncertainties

Polymorphisms (coded with empersands between states) and uncertainties (coded with slashes between states) can be interpreted in different ways, including those that affect estimates of tree length. Hence four options are provided to the user here:

1. Missing (`polymorphism_behaviour = "missing"` or `uncertainty_behaviour = "missing"`). Here polymorphisms are simply replaced by the missing character (`NA`). This removes polymorphisms and uncertainties from the calculation process completely (likely leading to undercounts), and hence is not generally recommended.
2. Uncertainty (`polymorphism_behaviour = "uncertainty"` or `uncertainty_behaviour = "uncertainty"`). This is the intended use of uncertain codings (e.g., `0/1`) and constrains the tree length calculation to having to explain the *least* costly transition of those in the uncertainty. This is recommended for uncertain characters (although note that it biases the result towards the shortest possible length), but not truly polymorphic characters (as one or more state acquisitions are being missed, see Nixon and Davis 1991 and make_costmatrix for discussion of this). This is

also - to the best of my knowledge - the approach used by most parsimony software, such as PAUP* (Swofford 2003) and TNT (Goloboff et al. 2008; Goloboff and Catalano 2016).

3. Polymorphism (`polymorphism_behaviour = "polymorphism"` or `uncertainty_behaviour = "polymorphism"`). If polymorphisms are real then some means of accounting for the changes that produce them seems appropriate, albeit difficult (see Nixon and Davis 1991 and Swofford and Maddison 1992 for discussions). If this option is applied it triggers the downstream options in *make_costmatrix* (by internally setting `include_polymorphisms = TRUE`), and the user should look there for more information. This is tentatively recommended for true polymorphisms (but note that it complicates interpretation), but not uncertainties.
4. Random (`polymorphism_behaviour = "random"` or `uncertainty_behaviour = "random"`). Another means of dealing with multiple-state characters is simply to sample a single state at random for each one, for example as Watanabe (2016) did with their PERDA algorithm. This simplifies the process, but also logically requires running the function multiple times to quantify uncertainty. This is not recommended for true polymorphisms (as interpretation is confounded), but may be appropriate for a less downwards biased tree count than `"uncertainty"`.

These choices can also effect ancestral state estimation (see reconstruct_ancestral_states).

Polytomies

Polytomies are explicitly allowed by the function, but will always be treated as "hard" (i.e., literal multifurcations). Note that typically these will lead to higher tree lengths than fully bifurcating trees and indeed that the maximum cost is typically calculated from the star tree (single multifurcation).

Further constraints

In future the function will allow restrictions to be placed on the state at particular internal nodes. This can have multiple applications, including (for example) treating some taxa as ancestral such that their states are directly tied to specific nodes, e.g., in stratocladistics (Fisher 1994; Marcot and Fox 2008).

Character weights

Tree lengths output already include corrections for character weights as supplied in the `cladistic_matrix` input. So, for example, if a binary character costs two on the tree, but is weighted five then it will contribute a total cosr of 10 to the result.

Value

A list with multiple components, including:

`input_trees` The tree(s) used as input.

`input_matrix` The raw (unmodified) `cladistic_matrix` input.

`input_options` The various input options used. Output for use by downstream functions, such as ancestral state estimation and stochastic character mapping.

`costmatrices` The costmatrices (one for each character) used. These are typically generated automatically by the funcion, but are output here for later use in ancestral state estimation and stochastic character mapping functions.

`character_matrix` The single character matrix object used. Essentially the `input_matrix` modified by the `input_options`.

`character_lengths`
: A matrix of characters (rows) and trees (columns) with values indicating the costs. The column sums of this matrix are the `tree_lengths` values. This output can also be used for homoplasy metrics.

`character_weights`
: A vector of the character weights used.

`tree_lengths`
: The primary output - the length for each input tree in total cost.

`node_values`
: The values (lengths for each state) for each node acrss trees and characters. This is used by reconstruct_ancestral_states for ancestral state reconstruction.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Brazeau, M. D., Guillerme, T. and Smith, M. R., 2019. An algorithm for morphological phylogenetic analysis with inapplicable data. *Systematic Biology*, **68**, 619-631.

Fisher, D. C., 1994. Stratocladistics: morphological and temporal patterns and their relation to phylogenetic process. In L. Grande and O. Rieppel (eds.), *Interpreting the Hierarchy of Nature*. Academic Press, San Diego. pp133–171.

Goloboff, P. A. and Catalano, S. A., 2016. TNT version 1.5, including a full implementation of phylogenetic morphometrics/ *Cladistics*, **32**. 221-238

Goloboff, P., Farris, J. and Nixon, K., 2008. TNT, a free program for phylogenetic analysis. *Cladistics*, **24**, 774-786.

Goloboff, P. A., De Laet, J., Rios-Tamayo, D. and Szumik, C. A., in press. A reconsideration of inapplicable characters, and an approximation with step-matrix recoding. *Cladistics*.

Marcot, J. D. and Fox, D. L., 2008. StrataPhy: a new computer program for stratocladistic analysis. *Palaeontologia Electronica*, **11**, 5A.

Nixon, K. C. and Davis, J. I., 1991. Polymorphic taxa, missing values and cladistic analysis. *Cladistics*, **7**, 233-241.

Swofford, D. L., 2003. *PAUP*. Phylogenetic Analysis Using Parsimony (*and Other Methods). Version 4*. Sinauer Associates, Sunderland, Massachusetts.

Swofford, D. L. and Maddison, W. P., 1992. Parsimony, character-state reconstructions, and evolutionary inferences. *In* R. L. Mayden (ed.) Systematics, Historical Ecology, and North American Freshwater Fishes. Stanford University Press, Stanford, p187-223.

Watanabe, A., 2016. The impact of poor sampling of polymorphism on cladistic analysis. *Cladistics*, **32**, 317-334.

See Also

make_costmatrix, reconstruct_ancestral_states

Examples

```
# Use Gauthier 1986 as example matrix:
cladistic_matrix <- Claddis::gauthier_1986

# Use one of the MPTs from a TNT analysis as the tree:
tree <- ape::read.tree(
  text = paste(
    "(Outgroup,(Ornithischia,(Sauropodomorpha,(Ceratosauria,Procompsognathus,",
    "Liliensternus,(Carnosauria,(Ornithmimidae,Saurornitholestes,Hulsanpes,(Coelurus,",
    "Elmisauridae,(Compsognathus,(Ornitholestes,Microvenator,Caenagnathidae,",
    "(Deinonychosauria,Avialae)))))))));",
    sep = ""
  )
)

# Calculate tree length (and only use tree lengths from output):
calculate_tree_length(
  trees = tree,
  cladistic_matrix = cladistic_matrix,
  inapplicables_as_missing = TRUE,
  polymorphism_behaviour = "uncertainty",
  uncertainty_behaviour = "uncertainty",
  polymorphism_geometry = "simplex",
  polymorphism_distance = "euclidean",
  state_ages = c(200, 100),
  dollo_penalty = 999
)$tree_lengths
```

`calculate_WMPD` *Calculate weighted mean pairwise distances*

Description

Given distanceMatrices and taxonGroups objects calculates their weighted mean pairwise distances.

Usage

```
calculate_WMPD(distances, taxon_groups)
```

Arguments

`distances`	An object of class `distanceMatrices`.
`taxon_groups`	An object of class `taxonGroups`.

Details

Not all measures of disparity (morphological distance) require an ordination space. For example, the pariwise distances between taxa are themselves a disparity metric. However, due to variable amounts of missing data each pairwise distance should not necessarily be considered equal. Specifically, it could be argued that for a group of taxa the mean distance should be weighted by the number of characters that distance is based on, or more specifically the sum of the weights of those characters (e.g., Close et al. 2015).

This function takes the output from calculate_morphological_distances and a set of taxon groups and returns the weighted mean pairwise distance for those groups.

Value

A labelled vector of weighted mean pairwise distances.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Close, R. A., Friedman, M., Lloyd, G. T. and Benson, R. B. J., 2015. Evidence for a mid-Jurassic adaptive radiation in mammals. *Current Biology*, **25**, 2137-2142.

Examples

```
# Get morphological distances for the Day et al. (2016) data set:
distances <- calculate_morphological_distances(
  cladistic_matrix = day_2016,
  distance_metric = "mord",
  distance_transformation = "none"
)

# Build simple taxonomic groups for Day et al. (2016) data set:
taxon_groups <- list(nonBurnetiamorpha = c("Biarmosuchus_tener", "Hipposaurus_boonstrai",
 "Bullacephalus_jacksoni", "Pachydectes_elsi", "Niuksenitia_sukhonensis", "Ictidorhinus_martinsi",
 "RC_20", "Herpetoskylax_hopsoni"), Burnetiamorpha = c("Lemurosaurus_pricei", "Lobalopex_mordax",
  "Lophorhinus_willodenensis", "Proburnetia_viatkensis", "Lende_chiweta",
  "Paraburnetia_sneeubergensis", "Burnetia_mirabilis", "BP_1_7098"))

# Set class as taxonGroups:
class(taxon_groups) <- "taxonGroups"

# Calculate mean pairiwise distances:
calculate_MPD(distances, taxon_groups)

# Now calculate weighted mean pairwise distances:
calculate_WMPD(distances, taxon_groups)
```

check_cladisticMatrix *Check cladisticMatrix object for errors*

Description

Internal function to check cladisticMatrix object for errors.

Usage

```
check_cladisticMatrix(cladistic_matrix)
```

Arguments

`cladistic_matrix`
: An object of class `cladisticMatrix`.

Details

Internal Claddis function. Nothing to see here. Carry on.

Value

An error message or empty vector if no errors found.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Check that this is a valid cladisticMatrix object (will return error message as class
# is not set):
check_cladisticMatrix(cladistic_matrix = day_2016)
```

check_costMatrix *Check a costMatrix object for errors*

Description

Internal function to check a costMatrix object for errors.

Usage

```
check_costMatrix(costmatrix)
```

Arguments

`costmatrix` A costMatrix object.

Details

Costmatrix objects are more complex than what will typically be shown to the user. This function checks this hidden structure and reports any errors it finds.

These checks include rules 1-7 from Hoyal Cuthill and lloyd (i prep.).

Value

An error message or empty vector if no errors found.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Make an unordered costmatrix:
costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 2,
  character_type = "unordered"
)

# Check that this is a valid costMatrix object (should return empty vector):
check_costMatrix(costmatrix = costmatrix)
```

`check_stateGraph` *Check a stateGraph object for errors*

Description

Internal function to check a stateGraph object for errors.

Usage

```
check_stateGraph(stategraph)
```

Arguments

`stategraph` A stateGraph object.

Details

Stategraph objects are more complex than what will typically be shown to the user. This function checks this hidden structure and reports any errors it finds.

These checks include rules 1-7 from Hoyal Cuthill and Lloyd (i prep.).

Value

An error message or empty vector if no errors found.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Make an unordered costmatrix:
costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 2,
  character_type = "unordered"
)

# Convert costmatrix to stategraph:
stategraph <- convert_costmatrix_to_stategraph(costmatrix = costmatrix)

# Check that this is a valid stateGraph object (should return empty vector):
check_stateGraph(stategraph = stategraph)
```

---

check_taxonGroups *Check taxonGroups object for errors*

Description

Internal function to check taxonGroups object for errors.

Usage

```
check_taxonGroups(taxon_groups)
```

Arguments

taxon_groups	An object of class taxonGroups.

Details

Internal Claddis function. Nothing to see here. Carry on.

Value

An error message or empty vector if no errors found.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Create a taxon groups object:
taxon_groups <- list(
  Group_A = c("Species_1", "Species_2", "Species_3"),
  Group_B = c("Species_3", "Species_4"),
  Group_C = c("Species_5", "Species_6", "Species_7", "Species_8")
)

# Check that this is a valid taxonGroups object (will return error message as class
# is not set):
check_taxonGroups(taxon_groups = taxon_groups)
```

---

check_timeBins *Check timeBins object for errors*

Description

Internal function to check timeBins object for errors.

Usage

```
check_timeBins(time_bins)
```

Arguments

time_bins	A timeBins object.

Details

Internal Claddis function. Nothing to see here. Carry on.

Value

An error message or empty vector if no errors found.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Create a time bins object:
time_bins <- matrix(
  data = c(99.6, 93.5, 93.5, 89.3, 89.3, 85.8, 85.8, 83.5, 83.5, 70.6, 70.6, 65.5),
  ncol = 2,
  byrow = TRUE,
  dimnames = list(
    c("Cenomanian", "Turonian", "Coniacian", "Santonian", "Campanian", "Maastrichtian"),
    c("fad", "lad")
  )
)

# Check that this is a valid timeBins object (will return error message as class
# is not set):
check_timeBins(time_bins = time_bins)
```

---

classify_costmatrix *Classify a costmatrix character*

Description

Given a costmatrix, classifies it into one of twelve distinct types.

Usage

```
classify_costmatrix(costmatrix)
```

Arguments

costmatrix	An object of class costMatrix.

Details

Hoyal Cuthill and Lloyd (in press) classified discrete characters into twelve different types. This function applies their classification to a costmatrix object. The twelve different types are defined below alongside an empirical example for each, following Hoyal Cuthill and Lloyd (in review).

Type I - Constant

The simplest character type - only one state

Example: Character 18 of Sterli et al. (2013)

Costmatrix:

```
    -----
    | 0 |
---------
| 0 | 0 |
---------
```

State graph:

```
0
```

Type II - Binary symmetric

Exactly two states with equal and finite transition costs between them.

Example: Character 1 of Sterli et al. (2013)

Costmatrix:

```
    ---------
    | 0 | 1 |
-------------
| 0 | 0 | 1 |
-------------
| 1 | 1 | 0 |
-------------
```

State graph:

```
  1
0---1
```

Type III - Multistate unordered

Three or more states where every state-to-state transition has equal and finite cost.

Example: Character 53 of Sterli et al. (2013)

Costmatrix:

```
    -------------
    | 0 | 1 | 2 |
-----------------
| 0 | 0 | 1 | 1 |
-----------------
| 1 | 1 | 0 | 1 |
-----------------
| 2 | 1 | 1 | 0 |
-----------------
```

State graph:

```
    1
   / \
 1/   \1
 /     \
0-------2
    1
```

Type IV - linear ordered symmetric

Three or more states arranged as a single path with all connected states having equal and finite transition cost.

Example: Character 7 of Sterli et al. (2013)

Costmatrix:

```
    -------------
    | 0 | 1 | 2 |
-----------------
| 0 | 0 | 1 | 2 |
-----------------
| 1 | 1 | 0 | 1 |
-----------------
| 2 | 2 | 1 | 0 |
-----------------
```

State graph:

```
  1   1
0---1---2
```

Type V - Non-linear ordered symmetric

Four or more states arranged into a state graph with at least one branching point (vertex of degree three or greater) with all connected states having equal and finite transition costs.

Example: Character 71 of Hooker (2014)

Costmatrix:

```
    -------------
    | 0 | 1 | 2 | 3 |
---------------------
| 0 | 0 | 1 | 2 | 2 |
---------------------
| 1 | 1 | 0 | 1 | 1 |
---------------------
| 1 | 2 | 1 | 0 | 2 |
---------------------
| 1 | 2 | 1 | 2 | 0 |
---------------------
```

State graph:

```
          2
         /
       1/
   1   /
0-----1
```

```
        \
        1\
          \
           3
```

Type VI - Binary irreversible

Two states where transitions in one direction have infinite cost and the other finite cost.

Example: Character 119 of Gunnell et al. (2018)

Costmatrix:

```
    ---------
    | 0 | 1 |
-------------
| 0 | 0 | 1 |
-------------
| 1 | i | 0 |
-------------
```

(i = infinity)

State graph:

```
  1
0-->1
```

Type VII - Multistate irreversible

Three or more states where transitions in one direction have infinite cost and the other finite cost.

Example: Character 21 of Gunnell et al. (2018)

Costmatrix:

```
    -------------
    | 0 | 1 | 2 |
-----------------
| 0 | 0 | 1 | 2 |
-----------------
| 1 | i | 0 | 1 |
-----------------
| 2 | i | i | 0 |
-----------------
```

(i = infinity)

State graph:

```
  1   1
0-->1-->2
```

Type VIII - Binary Dollo

Two states where transitions in one direction can be made at most once, with no restriction in the other direction.

Example: Character 10 of Paterson et al. (2014)

Costmatrix:

```
    ----------
    | 0 | 1 |
-------------
| 0 | 0 | D |
-------------
| 1 | 1 | 0 |
-------------
```

(D = a Dollo penalty applied to avoid multiple transitions, see Swofford and Olsen 1990)

State graph:

```
      1
 --- <-- ---
| 0 |   | 1 |
 --- --> ---
      D
```

Type IX - Multistate Dollo

Three or more states where transitions in one direction can be made at most once, with no restriction in the other direction.

Example: Character 15 of Paterson et al. (2014)

Costmatrix:

```
    --------------
    | 0 | 1 |  2 |
------------------
| 0 | 0 | D | 2D |
------------------
| 1 | 1 | 0 |  D |
------------------
| 2 | 2 | 1 |  0 |
------------------
```

(D = a Dollo penalty applied to avoid multiple transitions, see Swofford and Olsen 1990)

State graph:

```
      1       1
 --- <-- --- <-- ---
| 0 |   | 1 |   | 2 |
 --- --> --- --> ---
      D       D
```

Type X - Multistate symmetric

Three or more states where connected states have symmetric finite costs, but where these are not all equal.

Example: Character 8 of Sumrall abd Brett (2002)

Costmatrix:

```
    -------------------------
    | 0 | 1 | 2 | 3 | 4 | 5 |
-----------------------------
| 0 | 0 | 1 | 2 | 3 | 2 | 3 |
-----------------------------
| 1 | 1 | 0 | 3 | 2 | 1 | 2 |
-----------------------------
| 2 | 2 | 3 | 0 | 3 | 2 | 1 |
-----------------------------
| 3 | 3 | 2 | 3 | 0 | 1 | 2 |
-----------------------------
| 4 | 2 | 1 | 2 | 1 | 0 | 1 |
-----------------------------
| 5 | 3 | 2 | 1 | 2 | 1 | 0 |
-----------------------------
```

State graph:

```
   1
 2---5   3
 |    \  |
2|    1\ |1
 |      \|
 0---1---4
   1   1
```

Type XI - Binary asymmetric

Two states where transition costs are finite but non-equal.

Example: Character 3 of Gheerbrant et al. (2014)

Costmatrix:

```
    ----------
    |  0 | 1 |
--------------
| 0 |  0 | 1 |
--------------
| 1 | 10 | 0 |
--------------
```

State graph:

```
      10
 --- <--- ---
| 0 |    | 1 |
 --- ---> ---
      1
```

Type XII - Multistate asymmetric

Three or more states where at least one transition is asymmetric, but the character does not meet the definition of other multistate asymmetric characters (i.e., Type VII and IX).

Example: Character 7 of Gheerbrant et al. (2014)

Costmatrix:

```
    ---------------
    |  0 |  1 | 2 |
-------------------
| 0 |  0 |  1 | 1 |
-------------------
| 1 |  1 |  0 | 1 |
-------------------
| 1 | 10 | 10 | 0 |
-------------------
```

State graph:

```
          10
    ______________
   /              \
  L   1       10   \
 --- <-- --- <--- ---
| 0 |   | 1 |    | 2 |
 --- --> --- ---> ---
  \   1        1   ^
   \______________/
           1
```

Other character types

The classification of Hoyal Cuthill and Lloyd (in review) was derived as a means to delineate a mathematical problem. As such it may be of use to consider where some special kinds of character would fall in this classification.

Stratigraphic characters

These would be classified as either Type VI, Type VII, Type XI, or Type XII characters (depending on state count and spacing of stratigraphic units).

Gap-weighted characters

A gap-weighted (Thiele 1993) character would be classified as either Type II or Type IV (depending on state count), but differ in the implicit assumption of whether all states are expected to be sampled.

For a gap-weighted character there is no presumption that every state be sampled as the spacing between states is the primary information being considered.

Continuous characters

Continuous characters are necessarily not classifiable here as they are by definition non-discrete (except where translated into a gap-weighted character, above).

Value

A text string indicating the classification result (e.g., `"Type VIII"`).

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Gunnell, G. F., Boyer, D. M., Friscia, A. R., Heritage, S., Manthi, F. K., Miller, E. R., Sallam, H. M., Simmons, N. B., Stevens, N. J. and Seiffert, E. R., 2018. Fossil lemurs from Egypt and Kenya suggest an African origin for Madagascar's aye-aye. *Nature Communications*, **9**, 3193.

Hooker, J. J., 2014. New postcranial bones of the extinct mammalian family Nyctitheriidae (Paleogene, UK): primitive euarchontans with scansorial locomotion. *Palaeontologia Electronica*, **17.3.47A**, 1-82.

Hoyal Cuthill, J. F. and Lloyd, G. T., in press. Measuring homoplasy I: comprehensive measures of maximum and minimum cost under parsimony across discrete cost matrix character types. *Cladistics*, bold, .

Paterson, A. M., Wallis, G. P., Kennedy, M. and Gray, R. D., 2014. Behavioural evolution in penguins does not reflect phylogeny. *Cladistics*, **30**, 243-259.

Sterli, J., Pol, D. and Laurin, M., 2013. Incorporating phylogenetic uncertainty on phylogeny-based palaeontological dating and the timing of turtle diversification. *Cladistics*, **29**, 233-246.

Sumrall, C. D. and Brett, C. E., 2002. A revision of *Novacystis hawkesi* Paul and Bolton 1991 (Middle Silurian: Glyptocystitida, Echinodermata) and the phylogeny of early callocystitids. *Journal of Paleontology*, **76**, 733-740.

Swofford, D. L. and Olsen, G. J., 1990. Phylogeny reconstruction. In D. M. Hillis and C. Moritz (eds.), *Molecular Systematics*. Sinauer Associates, Sunderland. pp411-501.

Thiele, K.. 1993. The Holy Grail of the perfect character: the cladistic treatment of morphometric data. *Cladistics*, **9**, 275-304.

Examples

```
# Create a Type I character costmatrix:
constant_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 0,
  character_type = "unordered"
)

# Classify costmatrix:
```

```
classify_costmatrix(costmatrix = constant_costmatrix)

# Create a Type II character costmatrix:
binary_symmetric_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 1,
  character_type = "unordered"
)

# Classify costmatrix:
classify_costmatrix(costmatrix = binary_symmetric_costmatrix)

# Create a Type III character costmatrix:
unordered_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state= 2,
  character_type = "unordered"
)

# Classify costmatrix:
classify_costmatrix(costmatrix = unordered_costmatrix)

# Create a Type IV character costmatrix:
linear_ordered_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state= 2,
  character_type = "ordered"
)

# Classify costmatrix:
classify_costmatrix(costmatrix = linear_ordered_costmatrix)

# Create a Type V character costmatrix:
nonlinear_ordered_costmatrix <- convert_adjacency_matrix_to_costmatrix(
  adjacency_matrix = matrix(
    data = c(
      0, 1, 0, 0,
      1, 0, 1, 1,
      0, 1, 0, 0,
      0, 1, 0, 0
    ),
    nrow = 4,
    dimnames = list(0:3, 0:3)
  )
)

# Classify costmatrix:
classify_costmatrix(costmatrix = nonlinear_ordered_costmatrix)

# Create a Type VI character costmatrix:
binary_irreversible_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state= 1,
```

```
  character_type = "irreversible"
)

# Classify costmatrix:
classify_costmatrix(costmatrix = binary_irreversible_costmatrix)

# Create a Type VII character costmatrix:
multistate_irreversible_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state= 2,
  character_type = "irreversible"
)

# Classify costmatrix:
classify_costmatrix(costmatrix = multistate_irreversible_costmatrix)

# Create a Type VIII character costmatrix:
binary_dollo_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state= 1,
  character_type = "dollo"
)

# Classify costmatrix:
classify_costmatrix(costmatrix = binary_dollo_costmatrix)

# Create a Type IX character costmatrix:
multistate_dollo_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state= 2,
  character_type = "dollo"
)

# Classify costmatrix:
classify_costmatrix(costmatrix = multistate_dollo_costmatrix)

# Create a Type X character costmatrix:
multistate_symmetric_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state= 5,
  character_type = "ordered"
)
multistate_symmetric_costmatrix$type <- "custom"
multistate_symmetric_costmatrix$costmatrix <- matrix(
  data = c(
    0, 1, 2, 3, 2, 3,
    1, 0, 3, 2, 1, 2,
    2, 3, 0, 3, 2, 1,
    3, 2, 3, 0, 1, 2,
    2, 1, 2, 1, 0, 1,
    3, 2, 1, 2, 1, 0
  ),
  nrow = multistate_symmetric_costmatrix$size,
```

```
  ncol = multistate_symmetric_costmatrix$size,
  byrow = TRUE,
  dimnames = list(
    multistate_symmetric_costmatrix$single_states,
    multistate_symmetric_costmatrix$single_states
  )
)

# Classify costmatrix:
classify_costmatrix(costmatrix = multistate_symmetric_costmatrix)

# Create a Type XI character costmatrix:
binary_asymmetric_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state= 1,
  character_type = "ordered"
)
binary_asymmetric_costmatrix$type <- "custom"
binary_asymmetric_costmatrix$costmatrix <- matrix(
  data = c(
    0, 1,
    10, 0
  ),
  nrow = binary_asymmetric_costmatrix$size,
  ncol = binary_asymmetric_costmatrix$size,
  byrow = TRUE,
  dimnames = list(
    binary_asymmetric_costmatrix$single_states,
    binary_asymmetric_costmatrix$single_states
  )
)
binary_asymmetric_costmatrix$symmetry <- "Asymmetric"

# Classify costmatrix:
classify_costmatrix(costmatrix = binary_asymmetric_costmatrix)

# Create a Type XII character costmatrix:
multistate_asymmetric_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state= 2,
  character_type = "ordered"
)
multistate_asymmetric_costmatrix$type <- "custom"
multistate_asymmetric_costmatrix$costmatrix <- matrix(
  data = c(
    0, 1, 1,
    1, 0, 1,
    10, 10, 0
  ),
  nrow = multistate_asymmetric_costmatrix$size,
  ncol = multistate_asymmetric_costmatrix$size,
  byrow = TRUE,
  dimnames = list(
```

```
    multistate_asymmetric_costmatrix$single_states,
    multistate_asymmetric_costmatrix$single_states
  )
)
multistate_asymmetric_costmatrix$symmetry <- "Asymmetric"

# Classify costmatrix:
classify_costmatrix(costmatrix = multistate_asymmetric_costmatrix)
```

---

compactify_cladistic_matrix

Collapses matrix to unique character state distributions

Description

Collapses a cladistic matrix to just unique character state distributions and taxon names.

Usage

```
compactify_cladistic_matrix(cladistic_matrix, message = TRUE)
```

Arguments

`cladistic_matrix`
: The cladistic matrix in the format imported by read_nexus_matrix.

`message`
: Logical indicating whether or not a message should be printed to the screen if the matrix cannot be compactified.

Details

Important: not recommended for general use.

This function is intended to make a matrix with redundant character state distributions smaller by collapsing these to single characters and upweighting them accordingly. It is intended purely for use with MRP matrices, but may have some very restricted uses elsewhere.

The function also deletes any characters weighted zero from the matrix and will merge duplicate taxon names into unique character strings.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

See Also

build_cladistic_matrix, prune_cladistic_matrix, read_nexus_matrix, safe_taxonomic_reduction, write_nexus_matrix, write_tnt_matrix

Examples

```
# Examine the matrix pre-compactification:
michaux_1989$matrix_1$matrix

# Examine the weights pre-compactification:
michaux_1989$matrix_1$character_weights

# Compactify the matrix:
michaux_1989compact <- compactify_cladistic_matrix(michaux_1989)

# Examine the matrix post-compactification:
michaux_1989compact$matrix_1$matrix

# Examine the weights post-compactification:
michaux_1989compact$matrix_1$character_weights
```

---

convert_adjacency_matrix_to_costmatrix

Converts an adjacency matrix to a costmatrix

Description

Takes an adjacency matrix as input and returns the corresponding costmatrix.

Usage

```
convert_adjacency_matrix_to_costmatrix(adjacency_matrix)
```

Arguments

`adjacency_matrix`
: A labelled square matrix with zeroes denoting non-adjacencies and ones denoting adjacencies.

Details

This function is intended for internal use, but as it also generalizes to solving a general graph theory problem - generating a distance matrix corresponding to each shortest path between vertices of a connected graph represented as an adjacency matrix - it is made available explicitly here.

The process is best understood with an example. Imagine we have a graph like this:

```
0-1-2
|
3
```

I.e., we have four labelled vertices, 0-3, and three edges (connections) between them:

```
0-1
1-2
0-3
```

Note: here we assume symmetry, 0-1 = 1-0.

Graphs like this can be explicitly captured as adjacency matrices, where a one denotes two vertices are "adjacent" (connected by an edge) and a zero that they are not.

```
    _________________
    | 0 | 1 | 2 | 3 |
---------------------
| 0 | 0 | 1 | 0 | 1 |
---------------------
| 1 | 1 | 0 | 1 | 0 |
---------------------
| 2 | 0 | 1 | 0 | 0 |
---------------------
| 3 | 1 | 0 | 0 | 0 |
---------------------
```

But what such matrices do not tell us is how far every vertex-to-vertex path is in terms of edge counts. E.g., the path length from vertex 3 to vertex 2.

This function simply takes the adjacency matrix and returns the corresponding costmatrix, corresponding to every minimum vertex-to-vertex path length.

Value

An object of class `costMatrix`.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

See Also

convert_state_tree_to_adjacency_matrix, locate_bracket_positions

Examples

```
# Build the example adjacency matrix for the graph above:
adjacency_matrix <- matrix(
  data = c(
    0, 1, 0, 1,
    1, 0, 1, 0,
    0, 1, 0, 0,
    1, 0, 0, 0
  ),
  nrow = 4,
  ncol = 4,
```

```
  dimnames = list(0:3, 0:3)
)

# Convert this to a costmatrix:
convert_adjacency_matrix_to_costmatrix(
  adjacency_matrix = adjacency_matrix
)
```

---

convert_costmatrix_to_stategraph

Convert a costmatrix to a minimal state graph

Description

Given a costmatrix, returns the smallest possible state graph (fewest arcs).

Usage

```
convert_costmatrix_to_stategraph(costmatrix)
```

Arguments

costmatrix	An object of class costMatrix.

Details

A costmatrix summarises all possible state-to-state transition costs and hence each entry can also be considered as an arc of a directed state graph. However, many of these arcs could be removed and a complete description of the graph still be provided. For example, the diagonal (any transition from a state to itself - a loop) can be removed, as can any arc with infinite cost (as this arc would never be traversed in practice). Finally, some arcs are redundant as indirect paths already represent the same cost.

As an example, we can consider the linear ordered costmatrix:

```
    -------------
    | 0 | 1 | 2 |
-----------------
| 0 | 0 | 1 | 2 |
-----------------
| 1 | 1 | 0 | 1 |
-----------------
| 2 | 2 | 1 | 0 |
-----------------
```

A maximum directed graph representation would thus be:

```
----------------------
| from | to | weight |
----------------------
|   0  | 0  |   0    |
----------------------
|   0  | 1  |   1    |
----------------------
|   0  | 2  |   2    |
----------------------
|   1  | 0  |   1    |
----------------------
|   1  | 1  |   0    |
----------------------
|   1  | 2  |   1    |
----------------------
|   2  | 0  |   2    |
----------------------
|   2  | 1  |   1    |
----------------------
|   2  | 2  |   0    |
----------------------
```

But the following description is still complete, and minimal:

```
----------------------
| from | to | weight |
----------------------
|   0  | 1  |   1    |
----------------------
|   1  | 0  |   1    |
----------------------
|   1  | 2  |   1    |
----------------------
|   2  | 1  |   1    |
----------------------
```

This function effectively generates the latter (the minimal directed graph representation as a matrix of arcs).

Value

An object of class `stateGraph`.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

See Also

convert_adjacency_matrix_to_costmatrix

Examples

```
# Make a six-state unordered character costmatrix:
unordered_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state= 5,
  character_type = "unordered"
)

# Find the minimal directed graph representation:
convert_costmatrix_to_stategraph(costmatrix = unordered_costmatrix)

# Make a six-state ordered character costmatrix:
ordered_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state= 5,
  character_type = "ordered"
)

# Find the minimal directed graph representation:
convert_costmatrix_to_stategraph(costmatrix = ordered_costmatrix)

# Make a six-state stratigraphic character costmatrix:
stratigraphic_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state= 5,
  character_type = "stratigraphy",
  state_ages = c(103, 91.4, 78.2, 73.4, 66.0, 59.7)
)

# Find the minimal directed graph representation:
convert_costmatrix_to_stategraph(costmatrix = stratigraphic_costmatrix)
```

---

convert_stategraph_to_costmatrix

Convert a minimal state graph to a costmatrix

Description

Given a state graph returns the corresponding costmatrix.

Usage

```
convert_stategraph_to_costmatrix(stategraph)
```

Arguments

stategraph	An object of class stateGraph.

Details

A state graph describes the relationship between character states in terms of a graph of vertices (character states) and edges (weights). Edges can be "symmetric" (the graph is undirected) or "asymmetric" (the graph is directed, a digraph).

For example, a simple symmetric binary state graph might look ike this:

```
  1
A---B
```

Here the two states are A and B and the weight (cost of transitioning from A to B or B to A) is one.

In Claddis this graph can be represented using a `stateGraph` object including the arcs that describe the (di)graph:

```
from to weight
   0  1      1
   1  0      1
```

Each row represents an arc from one vertex to another, and the corresponding weight (transition cost). Note that for symmetric graphs the edge is stated using two arcs (from *i* to *j* and from *j* to *i*).

This function converts these state transitions to costmatrices, including interpolating any missing transitions by using the shortest indirect cost. In the example used here the costmatrix would look quite simple:

```
    ---------
    | 0 | 1 |
-------------
| 0 | 0 | 1 |
-------------
| 1 | 1 | 0 |
-------------
```

Value

An object of class `costMatrix`.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

See Also

convert_adjacency_matrix_to_costmatrix convert_costmatrix_to_stategraph

Examples

```
# Make state graph for a six-state linear ordered character:
stategraph <- list(
  n_vertices = 6,
  n_arcs = 10,
  n_states = 6,
  single_states = c("0", "1", "2", "3", "4", "5"),
  type = "ordered",
  arcs = data.frame(
    from = c("1", "0", "2", "1", "3", "2", "4", "3", "5", "4"),
    to = c("0", "1", "1", "2", "2", "3", "3", "4", "4", "5"),
    weight = c(1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
  ),
  vertices = data.frame(
    label = c("0", "1", "2", "3", "4", "5"),
    in_degree = c(1, 2, 2, 2, 2, 1),
    out_degree = c(1, 2, 2, 2, 2, 1),
    eccentricity = c(5, 4, 3, 3, 4, 5),
    periphery = c(1, 0, 0, 0, 0, 1),
    centre = c(0, 0, 1, 1, 0, 0)
  ),
  radius = 3,
  diameter = 5,
  adjacency_matrix = matrix(
    data = c(
      0, 1, 0, 0, 0, 0,
      1, 0, 1, 0, 0, 0,
      0, 1, 0, 1, 0, 0,
      0, 0, 1, 0, 1, 0,
      0, 0, 0, 1, 0, 1,
      0, 0, 0, 0, 1, 0
    ),
    ncol = 6,
    byrow = TRUE,
    dimnames = list(
      c("0", "1", "2", "3", "4", "5"),
      c("0", "1", "2", "3", "4", "5")
    )
  ),
  directed = FALSE,
  includes_polymorphisms = FALSE,
  polymorphism_costs = "additive",
  polymorphism_geometry = "simplex",
  polymorphism_distance = "euclidean",
  includes_uncertainties = FALSE,
  pruned = FALSE,
  dollo_penalty = 999,
  base_age = 1,
  weight = 1
)

# Set calss as stateGraph:
```

```
class(x = stategraph) <- "stateGraph"

# View state graph:
stategraph

# Convert state graph to a costmatrix:
costmatrix <- convert_stategraph_to_costmatrix(stategraph = stategraph)

# Show costmatrix reflects linear ordered costs:
costmatrix$costmatrix
```

`convert_state_tree_to_adjacency_matrix`
Converts a character state tree to an adjacency matrix

Description

Takes a character state tree as input and returns the corresponding adjacency matrix.

Usage

```
convert_state_tree_to_adjacency_matrix(state_tree)
```

Arguments

`state_tree`	A text string describing a character state tree.

Details

There are multiple ways to define discrete character states for use in phylogenetic analysis or ancestral state reconstruction in terms of transitions between states. The two most common being "ordered" ((Wagner optimization; Farris, 1970) or "unordered" (Fitch optimization; Fitch, 1971). These can be sonsidered as representing simple Markov models with symmetric transition probabilities, with the only difference being whether any direct state-tostae transition is possible (unordered) or only immediately "adjacent" states can be transitioned too.

In practice, however, there may be reason to believe in more complex transition models. For an example, see various characters in Hooker (2014). If these are still symmetric (i.e., going from state X to state Y is just as probable as going from state Y to state X) then they can be expressed using a "character state tree". For example, the model:

```
0-1-2
|
3
```

This model has four states (labelled 0-3). However, it is clearly not an ordered model:

```
0-1-2-3
```

Where only adjacent state transitions are possible, e.g., to get from state 0 to state 2 requires passing through state 1.

Nor is it an unordered model:

```
0---1
|\ /|
| X |
|/ \|
2---3
```

Where any state-to-state transition is possible.

In other words we cannot simply label this model as "ordered" or "unordered". It can, however, be expressed directly as character state tree:

```
(3,(2)1)0
```

Superficially these appear very similar to Newick strings (ways or representing bifurcating or multifurcating trees of species). However, the major difference is that with Newick strings we often only label the "tips" (terminal nodes, or vertices of degree one). Here there are no unlabelled "internal nodes", also known as "hypothetical ancestors". Instead, numbers after parentheses indicate the label of that node (vertex).

Thus we can read the above tree as node 2 is only directly connected to node 1, 3 and 1 are directly connected to node zero.

This is a very compact way of representing a Markov model, but unfortunately it is hard to read and difficult to do anything quantitative with. Thus this function takes such text strings and converts them to a more easily usable graph representation: the adjacency matrix.

Adjacency matrices represent graphs by showing links between vertices as values of one in symmetric square matrices. These always have a diagonal of zero (you cannot be adjacent to yourself) and any other vertices that are not directly linked are also coded as zero.

Note that here such matrices are also considered symmetric and hence the diagonal is also a line of reflection.

In practice more complex relationships may be desired, including differential weighting of specific transitions (without additional intermediate states; where going from state X to state Y has a cost other than zero or one), or asymmetric models (where going from state X to state Y has a different "cost" than going from state Y to state X). These are still possible in Claddis, and phylogenetic analysis in general, but require "costmatrices" to define them. (Character state trees are insufficent.)

Thus, costmatrices (or their probabilistic equivalent, Q-matrices) are the only generalisable form of defining *any* Markov model.

The output from this function can also be represented as a costmatrix with convert_adjacency_matrix_to_costmatrix.

Value

A symmetric adjacency matrix indicating which states are linked (value 1) by an edge.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Farris, J. S., 1970. Methods for computing Wagner trees. *Systematic Zoology*, **19**, 83-92.

Fitch, W. M., 1971. Towards defining the course of evolution: minimum change for a specified tree topology. *Systematic Zoology*, **20**, 406-416.

Hooker, J. J., 2014. New postcranial bones of the extinct mammalian family Nyctitheriidae (Paleogene, UK): primitive euarchontans with scansorial locomotion. *Palaeontologia Electronica*, **17.3.47A**, 1-82.

See Also

convert_adjacency_matrix_to_costmatrix, locate_bracket_positions

Examples

```
# Convert a simple state tree to an adjacency matrix:
convert_state_tree_to_adjacency_matrix(
 state_tree = "(3,(2)1)0"
)

# Convert a more complex state tree to an adjacency matrix:
convert_state_tree_to_adjacency_matrix(
  state_tree = "(((5)4)3,(2)1)0"
)
```

---

count_cherries *Counts the number of cherries in a tree*

Description

Given a set of phylogenetic tree(s) returns the number of cherries in each one.

Usage

```
count_cherries(tree)
```

Arguments

tree	A tree (phylo or multiPhylo object).

Details

Cherries are components of a phylogenetic tree defined as internal nodes with exactly two terminal descendants.

This function simply counts the number present in a given tree.

Note that any fully dichotomous phylogenetic tree must have at least one cherry.

Value

Returns a vector of cherry counts for each tree retaining the order in which they were supplied.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Create simple two-cherry tree:
tree <- ape::read.tree(text = "((A,B),(C,D));")

# Show count of cherries is two:
count_cherries(tree = tree)

# Create a star tree:
tree <- ape::read.tree(text = "(A,B,C,D);")

# Show count of cherries is zero:
count_cherries(tree = tree)
```

---

date_nodes *Returns node ages for a time-scaled tree*

Description

Given a tree with branch-lengths scaled to time and a value for `$root.time` will return a vector of node ages.

Usage

```
date_nodes(time_tree)
```

Arguments

`time_tree`	A tree (phylo object) with branch lengths representing time and a value for `$root.time`.

Details

Returns a vector of node ages (terminal and internal) labelled by their node number.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Create simple four-taxon tree with edge lengths all
# set to 1 Ma:
time_tree <- ape::read.tree(text = "(A:1,(B:1,(C:1,D:1):1):1);")

# Set root.time as 10 Ma:
time_tree$root.time <- 10

# Get node ages:
date_nodes(time_tree = time_tree)
```

---

day_2016 *Character-taxon matrix from Day et al. 2016*

Description

The character-taxon matrix from Day et al. (2016).

Format

A character-taxon matrix in the format imported by read_nexus_matrix.

References

Day, M. O., Rubidge, B. S. and Abdala, F., 2016. A new mid-Permian burnetiamorph therapsid from the Main Karoo Basin of South Africa and a phylogenetic review of Burnetiamorpha. *Acta Palaeontologica Polonica*, **61**, 701-719.

---

drop_time_tip *Drop tips from a time-scaled tree*

Description

Drop tips from a time-scaled tree and update root.time accordingly

Usage

```
drop_time_tip(time_tree, tip_names, ...)
```

Arguments

time_tree	A time-scaled tree in phylo format where branch lengths are durations and where a `$root.time` value indicates the root age.
tip_names	A vector of tip names to be pruned from the tree.
...	Additional options to be passed to `ape::drop.tip`.

Details

(NB: This function is designed to only cope with trees containing at least three tips.)

Usually ape formatted trees are pruned with the drop.tip function in ape. However, trees time-scaled using either the `paleotree` or `strap` packages have an additional important component, the root age (`$root.time`) that may need updating when tips are removed. (See fix_root_time.) Thus this function is a modified version of drop.tip that also performs the fix_root_time step.

Note that `dropPaleoTip` in the `paleotree` package performs the exact same function, but is not called here to reduce the number of dependencies for `Claddis`.

Value

Returns a tree (phylo object) with pruned tips and corrected `$root.time`.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

See Also

fix_root_time

Examples

```
# Create a simple four-taxon tree with branch lengths:
tree <- ape::read.tree(text = "(A:1,(B:1,(C:1,D:1):1):1);")

# Set root age as 20 Ma:
tree$root.time <- 20

# Now prune taxon A:
pruned_tree <- ape::drop.tip(phy = tree, tip = "A")

# Show that drop.tip has not updated the tree's root time:
pruned_tree$root.time

# Use the function to fix the root time:
pruned_tree <- drop_time_tip(time_tree = tree, tip_names = "A")

# Show that the root time is now fixed (19 Ma):
pruned_tree$root.time
```

---

estimate_ancestral_states

Ancestral Character State Estimation

Description

Given a tree and a cladistic matrix uses likelihood to estimate the ancestral states for every character.

Usage

```
estimate_ancestral_states(
  cladistic_matrix,
  time_tree,
  estimate_all_nodes = FALSE,
  estimate_tip_values = FALSE,
  inapplicables_as_missing = FALSE,
  polymorphism_behaviour = "equalp",
  uncertainty_behaviour = "equalp",
  threshold = 0.01,
  all_missing_allowed = FALSE
)
```

Arguments

`cladistic_matrix`
: A character-taxon matrix in the format imported by read_nexus_matrix.

`time_tree`
: A tree (phylo object) with branch lengths that represents the relationships of the taxa in `cladistic_matrix`.

`estimate_all_nodes`
: Logical that allows the user to make estimates for all ancestral values. The default (`FALSE`) will only make estimates for nodes that link coded terminals (recommended).

`estimate_tip_values`
: Logical that allows the user to make estimates for tip values. The default (`FALSE`) will only makes estimates for internal nodes (recommended).

`inapplicables_as_missing`
: Logical that decides whether or not to treat inapplicables as missing (TRUE) or not (FALSE, the default and recommended option).

`polymorphism_behaviour`
: One of either "equalp" or "treatasmissing".

`uncertainty_behaviour`
: One of either "equalp" or "treatasmissing".

`threshold`
: The threshold value to use when collapsing marginal likelihoods to discrete state(s).

`all_missing_allowed`
: Logical to allow all missing character values (generally not recommended, hence default is FALSE).

Details

At its' core the function uses either the rerootingMethod (Yang et al. 1995) as implemented in the phytools package (for discrete characters) or the ace function in the ape package (for continuous characters) to make ancestral state estimates. For discrete characters these are collapsed to the most likely state (or states, given equal likelihoods or likelihood within a defined `threshold` value). In the latter case the resulting states are represented as an uncertainty (i.e., states separated by a slash, e.g., 0/1). This is the method developed for Brusatte et al. (2014).

The function can deal with ordered or unordered characters and does so by allowing only indirect transitions (from 0 to 2 must pass through 1) or direct transitions (from 0 straight to 2), respectively. However, more complex costmatrix transitions are not currently supported.

Ancestral state estimation is complicated where polymorphic or uncertain tip values exist. These are not currently well handled here, although see the `fitpolyMk` function in phytools for a way these could be dealt with in future. The only available options right now are to either treat multiple states as being equally probable of the "true" tip state (i.e., a uniform prior) or to avoid dealing with them completely by treating them as missing (NA) values.

It is also possible to try to use phylogenetic information to infer missing states, both for internal nodes (e.g., those leading to missing tip states) and for tips. This is captured by the `estimate_all_nodes` and `estimate_tip_values` options. These have been partially explored by Lloyd (2018), who cuationed against their use.

Value

The function will return the same `cladistic_matrix`, but with two key additions: 1. Internal nodes (numbered by ape formatting) will appear after taxa in each matrix block with estimated states coded for them, and 2. The time-scaled tree used will be added to `cladistic_matrix` as `cladistic_matrix$topper$tree`. Note that if using the `estimate_tip_values = TRUE` option then tip values may also be changed from those provided as input.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com> and Thomas Guillerme <guillert@tcd.ie>

References

Brusatte, S. L., Lloyd, G. T., Wang, S. C. and Norell, M. A., 2014. Gradual assembly of avian body plan culminated in rapid rates of evolution across dinosaur-bird transition. *Current Biology*, 24, 2386-2392.

Lloyd, G. T., 2018. Journeys through discrete-character morphospace: synthesizing phylogeny, tempo, and disparity. *Palaeontology*, **61**, 637-645.

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

Examples

```
# Set random seed:
set.seed(4)

# Generate a random tree for the Day data set:
time_tree <- ape::rtree(n = nrow(day_2016$matrix_1$matrix))

# Update taxon names to match those in the data matrix:
time_tree$tip.label <- rownames(x = day_2016$matrix_1$matrix)

# Set root time by making youngest taxon extant:
time_tree$root.time <- max(diag(x = ape::vcv(phy = time_tree)))
```

```
# Use Day matrix as cladistic matrix:
cladistic_matrix <- day_2016

# Prune most characters out to make example run fast:
cladistic_matrix <- prune_cladistic_matrix(cladistic_matrix,
  characters2prune = c(2:3, 5:37)
)

# Estimate ancestral states:
estimate_ancestral_states(
  cladistic_matrix = cladistic_matrix,
  time_tree = time_tree
)
```

`estimate_squared_change_ancestors` *Estimate ancestral states for a continuous character under squared-change parsimony*

Description

Given a phylogeny and set of continuous tip values, returns a single estimate of ancestral states that minimise the total squared-change cost on the tree.

Usage

```
estimate_squared_change_ancestors(tree, tip_values)
```

Arguments

`tree`	A tree (object of class `phylo`) with or without branch lengths and having tip labels that match the elements of `tip_values`.
`tip_values`	A labelled vector of continuous tip values, where the labels match the tip labels in `tree`. Tip values can be single numbers or a minimum-maximum range, see details.

Details

Multiple algorithms exist for estimating ancestral states for a univariate continuous character. This function specifically provides an ancestral state estimation that minimises the squared-change cost between each internal node and every other node that node is directly connected to. Note: in practice this approach can be identical to maximum likelihood ancestral state estimation under Brownian motion (e.g., Maddison 1991), leading Goloboff (2022) to argue that it isn't strictly a parsimony approach. However, this function extends squared-change parsimony to allow for ranges in tip values (see below) as well as implementing squared-change parsimony where branch lengths are variable and (hard) polytomies are permitted. As such it may still be of use to some users as distinct from other ancestral state estimation approaches for continuous characters.

Algorithm

Although squared-change parsimony ancestral state estimates can be directly calculated using the approach of Maddison (1991), the algorithm used here is an optimisation approach as this allows tip ranges to be accommodated (see below). In practice this leads to minimal speed reduction in most cases as an initial estimate is made using the `fastAnc` function from the phytools package (Revell 2024) that will frequently be identical to the final solution (except where ranges in tip values are used).

The optimisation used here is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm which was independently developed by Broyden (1970), Fletcher (1970), Goldfarb (1970), and Shanno (1970) and is implemented as the `"L-BFGS-B"` method in the R `stats` function `optim()`. The choice of this method is what allows ranges in tip values to be accommodated by treating these as an additional parameter to be estimated but with box constraints (i.e., minimum and maximum possible values - the ranges for the tip value) applied.

Ranges in tip values

Unlike other implementations of squared-change parsimony (to the best of my knowledge at least) this function removes the constraint that each tip must be represented by a single numeric value. Instead, any number of tip values can instead be represented by a range of values (i.e., a minimum and maximum value). These must be specified in the `tip_values` variable by separating the minimum and maximum values with an underscore character (_). E.g., `"0.33_0.53"` is the range 0.33 to 0.53. Note: an underscore is used rather than a dash as negative values are permitted and the dash symbol is reserved to indicate these.

In practice ranges are treated as an additional paramater to be estimated (alongside the internal node estimates) with the same criterion of minimising the total length, or cost (cum of squares), for the tree as a whole. As such this is truly a parsimony algorithm and negates the criticism of Goloboff (2022) that squared-change parsimony is not a true parsimony algorithm and also differs from (say) a maximum likelihood estimate that assumes a central tendency (i.e., midpoint) to the range of values at a tip.

The estimated tip values can also be examined a posteriori using the `tip_values` value from the function output.

Polytomies in the input tree

The function can also handle polytomies in the input tree, albeit it only does so by treating these as "hard" polytomies. I.e., it does not permute the minimum sum of squares for every possible resolution of a polytomy.

"Weighted" squared-change parsimony and branch lengths in the input tree

By default the function will apply so-called "unweighted" squared-change parsimony, meaning the user does not need to provide an input tree with branch lengths (only the topology). However, in practice "unweighted" parsimony really means treating every branch as having unit length (which is still *a* weighting). However, a user may wish to estimate ancestral states where variable branch lengths are accounted for (e.g., such that a cherry with variable terminal branch lengths means the subtending node is more strongly influenced by the tip value at the end of the shorter terminal).

In practice the function will implement weighted squared-change parsimony whenever the input tree already has edge-lengths applied. As such if the user does wish to implement "unweighted" squared-change parsimony they should be careful to supply an input tree where there are either no edge lengths or every edge is set to length one.

Value

The function returns a list with three elements:

`ancestral_state_estimates`
: A single set of ancestral state estimates that represent a solution that minimises the total cost, or length, of the tree.

`tip_values`
: The (estimated) tip values that minimise the sum of squares. This will be identical to the `tip_values` input if no ranges in tip values were used, but otherwise will show a single tip value that minimised the total cost, or length, of the tree.

`sum_of_squares`
: A single numeric value indicating the total cost, or length, of the tree which is the sum of the squared length of each edge of the tree.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Broyden, C. G., 1970. The convergence of a class of double-rank minimization algorithms. *Journal of the Institute of Mathematics and Its Applications*, **6**, 76-90.

Fletcher, R., 1970. A new approach to variable metric algorithms. *Computer Journal*, **13**, 317-322.

Goloboff, P. A., 2022. *Phylogenetic Analysis of Morphological Data, Volume 2: Refining Phylogenetic Analyses*. CRC Press, Boca Raton. 291 pp.

Goldfarb, D., 1970. A family of variable metric updates derived by variational means. *Mathematics of Computation*, **24**, 23-26.

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

Revell, L. J., 2024. phytools 2.0: an updated R ecosystem for phylogenetic comparative methods (and other things). *PeerJ*, **12**, e16505.

Shanno, D. F., 1970. Conditioning of quasi-Newton methods for function minimization. *Mathematics of Computation*, **24**, 647-656.

Examples

```
# Estimate squared-change parsimony for four tips with single values:
scp <- estimate_squared_change_ancestors(
  tree = ape::read.tree(text = "((A,B),(C,D));"),
  tip_values = tip_values <- c("A" = 3.6, "B" = 4.1, "C" = 7.3, "D" = 8.8)
)

# Plot results:
ape::plot.phylo(
  x = ape::read.tree(text = "((A,B),(C,D));"),
  main = paste("Sum of squares =", scp$sum_of_squares)
)
ape::tiplabels(text = scp$tip_values)
ape::nodelabels(text = round(x = scp$ancestral_state_estimates, digits = 2))
```

```
# Estimate squared-change parsimony for four tips with a ranged value:
scp <- estimate_squared_change_ancestors(
  tree = ape::read.tree(text = "((A,B),(C,D));"),
  tip_values = tip_values <- c("A" = "3.6", "B" = "4.1", "C" = "7.3", "D" = "8.8_9.3")
)

# Plot results:
ape::plot.phylo(
  x = ape::read.tree(text = "((A,B),(C,D));"),
  main = paste("Sum of squares =", scp$sum_of_squares)
)
ape::tiplabels(text = scp$tip_values)
ape::nodelabels(text = round(x = scp$ancestral_state_estimates, digits = 2))

# Estimate weighted squared-change parsimony for four tips with single values:
scp <- estimate_squared_change_ancestors(
  tree = ape::read.tree(text = "((A:1,B:2):2,(C:2,D:3):4);"),
  tip_values = tip_values <- c("A" = 3.6, "B" = 4.1, "C" = 7.3, "D" = 8.8)
)

# Plot results:
ape::plot.phylo(
  x = ape::read.tree(text = "((A:1,B:2):2,(C:2,D:3):4);"),
  main = paste("Sum of squares =", scp$sum_of_squares)
)
ape::tiplabels(text = scp$tip_values)
ape::nodelabels(text = round(x = scp$ancestral_state_estimates, digits = 2))

# Estimate squared-change parsimony for four tips with single values and a single polytomy:
scp <- estimate_squared_change_ancestors(
  tree = ape::read.tree(text = "(A,(B,C,D));"),
  tip_values = tip_values <- c("A" = 3.6, "B" = 4.1, "C" = 7.3, "D" = 8.8)
)

# Plot results:
ape::plot.phylo(
  x = ape::read.tree(text = "(A,(B,C,D));"),
  main = paste("Sum of squares =", scp$sum_of_squares)
)
ape::tiplabels(text = scp$tip_values)
ape::nodelabels(text = round(x = scp$ancestral_state_estimates, digits = 2))
```

`find_costmatrix_minimum_span`

Finds a minimum spanning tree of a costmatrix

Description

Given a costmatrix, returns a shortest tree connecting every state.

Usage

```
find_costmatrix_minimum_span(costmatrix)
```

Arguments

`costmatrix`	An object of class `costMatrix`.

Details

The minimum parsimony length a phylogenetic hypothesis can have depends on both the costmatrix of transition costs and the states actually sampled. If the costmatrix rows and columns already represent sampled states (the assumption here) then this minimum length is reduced to a graph theory problem - the minimum spanning tree or, if a directed graph, the minimum weight spanning arboresence. (NB: if there are unsampled states then the `find_steiner_tree_of_stategraph` function should be used instead.) This function returns one such shortest tree (although others may exist). The sum of the weights of the edges or arcs returned is the minimum cost.

As the algorithms used are graph theory based the function operates by simply calling convert_costmatrix_to_stategraph and find_stategraph_minimum_span. In practice, if the costmatrix represents a graph (transition costs are all symmetric) then Kruskal's algorithm is applied (Kruskal 1956). If costs are asymmetric, however, then the graph representation is a directed graph (or digraph) and so a version of Edmonds' algorithm is applied (Edmonds 1967).

Note that Dollo characters represent a special case solution as although a penalty weight is applied to the edges intended to only ever be traversed once this weight should not be used when calculating tree lengths. The function catches this and returns the edges with the weight that would actually be counted for a minimum weight spanning tree.

Value

A `data.frame` object describing a minimum spanning tree or minimum weight arboresence as a series of edges or arcs.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Edmonds, J., 1967. Optimum branchings. *Journal of Research of the National Bureau of Standards Section B*, **71B**, 233-240.

Kruskal, J. B., 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, **7**, 48-50.

See Also

find_shortest_costmatrix_path

Examples

```
# Make a four-state ordered character costmatrix:
ordered_costmatrix <- make_costmatrix(
  min_state = "0",
  max_state = "3",
  character_type = "ordered"
)

# Find length of shortest spanning tree of costmatrix:
find_costmatrix_minimum_span(costmatrix = ordered_costmatrix)

# Make a four-state unordered character costmatrix:
unordered_costmatrix <- make_costmatrix(
  min_state = "0",
  max_state = "3",
  character_type = "unordered"
)

# Find length of shortest spanning tree of costmatrix:
find_costmatrix_minimum_span(costmatrix = unordered_costmatrix)

# Make a four-state irreversible character costmatrix:
irreversible_costmatrix <- make_costmatrix(
  min_state = "0",
  max_state = "3",
  character_type = "irreversible"
)

# Find length of shortest spanning tree of costmatrix:
find_costmatrix_minimum_span(costmatrix = irreversible_costmatrix)

# Make a four-state Dollo character costmatrix:
dollo_costmatrix <- make_costmatrix(
  min_state = "0",
  max_state = "3",
  character_type = "dollo"
)

# Find length of shortest spanning tree of costmatrix:
find_costmatrix_minimum_span(costmatrix = dollo_costmatrix)
```

`find_descendant_edges` *Gets descendant edges of an internal node*

Description

Returns all descendant edges of an internal node for a phylo object.

Usage

```
find_descendant_edges(n, tree)
```

Arguments

n	An integer corresponding to the internal node for which the descendant edges are sought.
tree	A tree as a phylo object.

Details

Returns a vector of integers corresponding to row numbers in $edge or cells in $edge.length of the descendant edges of the internal node supplied.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Create simple four-taxon tree:
tree <- ape::read.tree(text = "(A,(B,(C,D)));")

# Plot tree:
plot(tree)

# Show nodelabels:
nodelabels()

# Show edgelabels (note that edges 5 and 6
# are descendants of node 7):
edgelabels()

# Use find_descendant_edges to show that edges
# 5 and 6 are descendants of node 7:
find_descendant_edges(n = 7, tree = tree)
```

---

find_linked_edges *Find linked edges for a tree*

Description

Given a tree finds edges that are linked to each other.

Usage

```
find_linked_edges(tree)
```

Arguments

`tree` A tree (phylo object).

Details

Finds all edges that link (share a node) with each edge of a tree.

This is intended as an internal function, but may be of use to someone else.

Value

Returns a matrix where links are scored 1 and everything else 0. The diagonal is left as zero.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Create a simple four-taxon tree:
tree <- ape::read.tree(text = "(A,(B,(C,D)));")

# Find linked (1) edges matrix for tree:
find_linked_edges(tree)
```

---

find_minimum_spanning_edges

Get edges of minimum spanning tree

Description

Returns edges of a minimum spanning tree given a distance matrix.

Usage

```
find_minimum_spanning_edges(distance_matrix)
```

Arguments

`distance_matrix` A square matrix of distances between objects.

Details

This function is a wrapper for mst in the ape package, but returns a vector of edges rather than a square matrix of links.

Value

A vector of named edges (X->Y) with their distances. The sum of this vector is the length of the minimum spanning tree.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Create a simple square matrix of distances:
distance_matrix <- matrix(c(0, 1, 2, 3, 1, 0, 1, 2, 2, 1, 0, 1, 3, 2, 1, 0),
  nrow = 4,
  dimnames = list(LETTERS[1:4], LETTERS[1:4])
)

# Show matrix to confirm that the off diagonal has the shortest
# distances:
distance_matrix

# Use find_minimum_spanning_edges to get the edges for the minimum spanning
# tree:
find_minimum_spanning_edges(distance_matrix)

# Use sum of find_minimum_spanning_edges to get the length of the minimum
# spanning tree:
sum(find_minimum_spanning_edges(distance_matrix))
```

---

find_mrca *Find ancestor*

Description

Finds the last common ancestor (node) of a set of two or more descendant tips.

Usage

```
find_mrca(descendant_names, tree)
```

Arguments

descendant_names	A vector of mode character representing the tip names for which an ancestor is sought.
tree	The tree as a phylo object.

Details

Intended for use as an internal function for trim_matrix, but potentially of more general use.

Value

`ancestor_node` The ancestral node number.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Create a simple four-taxon tree:
tree <- ape::read.tree(text = "(A,(B,(C,D)));")

# Plot the tree:
ape::plot.phylo(tree)

# Add nodelabels and show that the most recent common
# ancestor of B, C, and D is node 6:
ape::nodelabels()

# Use find_mrca to show that the most recent common
# ancestor of B, C, and D is node 6:
find_mrca(
  descendant_names = c("B", "C", "D"),
  tree = tree
)
```

---

find_shortest_costmatrix_path

Finds the shortest path between two states in a costmatrix

Description

Given a start and end state, returns the shortest path through a costmatrix.

Usage

```
find_shortest_costmatrix_path(costmatrix, start, end)
```

Arguments

`costmatrix`	An object of class `costMatrix`.
`start`	The start state for the requested path.
`end`	The end state of the requested path.

Details

A common problem in graph theory is identifying the shortest path to take between two vertices of a connected graph. A costmatrix also describes a graph, with transition costs representing edge weights that can be asymmetric (e.g., going from 0 to 1 can have a different weight than going from 1 to 0). Finding the shortest path between states - i.e., the path that minimises total weight (cost) - from a costmatrix has two main applications in Claddis: 1. to check a costmatrix is internally consistent (no cost is misidentified due to a "cheaper" route being available - solving a problem identified in Maddison and Maddison 2003), and 2. to identify the minimum cost a character could have on a tree (an essential aspect of various homoplasy metrics, see Hoyal Cuthill et al. 2010).

The function returns a vector describing (one) shortest (i.e., lowest cost) path between `start` and `end`. If the direct path is shortest this will be simply `start` and `end`, but if an indirect route is cheaper then other node(s) will appear between these values.

In operation the function is inspired by Dijkstra's algorithm (Dijkstra 1959) but differs in some aspects to deal with the special case of a cladistic-style costmatrix. Essentially multiple paths are considered with the algorithm continuing until either the destination node (`end`) is reached or the accumulated cost (path length) exceeds the direct cost (meaning the path cannot be more optimal, lower cost, than the direct one).

Note: that because infinite costs are allowed in costmatrices to convey that a particular transition is not allowed these are always likely to be replaced (meanng the path does become possible) unless they apply to entire rows or columns of a costmatrix.

Note: negative costs are not allowed in costmatrices as they will confound the halting criteria of the algorithm. (They also do not make logical sense in a costmatrix anyway!)

Note: if multiple equally optimal solutions are possible, the function will only return one of them. I.e., just because a solution is not presented it cannot be assumed it is suboptimal.

Value

A vector of states describing (one) of the optimal path(s) in the order `start` to `end`.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Dijkstra, E. W., 1959. A note on two problems in connexion with graphs. *Numerische Mathematik*, **1**, 269–271.

Hoyal Cuthill, J. F., Braddy, S. J. and Donoghue, P. C. J., 2010. A formula for maximum possible steps in multistate characters: isolating matrix parameter effects on measures of evolutionary convergence. *Cladistics*, **26**, 98-102.

Maddison, D. R. and Maddison, W. P., 2003. *MacClade 4: Analysis of phylogeny and character evolution*. Version 4.06. Sinauer Associates, Sunderland, Massachusetts.

See Also

convert_adjacency_matrix_to_costmatrix, make_costmatrix

Examples

```
# Make a four-state Dollo costmatrix:
costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 3,
  character_type = "dollo",
  dollo_penalty = 100
)

# Find the shortest path from state 0 to state 3:
find_shortest_costmatrix_path(
  costmatrix = costmatrix,
  start = "0",
  end = "3"
)

# Show that this is directional by asking for the reverse path:
find_shortest_costmatrix_path(
  costmatrix = costmatrix,
  start = "3",
  end = "0"
)
```

---

find_stategraph_minimum_span

Finds a minimum spanning tree of a stategraph

Description

Given a stategraph, returns a shortest tree connecting every state.

Usage

```
find_stategraph_minimum_span(stategraph)
```

Arguments

stategraph	An object of class stateGraph.

Details

The minimum parsimony length a phylogenetic hypothesis can have depends on both the stategraph of transition costs and the states actually sampled. If the stategraph vertices already represent sampled states (the assumption here) then this minimum length is reduced to a graph theory problem - the minimum spanning tree or, if a directed graph, the minimum weight spanning arboresence. (NB: if there are unsampled states then the find_steiner_tree_of_stategraph function should be used instead.) This function returns one such shortest tree (although others may exist). The sum of the weights of the edges or arcs returned is the minimum cost.

As the algorithms used are graph theory based the function operates by simply calling convert_costmatrix_to_stategraph and find_stategraph_minimum_span. In practice, if the costmatrix represents a graph (transition costs are all symmetric) then Kruskal's algorithm is applied (Kruskal 1956). If costs are asymmetric, however, then the graph representation is a directed graph (or digraph) and so a version of Edmonds' algorithm is applied (Edmonds 1967).

Note that Dollo characters represent a special case solution as although a penalty weight is applied to the edges intended to only ever be traversed once this weight should not be used when calculating tree lengths. The function catches this and returns the edges with the weight that would actually be counted for a minimum weight spanning tree.

Value

A `data.frame` object describing a minimum spanning tree or minimum weight arboresence as a series of edges or arcs.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Edmonds, J., 1967. Optimum branchings. *Journal of Research of the National Bureau of Standards Section B*, **71B**, 233-240.

Kruskal, J. B., 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, **7**, 48-50.

See Also

find_shortest_costmatrix_path

Examples

```
# Make a four-state ordered character stategraph:
ordered_costmatrix <- make_costmatrix(
  min_state = "0",
  max_state = "3",
  character_type = "ordered"
)
ordered_stategraph <- convert_costmatrix_to_stategraph(costmatrix = ordered_costmatrix)

# Find length of shortest spanning tree of stategraph:
find_stategraph_minimum_span(stategraph = ordered_stategraph)

# Make a four-state unordered character stategraph:
unordered_costmatrix <- make_costmatrix(
  min_state = "0",
  max_state = "3",
  character_type = "unordered"
)
unordered_stategraph <- convert_costmatrix_to_stategraph(costmatrix = unordered_costmatrix)
```

```
# Find length of shortest spanning tree of stategraph:
find_stategraph_minimum_span(stategraph = unordered_stategraph)

# Make a four-state irreversible character stategraph:
irreversible_costmatrix <- make_costmatrix(
  min_state = "0",
  max_state = "3",
  character_type = "irreversible"
)
irreversible_stategraph <- convert_costmatrix_to_stategraph(costmatrix = irreversible_costmatrix)

# Find length of shortest spanning tree of stategraph:
find_stategraph_minimum_span(stategraph = irreversible_stategraph)

# Make a four-state Dollo character stategraph:
dollo_costmatrix <- make_costmatrix(
  min_state = "0",
  max_state = "3",
  character_type = "dollo"
)
dollo_stategraph <- convert_costmatrix_to_stategraph(costmatrix = dollo_costmatrix)

# Find length of shortest spanning tree of stategraph:
find_stategraph_minimum_span(stategraph = dollo_stategraph)
```

`find_time_bin_midpoints`

Find time bin midpoints

Description

Find the midpoint values for each bin from a timeBins object

Usage

```
find_time_bin_midpoints(time_bins)
```

Arguments

`time_bins`	A timeBins object.

Details

Frequently the midpoints of a series of time bins (defined by a beginning and ending) will be required, for example, when plotting binned data as a time series. Although the calculation involved is trivial (i.e., start date + end date / 2) this is a sufficiently common operation it is made into a formal function here.

Note that this function is designed to work specifically with objects of class "timeBins" - a format specific to Claddis that looks something like this:

```
                fad  lad
Cenomanian      99.6 93.5
Turonian        93.5 89.3
Coniacian       89.3 85.8
Santonian       85.8 83.5
Campanian       83.5 70.6
Maastrichtian 70.6 65.5
```

I.e., a matrix with two columns (fad = first appearance date and lad = last appearance date) with rows corresponding to named time bins and indiviual values ages in millions of years ago (Ma). The object should also have class `timeBins` (see example below for hot to generate such an object). Note also that the convention here is to have time bins be ordered from oldest to youngest.

Value

A vector of time bin midpoint values.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Create a time bins object:
time_bins <- matrix(
  data = c(99.6, 93.5, 93.5, 89.3, 89.3, 85.8, 85.8, 83.5, 83.5, 70.6, 70.6, 65.5),
  ncol = 2,
  byrow = TRUE,
  dimnames = list(
    c("Cenomanian", "Turonian", "Coniacian", "Santonian", "Campanian", "Maastrichtian"),
    c("fad", "lad")
  )
)

# Set class as timeBins:
class(time_bins) <- "timeBins"

# Return midpoints for each time bin in sequence:
find_time_bin_midpoints(time_bins = time_bins)
```

`find_unique_trees` *Finds only the unique topologies amongst a set*

Description

Given a set of trees with the same tip labels, returns just the unique topologies present.

Usage

```
find_unique_trees(trees)
```

Arguments

`trees`	An object of class `multiPhylo`.

Details

Where labelled topologies are generated randomly or modified by (e.g.) removing a tip, it may be useful to isolate just those that are truly unique. The `ape` package already has a function for this (unique.multiPhylo), but it can be slow when the number of trees is large. This function is thus intended as a faster version.

The function works by breaking down a tree into its' component bipartitions and treating the combination of these as the definition of the tree. It thus escapes problems due to the principle of free rotation. Specifically, these two trees are actually identical:

```
A  B  C   D  E
 \/    \   \/
  \     \  /
   \     \/
    \    /
     \  /
      \/

B  A  D  E   C
 \/    \/   /
  \     \  /
   \     \/
    \    /
     \  /
      \/
```

This becomes clearer if we decompose them into their bipartitions:

AB, DE, CDE, ABCDE

These correspond to the descendants of each internal node (branching point) and the last one is actually ignored (the root node) as it will be present in any tree.

Value

An object of class "multiPhylo".

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

See Also

unique.multiPhylo

Examples

```
# Make a set of three identical trees (differing only in "rotation" of nodes):
trees <- ape::read.tree(text = c(
  "((A,B),(C,(D,E)));",
  "((C,(D,E)),(A,B));",
  "((B,A),(C,(E,D)));")
)

# Show that there is only one unique tree:
find_unique_trees(trees = trees)
```

---

fix_costmatrix *Fixes a costmatrix that has inconsistent costs*

Description

Given a costmatrix where transition costs are not self-consistent finds and returns a costmatrix that does.

Usage

```
fix_costmatrix(costmatrix, message = TRUE)
```

Arguments

costmatrix	A costMatrix object.
message	A logical indicating whether messages should be output (defaults to TRUE).

Details

A user may wish to consider a complicated set of transition costs between states when modelling discrete character evolution. This can be achieved with a custom costmatrix in Claddis (and elsewhere). However, some caution is urged when using such matrices to ensure that these costs are *self-consistent* (Maddison and Maddison 2003). More specifically, no direct state-to-state transition cost should be greater than is possible with an indirect path via one or more intermediate states.

This function offers a solution through an algorithm that will iteratively alter a costmatrix until all direct transition costs are self-consistent. It does so by finding the shortest state-to-state path for all possible transitions using the find_shortest_costmatrix_path function. Because the first solution may itself be inconsistent (as it relied on costs that have since updated) the algorithm is repeated until an equilibrium is reached. (This scenario is unlikely in most real world cases, but may be possible with very large matrices representing many states so was implemented here for safety.)

Note: infinite costs are allowed in costmatrices but unless they fill entire rows or columns (excluding the diagonal) they will not be self-consistent as there will always be a cheaper indirect cost.

Note: that both PAUP* (Swofford 2003) TNT (Goloboff et al. 2008; Goloboff and Catalano, 2016) offerthe same correction using the triangle inequality.

Note: other issues with a costmatrix may arise that are better revealed by using the check_costMatrix function, which returns informative error messages (with fix suggestions) where issues are found.

Value

A costMatrix object with self-consistent transition costs.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Goloboff, P. A. and Catalano, S. A., 2016. TNT version 1.5, including a full implementation of phylogenetic morphometrics. *Cladistics*, **32**, 221-238.

Goloboff, P., Farris, J. and Nixon, K., 2008. TNT, a free program for phylogenetic analysis. *Cladistics*, **24**, 774-786.

Maddison, D. R. and Maddison, W. P., 2003. *MacClade 4: Analysis of phylogeny and character evolution*. Version 4.06. Sinauer Associates, Sunderland, Massachusetts.

Swofford, D. L., 2003. *PAUP*. Phylogenetic Analysis Using Parsimony (*and Other Methods). Version 4*. Sinauer Associates, Sunderland, Massachusetts.

Examples

```
# Build a custom costmatrix with non-self consistent path lengths:
costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 2,
  character_type = "irreversible"
)
costmatrix$costmatrix[1:9] <- c(0, 2, 4, 1, 0, 3, 5, 3, 0)
```

```
costmatrix$symmetry <- "Asymmetric"
costmatrix$type <- "custom"

# Fix costmatrix:
fixed_costmatrix <- fix_costmatrix(costmatrix = costmatrix)

# Compare transition costs:
costmatrix$costmatrix
fixed_costmatrix$costmatrix
```

`fix_root_time` *Fixes root.time after taxa have been pruned from a tree*

Description

Fixes root.time after taxa have been pruned from a tree using ape::drop.tip

Usage

```
fix_root_time(original_tree, pruned_tree)
```

Arguments

`original_tree`	A tree in phylo format.
`pruned_tree`	A tree in phylo format that represents a pruned version of `original_tree`.

Details

(NB: This function is designed to only cope with trees containing at least three tips.)

When removing taxa from a time-scaled tree using drop.tip in ape `$root.time` is left unchanged. This can cause downstream problems if not fixed and that is what this function does.

Note that `fix_root_time` in the `paleotree` package performs the same function, but is not called here to reduce the number of libraries on which `Claddis` is dependent. Interested users should also refer to the `dropPaleoTip` function in `paleotree`.

Value

Returns a tree (phylo object) with a fixed `$root.time`.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

See Also

drop_time_tip

Examples

```
# Create a simple four-taxon tree with branch lengths:
tree <- ape::read.tree(text = "(A:1,(B:1,(C:1,D:1):1):1);")

# Set root age as 20 Ma:
tree$root.time <- 20

# Now prune taxon A:
pruned_tree <- ape::drop.tip(phy = tree, tip = "A")

# Show that drop.tip has not updated the tree's root time:
pruned_tree$root.time

# Use the function to fix the root time:
pruned_tree <- fix_root_time(original_tree = tree, pruned_tree = pruned_tree)

# Show that the root time is now fixed (19 Ma):
pruned_tree$root.time
```

---

gauthier_1986 *Character-taxon matrix from Gauthier 1986*

Description

The character-taxon matrix from Gauthier (1986).

Format

A character-taxon matrix in the format imported by read_nexus_matrix.

References

Gauthier, J. A., 1986. Saurischian monophyly and the origin of birds. In Padian, K. (ed.) *The Origin of Birds and the Evolution of Flight*. Towne and Bacon, San Francisco, CA, United States, 1-55.

---

is.cladisticMatrix *Cladistic matrix class*

Description

Functions to deal with the cladistic matrix class.

Usage

```
is.cladisticMatrix(x)
```

Arguments

x	A cladisticMatrix object.

Details

Claddis uses various classes to define specific types of data, here the use of cladistic data (the main input of the package) is assigned the class "cladisticMatrix".

`is.cladisticMatrix` checks whether an object is or is not a valid cladisticMatrix object.

Value

`is.cladisticMatrix` returns either TRUE or FALSE.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Check that this is a valid cladisticMatrix object (will succeed as format and
# class are correct):
is.cladisticMatrix(x = day_2016)
```

---

is.costMatrix *Costmatrix class*

Description

Functions to deal with the costmatrix class.

Usage

```
is.costMatrix(x)
```

Arguments

x	An object of class `costMatrix`.

Details

Claddis uses various classes to define specific types of data, here the use of costmatrices (to specify the parsimony costs of transitions between character states) are assigned the class "costMatrix".

`is.costMatrix` checks whether an object is or is not a valid costMatrix object.

Value

`is.costMatrix` returns either TRUE or FALSE.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Make an unordered costmatrix:
costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 2,
  character_type = "unordered"
)

# Check that this is a valid costMatrix object:
is.costMatrix(x = costmatrix)
```

---

is.stateGraph *Stategraph class*

Description

Functions to deal with the stategraph class.

Usage

```
is.stateGraph(x)
```

Arguments

x	An object of class stateGraph.

Details

Claddis uses various classes to define specific types of data, here the use of a character stategraph (to specify the parsimony costs of transitions between character states) are assigned the class "stateGraph".

is.stateGraph checks whether an object is or is not a valid stateGraph object.

Value

is.stateGraph returns either TRUE or FALSE.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Make an unordered costmatrix:
costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 2,
  character_type = "unordered"
)

# Convert costmatrix to stategraph:
stategraph <- convert_costmatrix_to_stategraph(costmatrix = costmatrix)

# Check that this is a valid costMatrix object:
is.stateGraph(x = stategraph)
```

`is.taxonGroups` *Taxon groups class*

Description

Functions to deal with the taxon groups class.

Usage

```
is.taxonGroups(x)
```

Arguments

`x`	An object of class `taxonGroups`.

Details

Claddis uses various classes to define specific types of data, here the use of taxon groups (to delineate different groups of taxa, e.g., clades, time bins, geographic regions etc.) ae assigned the class "taxonGroups".

`is.taxonGroups` checks whether an object is or is not a valid taxonGroups object.

Value

`is.taxonGroups` returns either TRUE or FALSE.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Create a taxon groups object:
taxon_groups <- list(
  Group_A = c("Species_1", "Species_2", "Species_3"),
  Group_B = c("Species_3", "Species_4"),
  Group_C = c("Species_5", "Species_6", "Species_7", "Species_8")
)

# Check that this is a valid taxonGroups object (will fail as class is not set):
is.taxonGroups(x = taxon_groups)

# Set class as taxonGroups:
class(taxon_groups) <- "taxonGroups"

# Check that this is a valid taxonGroups object (will succeed as format and
# class are correct):
is.taxonGroups(x = taxon_groups)
```

`is.timeBins` *Time bins class*

Description

Functions to deal with the time bins class.

Usage

```
is.timeBins(x)
```

Arguments

x	A timeBins object.

Details

Claddis uses various classes to define specific types of data, here the use of time bins (to bin any temporal data) ae assigned the class "timeBins" and should look something like this:

```
                fad  lad
Cenomanian     99.6 93.5
Turonian       93.5 89.3
Coniacian      89.3 85.8
Santonian      85.8 83.5
Campanian      83.5 70.6
Maastrichtian  70.6 65.5
```

I.e., a matrix with two columns (fad = first appearance date and lad = last appearance date) with rows corresponding to named time bins and individual values ages in millions of years ago (Ma). The object should also have class `timeBins` (see example below for how to generate a valid object). Note also that the convention in Claddis is to have time bins be ordered from oldest to youngest.

`is.timeBins` checks whether an object is or is not a valid timeBins object.

Value

`is.timeBins` returns either TRUE or FALSE.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Create a time bins object:
time_bins <- matrix(
  data = c(99.6, 93.5, 93.5, 89.3, 89.3, 85.8, 85.8, 83.5, 83.5, 70.6, 70.6, 65.5),
  ncol = 2,
  byrow = TRUE,
  dimnames = list(
    c("Cenomanian", "Turonian", "Coniacian", "Santonian", "Campanian", "Maastrichtian"),
    c("fad", "lad")
  )
)

# Check that this is a valid timeBins object (will fail as class is not set):
is.timeBins(x = time_bins)

# Set class as timeBins:
class(time_bins) <- "timeBins"

# Check that this is a valid timeBins object (will succeed as format and
# class are correct):
is.timeBins(x = time_bins)
```

`is_graph_connected` *Is a graph connected?*

Description

Is a graph represented by an adjacenecy matrix connected?

Usage

```
is_graph_connected(adjacency_matrix)
```

Arguments

`adjacency_matrix`
: An adjacency matrix where the diagonal is zeroes and the off-diagonal either ones (if the two vertices are directly connected) or zeroes (if not directly connected).

Details

Any undirected graph can be represented as an adjacency matrix and the properties of this matrix can be used to determine whether or not the graph is connected (i.e., a path between any two vertices exists) or not.

For example, the following graph:

```
6---4---5
    |   |\
    |   | 1
    |   |/
    3---2
```

Has the adjacency matrix:

```
    -------------------------
    | 1 | 2 | 3 | 4 | 5 | 6 |
-----------------------------
| 1 | 0 | 1 | 0 | 0 | 1 | 0 |
-----------------------------
| 2 | 1 | 0 | 1 | 0 | 1 | 0 |
-----------------------------
| 3 | 0 | 1 | 0 | 1 | 0 | 0 |
-----------------------------
| 4 | 0 | 0 | 1 | 0 | 1 | 1 |
-----------------------------
| 5 | 1 | 1 | 0 | 1 | 0 | 0 |
-----------------------------
| 6 | 0 | 0 | 0 | 1 | 0 | 0 |
-----------------------------
```

This functions run through the following checks in order to confirm the connectivity of the graph:

1. As the graph has more than one vertex then further checks are required (a single vertex graph is considered connected).
2. As no vertice has degree zero (no links) then the graph *may* be connected (further checks are required).
3. As there are more than two vertices the graph may be disconnected (further checks are required).
4. As no missing paths are found (that would separate the graph into one or more disconnected subgraphs) then the graph must be connected.

This ordering means more complex queries are not triggered unless simpler tests do not provide a definitive answer.

Value

A logical (TRUE or FALSE).

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Create the connected graph matrix:
x <- matrix(
  data = c(
    0, 1, 0, 0, 1, 0,
    1, 0, 1, 0, 1, 0,
    0, 1, 0, 1, 0, 0,
    0, 0, 1, 0, 1, 1,
    1, 1, 0, 1, 0, 0,
    0, 0, 0, 1, 0, 0
  ),
  ncol = 6,
  byrow = TRUE
)

# Check graph is connected:
is_graph_connected(adjacency_matrix = x)
```

---

locate_bracket_positions

Locates matching positions for sets of brackets in a text string

---

Description

Given a text string will return the positions of each matching pair of opening and closing brackets.

Usage

```
locate_bracket_positions(input_string, bracket_type = "()")
```

Arguments

input_string	An input string containing matching brackets, such as a Newick string or character state tree.
bracket_type	The type of bracket to use. Must be one of parentheses `()`, curly braces `{}`, or square brackets `[]`.

Details

This function is designed to deal with Newick strings and character state trees - ways of encoding information using nested parentheses. Although it is intended for internal use it seems sufficiently general to share as part of the package.

The function works by traversing the string from left to right and noting the position of each opening parenthesis and then storing the corresponding position for its' matching closing parenthesis.

It currently only works for a single string, but coud be built into a for loop or apply function if multiple strings are desired.

Value

A two-column matrix indicating opening and closing bracket positions within `input_string`.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

See Also

convert_state_tree_to_adjacency_matrix

Examples

```
# Locate the positions of a set of parentheses in a character state tree:
locate_bracket_positions(
  input_string = "(((5)4)3,(2)1)0",
  bracket_type = "()"
)
```

---

make_costmatrix *Make a costmatrix for a given set of states*

Description

Given a set of discrete states and a character type will make the approriate costmatrix for them.

Usage

```
make_costmatrix(
  min_state = 0,
  max_state,
  character_type,
  include_polymorphisms = FALSE,
  include_uncertainties = FALSE,
  polymorphism_costs = "additive",
  polymorphism_geometry = "simplex",
```

```
  polymorphism_distance = "euclidean",
  state_ages,
  dollo_penalty = 999,
  message = TRUE
)
```

Arguments

`min_state` The minimum character state (defaults to `0`).

`max_state` The maximum character state. Must be `1` or greater.

`character_type` The type of character desired. Must be one of: `"ordered"`, `"unordered"`, `"dollo"`, `"irreversible"`, or `"stratigraphy"`.

`include_polymorphisms` Logical indicating whether or not to include polymorphic state combinations (defaults to `FALSE`).

`include_uncertainties` Logical indicating whether or not to include uncertainty state combinations (defaults to `FALSE`).

`polymorphism_costs` Only used if `include_polymorphisms = TRUE`. See add_polymorphisms_to_costmatrix.

`polymorphism_geometry` Only used if `include_polymorphisms = TRUE`. See add_polymorphisms_to_costmatrix.

`polymorphism_distance` Only used if `include_polymorphisms = TRUE`. See add_polymorphisms_to_costmatrix.

`state_ages` A vector of ages assigned to each state. Only triggered if `character_type = "stratigraphy"`.

`dollo_penalty` The size of the cost penalty for the acquisition of a Dollo character (defaults to `999`). Only triggered if `character_type = "dollo"`. Note: this should always be a positive real value greater than one, and never infinity (`Inf`), as at least one acquisition is expected.

`message` Logical indicating whether (`TRUE`, the default) or not (`FALSE`) to provide messages about the process of the function.

Details

Costmatrices encode the parsimony cost (typically the number of evolutionary "steps") for each possible state-to-state transition. They can be used to estimate total lengths and make estimates (often also called reconstructions) for ancestral states at branching points on a given topology (tree). This function automates the generation of some common costmatrix types for use elsewhere in Claddis and hence is intended primarily as an internal function. However, as costmatrices are fundamental to various key analyses their operation is detailed extensively below.

Costmatrix basics

Although their usage is rare as explicit statements (see Hooker 2014 for an example), almost any character type (e.g., ordered, unordered) can be expressed as a costmatrix. These are always square, with rows and columns representing the same set of states that will usually represent, but are not restricted to, the range of values observed in the data. Individual values in a costmatrix represent

the cost of a particular transition, with the row value being the "from" state and the column value being the "to" state. By convention, the cost of going from any character to itself is zero and hence so is the diagonal of the matrix. An example costmatrix for a three state unordered character would thus look like this:

```
    -------------
    | 0 | 1 | 2 |
-----------------
| 0 | 0 | 1 | 1 |
-----------------
| 1 | 1 | 0 | 1 |
-----------------
| 2 | 1 | 1 | 0 |
-----------------
```

Hence going from state 2 (third row) to state 1 (second column) has a cost of 1. Indeed, as this is an unordered character, *any* off-diagonal value has the same cost of 1. By contrast an ordered matrix might look like this:

```
    -------------
    | 0 | 1 | 2 |
-----------------
| 0 | 0 | 1 | 2 |
-----------------
| 1 | 1 | 0 | 1 |
-----------------
| 2 | 2 | 1 | 0 |
-----------------
```

Now going from state 0 (first row) to state 2 (third column) costs two, as by implication this transition must pass through the intermediate state (1).

So far these examples are symmetric - i.e., you can imagine the diagonal as a line of reflection, or, alternatively, that going from state X to state Y *always* costs the same as going from state Y to state X. However, asymmetric costmatrices are also possible and there are multiple such character-types that may be relevant here. (Specifically, Dollo, irreversible, also known as Camin-Sokal, and stratigraphic.)

Character types

This function will generate a costmatrix for every possible transition (between `min_state` and `max_state`, inclusive) for a requested `character_type`, each of which is detailed further below.

Ordered - `character_type = "ordered"`

An ordered character (really a linear ordered character), also known as a Wagner character (Wagner 1961) and formalised by Farris (1970), is one where the order of states matters. State-to-state transitions must occur through a linear series. For example, for a three state character the states must be in order of transition, 0 <-> 1 <-> 2. Additionally, all transitions are symmetric. 0 -> 1 = 1 -> 0.

Unordered - `character_type = "unordered"`

An unordered character, also known as a Fitch character (Fitch 1971), is one where the order of states does not matter, and that state-to-state transitions are all direct and symmetric. For example, for a five-state character the transition 0 -> 4 is direct and costs the same as 3 -> 1, or any other transition, excepting those from any state to itself.

Dollo - `character_type = "dollo"`

Under a Dollo assumption the acquisition of a *derived* character state is considered to be sufficiently complex (biologically difficult) that in all probability it only occurred once. Even if lost later in evolution, it is considered that it cannot be reacquired. An example of this is the frequent loss of teeth in dinosaurs (Brocklehurst and Field 2021). Teeth were never reacquired in this group, with tooth-like serrations forming in many birds instead. Modelling such characters with a costmatrix is challenging (indeed, it cannot truly been done with *just* a costmatrix - see below). The way it is typically done (Swofford and Olsen 1990) is to form the costmatrix like this:

```
    -------------------
    |  0  |  1  |  2  |
-----------------------
| 0 |  0  |  1D |  2D |
-----------------------
| 1 |  1  |  0  |  1D |
-----------------------
| 2 |  2  |  1  |  0  |
-----------------------
```

Where `D` is some arbitrary large number, set here using the `dollo_penalty` option. Note that `D` should still be finite as the purpose is to weight acquisitions such that they are sufficiently expensive that any most parsimonious reconstruction will favour only one transition, but not make them so expensive that they are not favoured at all. Furthermore, in this example there are two derived states (1 and 2) and hence logically the weighting should be similar to an ordered character. Specifically, that there should be a single acquisition of state 1 (from state 0) and that this should precede a single acquistion of state 2 (from state 1). Most of the time, though, a Dollo character will be binary (e.g., see map_dollo_changes).

Importantly, and as stated above, a Dollo costmatrix, unlike the ordered and unordered costmatrices, cannot be used without further analytical restrictions. Specifically, because it assumes an asymmetric acquisition of *derived* states the root or "primitive" value must be 0. (It would not be logical to apply such a costmatrix where the root state is 1 or 2.) Thus use of a Dollo character requires the additional assumption that the primitive (root) state is known *a priori*. As always, it is up to the user to know that this is a valid assumption for their data, and to set a value for the `dollo_penalty` accordingly.

Note, that another way to conceive of a Dollo character is that the assumption is being applied that any homoplasy is in the form of reversals only and hence can be though of as making the opposite assumption to an irreversible character.

Irreversible (Camin-Sokal) - `character_type = "irreversible"`

Sometimes confused with a Dollo character, irreversible or Camin-Sokal (Camin and Sokal 1965) characters, do not allow for any reversals (returns to some prior state) once the derived state has been acquired. An irreversible costmatrix might look like this:

```
    -------------------
```

```
    |  0  |  1  |  2  |
-----------------------
| 0 |  0  |  1  |  2  |
-----------------------
| 1 | Inf |  0  |  1  |
-----------------------
| 2 | Inf | Inf |  0  |
-----------------------
```

The upper triangle (representing gains) is the same as for an ordered character, but the lower triangle (representing losses, or reversals) will always be comprised of infinite costs (`Inf`), precluding their possibility in a most parsimonious reconstruction. Thus, in practice, any number of gains is allowed, but losses never are.

Like a Dollo character, this approach assumes the direction of evolution is known *a priori* and that 0 is always the primitive, or root, state. Although here, and unlike a Dollo character, the root state is forced by the costmatrix alone.

Stratigraphic - `character_type = "stratigraphy"`

Stratigraphic characters, where states represent geologic time units in which an OTU is present, are also logically irreversible - as a younger unit cannot exist before (be "ancestral" to) an older one. Thus here, too, the lower triangle of the costmatrix is always populated by infinities to preclude their possibility. However, this time the upper triangle requires additional information to populate its' values, i.e., the gap between states in units of time (typically millions of years).

This is best illustrated with an example. Let's assume we have three states and they represent three consecutive geologic stages: 0 (Santonian, 85.8-83.5 Ma), 1 (Campanian, 83.5-70.6 Ma), and 2 (Maastrichtian, 70.6-65.5 Ma). Now the cost of a transition between states should be expressed in some difference between the ages of each state. In this example we will use the midpoints as our point estimate: 0 (84.65 Ma), 1 (77.05 Ma), and 2 (68.05 Ma). These must also be supplied to the function using the `state_ages` option, i.e.: `state_ages = c(84.65, 77.05, 68.05)`. The resulting costmatrix would look like this:

```
    --------------------
    |  0  |  1  |   2  |
------------------------
| 0 |  0  | 7.6 | 16.6 |
------------------------
| 1 | Inf |  0  |   9  |
------------------------
| 2 | Inf | Inf |   0  |
------------------------
```

Note that the user needn't use the midpoint (a start or end date may be preferable). Indeeed, a simple unit count could be applied instead but this would then be identical to an irreversible character (except, perhaps, in the case of polymorphisms - see add_polymorphisms_to_costmatrix).

The use of stratigraphy as a character is probably not generally recommended, but is offered here for those that wish to explore stratocladistic methods that aim to incorporate stratigraphic information in phylogenetic inference and identify ancestor-descendant relationships (see Fisher 1994).

As with the other asymmetric characters it is an additional assumption that the root represent 0 (the oldest sampled value), but as with irreverisble characters, the costmatrix alone forces this assumption.

Other character types

Although no other character types are offered here, other forms of costmatrix can of course exist (e.g., see those in Hooker 2014). These can still be used elsewhere in Claddis, but users must either generate them themselves or have them be specified within a NEXUS file imported into Claddis. Note that general advice is to always have the diagonal be zero and the off-diagonal values be positive, otherwise downstream use will typically be confounded.

Character weights

As discusssed in Hoyal Cuthill and Lloyd (in prep), the costs in a costmatrix are more properly considered as ratios. Consider the following two costmatrices:

```
    -------------
    | 0 | 1 | 2 |
-----------------
| 0 | 0 | 1 | 2 |
-----------------
| 1 | 1 | 0 | 1 |
-----------------
| 2 | 2 | 1 | 0 |
-----------------

    -------------
    | 0 | 1 | 2 |
-----------------
| 0 | 0 | 2 | 4 |
-----------------
| 1 | 2 | 0 | 2 |
-----------------
| 2 | 4 | 2 | 0 |
-----------------
```

Although the specific costs differ the ratio *between* costs within each costmatrix is identical. I.e., the cost of a 0 to 1 transition is half the cost of a 0 to 2 transition in both costmatrices. Or, to put it another way, the second costmatrix can be derived from the first by multiplying every cost by the same "weight", here two. As such, these two costmatrices can be considered identical for analytical purposes, save for this weight term. So, for example, the ancestral state estimates would be identical for a given tree and set of tip values. And the tree length (including the minimum and maximum length) would be the same save for the weight modifier.

This observation is largely irrelevant to this function. However, it is noted here so the user understands why the costmatrx structure (outlined below) includes a weight term. Further, it is important to state why this information should be encoded "outside" the costmatrix rather than encodng "inside" the costmatrix by simply multiplying every cost by the desired character weight (e.g., as in the 2-4-2 example above). The reason is that this can lead to confounded results in some cases. For example, if weighting a character zero to exclude it from an analysis would result in a costmatrix where all off diagonal costs were zero (the same cost as no change). In practice this would mean

any ancestral state estimation would become equally likely. If encoding the weight "outside" the costmatrix this situation can be handled correctly, both within Claddis and if exporting data for use elsewhere.

Polymorphisms and uncertainties

Where a taxon (terminal) has two or more states (or a single state cannot be identified) it represents either a polymorphism or an uncertainty. For some kinds of analyses, such as calculating tree lengths or estimating ancesral states, there is no need to explicitly address uncertainties in a costmatrix setting (Swofford and Maddison 1992). However, the same cannot be said of polymorphisms (Maddison and Maddison 1987). In any case, this function offers the chance to include either or both polymorphic or uncertain states into the costmatrix by calling add_polymorphisms_to_costmatrix or add_uncertainties_to_costmatrix, as appropriate. Interested users should consult the helpfiles for those functions for more details.

Relation to Q-matrices

Q-matrices (e.g., see Swofford and Olsen 1990) are used in likelihood and Bayesian approaches and also deal with state-to-state transitions, but are fundamentally distinct from costmatrices in multiple ways. For example, Q-matrices encode transition rates, not costs, and their values are typically parameters that are estimated from the data not prior statements about the data. However, both can be rendered as state graph or Markov chain diagrams. Put simply though, they are not directly equivalent or interchangable.

Value

An object of class `costMatrix` containing the following elements:

- `size` The number of columns (or rows) in the costmatrix.
- `n_states` The number of (single) states in the costmatrix.
- `single_states` The labels of the (single) states in the costmatrix.
- `type` The type of costmatrix. One of: `"ordered"`, `"unordered"`, `"dollo"`, `"irreversible"`, `"stratigraphy"`, or `"custom"`.
- `costmatrix` The costmatrix itself.
- `symmetry` The symmetry of the costmatrix. NB: This refers to the single states part only.
- `includes_polymorphisms` A logical indicating whether or not polymorphic states are included in the costmatrix.
- `polymorphism_costs` The means by which costs were assigned to polymorphisms. Must be one of: `"additive"`, `"geometric"`, `"maddison"`, or `"stratigraphic"`.
- `polymorphism_geometry` The geometry by which costs were assigned to polymorphisms. Must be one of: `"hypercube"`, `"hypersphere"`, or `"simplex"`.
- `polymorphism_distance` The distance metric by which costs were assigned to polymorphisms. Must be one of: `"euclidean"`, `"great_circle"`, or `"manhattan"`.
- `includes_uncertainties` A logical indicating whether or not uncertain states are included in the costmatrix.
- `pruned` A logical indicating whether or not the costmatrix represents a pruned version.
- `dollo_penalty` A numeric value indicating the penalty used for a Dollo character.
- `base_age` A numeric value indicating the base (oldest) age used for a stratigraphic character.
- `weight` A numeric value indicating the character weight.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Brocklehurst, N. and Field, D. J., 2021. Macroevolutionary dynamics of dentition in Mesozoic birds reveal no long-term selection towards tooth loss. *iScience*, **24**, 102243.

Camin, J. H. and Sokal, R. R., 1965. A method for deducing branching sequences in phylogeny. *Evolution*, **19**, 311-26.

Farris, J. S., 1970. Methods for computing Wagner trees. *Systematic Zoology*, **19**, 83-92.

Fisher, D. C., 1994. Stratocladistics: morphological and temporal patterns and their relation to phylogenetic process. In L. Grande and O. Rieppel (eds.), *Interpreting the Hierarchy of Nature*. Academic Press, San Diego. pp133–171.

Fitch, W. M., 1971. Towards defining the course of evolution: minimum change for a specified tree topology. *Systematic Zoology*, **20**, 406-416.

Hooker, J. J., 2014. New postcranial bones of the extinct mammalian family Nyctitheriidae (Paleogene, UK): primitive euarchontans with scansorial locomotion. *Palaeontologia Electronica*, **17.3.47A**, 1-82.

Maddison, W. P. and Maddison, D. R., 1987. MacClade 2.1, computer program and manual. Cambridge, Massachusetts.

Swofford, D. L. and Maddison, W. P., 1992. Parsimony, character-state reconstructions, and evolutionary inferences. In R. L. Mayden (ed.), *Systematics, Historical Ecology, and North American Freshwater Fishes*. Stanford University Press, Stanford. pp187-223.

Swofford, D. L. and Olsen, G. J., 1990. Phylogeny reconstruction. In D. M. Hillis and C. Moritz (eds.), *Molecular Systematics*. Sinauer Associates, Sunderland. pp411-501.

Wagner, W. H., 1961. Problems in the classification of ferns. *Recent Advances in Botany*, **1**, 841-844.

See Also

permute_all_polymorphisms

Examples

```
# Make an unordered three-state costmatrix:
make_costmatrix(
  min_state = 0,
  max_state = 2,
  character_type = "unordered"
)

# Make an ordered three-state costmatrix:
make_costmatrix(
  min_state = 0,
  max_state = 2,
  character_type = "ordered"
)
```

```
# Make a three-state Dollo costmatrix:
make_costmatrix(
  min_state = 0,
  max_state = 2,
  character_type = "dollo",
  dollo_penalty = 100
)

# Make a three-state irreversible costmatrix:
make_costmatrix(
  min_state = 0,
  max_state = 2,
  character_type = "irreversible",
)

# Make a three-state stratigraphic costmatrix:
make_costmatrix(
  min_state = 0,
  max_state = 2,
  character_type = "stratigraphy",
  state_ages = c(52, 34, 12)
)
```

`make_labels` *Make unique text labels*

Description

Given a requisite number, generates that many unique text labels.

Usage

```
make_labels(N)
```

Arguments

N	The number of labels required,

Details

Where a list of unique text labels are required (i.e., where simple numbering will not suffice) it can be useful to have a simple function that generates the required amount.

In practice, this is simple in R when N is 26 or less as the `LETTERS` object can be used for this purpose. For example, to get ten unique labels:

`LETTERS[1:10]`

This function works in a similar way but will add a second, third etc. letter where the value of N requires it.

Value

A character vector of N unique labels.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Make 40 unique text labels:
make_labels(N = 40)
```

---

map_dollo_changes *Stochastic Character Map For Dollo Character*

Description

Given a tree with binary tip states produces a stochastic Dollo character map.

Usage

```
map_dollo_changes(time_tree, tip_states)
```

Arguments

time_tree	A tree in phylo format with positive branch lengths and a value for $root.time.
tip_states	A named vector of tip states (must be 0 or 1), where the names match tree$tip.label.

Details

The non-ideal solution from Tarver et al. (2018) to the problem of generating a stochastic character map for a Dollo character (i.e., a single gain of the derived state, 1) with any number of losses (1 -> 0).

The function operates as follows:

1) Establishes the least inclusive clade exhibiting the derived state (1). 2) Assumes a single gain occurred with equal probability along the branch subtending this clade. 3) Prunes the inclusive clade to generate a subtree with a strong root prior of the derived state (1). 4) Calls make.simmap from the phytools package to generate a stochastic character map using a model where only losses are possible. 5) Outputs both the stochastic character map (time spent in each state on each branch) and a matrix of state changes.

NB: As the map is stochastic the answer will be different each time the function is run and multiple replicates are strongly advised in order to characterise this uncertainty.

Value

`changes` A matrix of all changes (gains and losses).

`stochastic_character_map` The stochastic character map.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Tarver, J. E., Taylor, R. S., Puttick, M. N., Lloyd, G. T., Pett, W., Fromm, B., Schirrmeister, B. E., Pisani, D., Peterson, K. J. and Donoghue, P. C. J., 2018. Well-annotated microRNAomes do not evidence pervasive miRNA loss. *Genome Biology and Evolution*, **6**, 1457-1470.

Examples

```
# Build example ten-tip tree:
time_tree <- ape::read.tree(text = paste0("(A:1,(B:1,((C:1,(D:1,(E:1,F:1):1):1):1,",
  "((G:1,H:1):1,(I:1,J:1):1):1):1):1);"))

# Arbitrarily add a root.time value of 100 Ma:
time_tree$root.time <- 100

# Build example tip state values:
tip_states <- c(A = 0, B = 0, C = 1, D = 1, E = 0, F = 1, G = 1, H = 1, I = 0, J = 1)

# Run map_dollo_changes on data and store output:
out <- map_dollo_changes(time_tree, tip_states)

# View matrix of changes:
out$changes

# View stochastic character map (time spent in each state on each branch):
out$stochastic_character_map
```

`match_tree_edges` *Edge matching function*

Description

Given two trees where one is a pruned version of the other gives matching edges and nodes of pruned tree to original tree.

Usage

```
match_tree_edges(original_tree, pruned_tree)
```

Arguments

original_tree	A tree in phylo format.
pruned_tree	A tree in phylo format that represents a pruned version of original_tree.

Details

Finds matching edge(s) and node(s) for a pruned tree in the original tree from which it was created. This is intended as an internal function, but may be of use to someone.

Value

matching_edges	A list of the matching edges.
matching_nodes	A matrix of matching node numbers.
removed_edges	A vector of the removed edges.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Create a random 10-taxon tree:
original_tree <- ape::rtree(n = 10)

# Remove three leaves:
pruned_tree <- ape::drop.tip(phy = original_tree, tip = c("t1", "t3", "t8"))

# Find matching edges:
X <- match_tree_edges(original_tree, pruned_tree)

# Show matching edges:
X$matching_edges

# Show matching nodes:
X$matching_nodes

# Show removed edges:
X$removed_edges
```

---

michaux_1989 *Character-taxon matrix from Michaux 1989*

Description

The character-taxon matrix from Michaux (1989).

Format

A character-taxon matrix in the format imported by read_nexus_matrix.

References

Michaux, B., 1989. Cladograms can reconstruct phylogenies: an example from the fossil record. *Alcheringa*, **13**, 21-36.

---

ordinate_cladistic_matrix *Principal Coordinates on a Cladistic Matrix*

Description

Performs Principal Coordinates Analysis (PCoA) on a cladistic matrix.

Usage

```
ordinate_cladistic_matrix(
  cladistic_matrix,
  distance_metric = "mord",
  ged_type = "wills",
  distance_transformation = "arcsine_sqrt",
  distance_polymorphism_behaviour = "min_difference",
  distance_uncertainty_behaviour = "min_difference",
  distance_inapplicable_behaviour = "missing",
  character_dependencies = NULL,
  alpha = 0.5,
  correction = "cailliez",
  time_tree = NULL,
  estimate_all_nodes = FALSE,
  estimate_tip_values = FALSE,
  inapplicables_as_missing = FALSE,
  ancestral_polymorphism_behaviour = "equalp",
  ancestral_uncertainty_behaviour = "equalp",
  threshold = 0.01,
  all_missing_allowed = FALSE
)
```

Arguments

`cladistic_matrix` A character-taxon matrix in the format imported by read_nexus_matrix.

`distance_metric` See calculate_morphological_distances.

`ged_type` See calculate_morphological_distances.

`distance_transformation`
: See calculate_morphological_distances.

`distance_polymorphism_behaviour`
: See calculate_morphological_distances.

`distance_uncertainty_behaviour`
: See calculate_morphological_distances.

`distance_inapplicable_behaviour`
: See calculate_morphological_distances.

`character_dependencies`
: See calculate_morphological_distances.

`alpha`
: See calculate_morphological_distances.

`correction`
: The negative eigenvalue correction to use (one of "lingoes", "none", or "cailliez" - the default). See pcoa for more details.

`time_tree`
: If a phylmorphospace is desired then a tree with root age and branch-lengths must be included.

`estimate_all_nodes`
: See estimate_ancestral_states.

`estimate_tip_values`
: See estimate_ancestral_states.

`inapplicables_as_missing`
: See estimate_ancestral_states.

`ancestral_polymorphism_behaviour`
: See estimate_ancestral_states.

`ancestral_uncertainty_behaviour`
: See estimate_ancestral_states.

`threshold`
: See estimate_ancestral_states.

`all_missing_allowed`
: See estimate_ancestral_states.

Details

Takes a cladistic matrix in the format imported by read_nexus_matrix and performs Principal Coordinates (Gower 1966) analysis on it.

This function is effectively a wrapper for the pipeline:

estimate_ancestral_states -> calculate_morphological_distances -> pcoa

With the first part being optional (if wanting a phylomorphospace) and the latter coming from the ape package (the user is referred there for some of the options, e.g., using the Caillez 1983 approach to avoiding negative eigenvalues). (See Lloyd 2016 for more on disparity pipelines.)

If providing a tree and inferring ancestral states then options to also infer missing or uncertain tips and whether to infer values for all characters at all internal nodes are provided by the estimate_ancestral_states part.

Other options within the function concern the distance metric to use and the transformation to be used if selecting a propotional distance (see calculate_morphological_distances).

IMPORTANT: The function can remove taxa (or if including a tree, nodes as well) if they lead to an incomplete distance matrix (see trim_matrix for more details).

Value

- `time_tree` The tree (if supplied). Note this may be pruned from the input tree by trim_matrix.
- `distance_matrix` The distance matrix. Note this may be pruned by trim_matrix and thus not include all taxa.
- `removed_taxa` A vector of taxa and/or nodes removed by trim_matrix. Returns NULL if none were removed.
- `note` See pcoa.
- `values` See pcoa.
- `vectors` See pcoa. Note: this will be the same as `vectors.cor` from the pcoa output if a correction was applied.
- `trace` See pcoa. Note: this will be the same as `trace.cor` from the pcoa output if a correction was applied.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Cailliez, F., 1983. The analytical solution of the additive constant problem. *Psychometrika*, **48**, 305-308.

Gower, J. C., 1966. Some distance properties of latent root and vector methods used in multivariate analysis. *Biometrika*, **53**, 325-338.

See Also

assign_taxa_to_bins, plot_chronophylomorphospace, plot_morphospace_stack, plot_morphospace, plot_multi_morphospace

Examples

```
# Run on Michaux (1989) data set with default settings:
x <- ordinate_cladistic_matrix(cladistic_matrix = michaux_1989)

# Show entire output:
x

# Generate a (made up) tree:
time_tree <- ape::rtree(n = length(x = rownames(x = michaux_1989$matrix_1$matrix)))

# Add taxon names to it:
time_tree$tip.label <- rownames(x = michaux_1989$matrix_1$matrix)

# Set root time by making youngest taxon extant:
time_tree$root.time <- max(diag(x = ape::vcv(phy = time_tree)))

# Run with tree:
```

```
y <- ordinate_cladistic_matrix(cladistic_matrix = michaux_1989, time_tree = time_tree)

# Show new output:
y
```

---

partition_time_bins *Time bin partitioner*

Description

Generates all possible contiguous partitions of N time bins.

Usage

```
partition_time_bins(n_time_bins, partition_sizes_to_include = "all")
```

Arguments

`n_time_bins` The number of time bins.

`partition_sizes_to_include` Either "all" (the default) or a vector of requested partition sizes.

Details

This function is designed for use with the test_rates function and generates all possible contiguous partitions of N time bins. This allows use of an information criterion like AIC to pick a "best" partition, weighing fit and partition number simultaneously.

You can also ask for only partitions of a specific number using the `partition_sizes_to_include` option. For example, `partition_sizes_to_include = c(1, 2, 3)` will only return partitions of 1, 2, or 3 sets of elements.

Value

Returns a list of lists of vectors ready for use in test_rates.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Get all partitions for four time bins:
partition_time_bins(n_time_bins = 4)

# Get all partitions for five time bins of size 2:
partition_time_bins(n_time_bins = 5, partition_sizes_to_include = 2)
```

permute_all_polymorphisms

Permute all possible polymorphisms for a given set of states

Description

Given a set of discrete states, will permute all possible polymorphic combinations of those states.

Usage

```
permute_all_polymorphisms(single_states)
```

Arguments

`single_states`	A vector of single states (e.g., 0, 1, 2 etc.).

Details

This function solves a simple phylogenetic combinatorics problem - what are all the possible outcomes for a character to be in given polymorphisms (of any size) are allowed?

For example, for three states (0, 1, 2) there are four possible polymorphisms: 0&1, 0&2, 1&2 and 0&1&2.

If the user is instead only interested in the size of this state space, this is simply given by 2^N - N - 1, where N is the number of single states. Thus, the first several outcomes are:

N states	N possible outcomes
2	1
3	4
4	11
5	26
6	57
7	120
8	247
9	502
10	1,013
11	2,036
12	4,083
13	8,178
14	16,369

Note that this function is really designed for internal use, but may have value to some users and so is available "visibly" here.

Value

A vector of all possible polymorphic states.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

See Also

make_costmatrix and permute_all_uncertainties

Examples

```
# Get all possible states for the character 0, 1, and 2:
permute_all_polymorphisms(single_states = 0:2)
```

---

permute_all_treeshape_labellings *Label treeshapes*

Description

Given a treeshape and set of labels, permutes all possible labelled phylogenetic trees.

Usage

```
permute_all_treeshape_labellings(treeshapes, labels)
```

Arguments

treeshapes	A vector of treeshape(s) in the same format as permute_treeshapes.
labels	A character vector of tip labels to use for labelling.

Details

A treeshape is an unlabelled phylogenetic tree and as such can be labelled to produce a phylogenetic tree. This function takes a treeshape and a set of labels and generates (permutes) all possible labellings, i.e., all phylogenetic trees which a treeshape represents.

Note that the star tree always allows only a single labelling, whereas any more resolved treeshape will have multiple labellings.

Here treeshapes are encoded in the same pseudo-Newick format as the permute_treeshapes function, e.g.:

(((3),1),(1,(2)));

(Where each pair of parentheses represents an internal node, each number the number of tips, each comma separates sets of tips, and the semicolon denotes the root clade.)

Value

A list of the same length as `treeshapes` composed of character vectors of labelled phylogenetic trees in the Newick format (Felsenstein 2004).

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Felsenstein, J., 2004. *Inferring Phylogenies*. Sinauer Associates, Inc., Sunderland.

Examples

```
# Label some six-tip treeshapes with the letters A-F:
permute_all_treeshape_labellings(
  treeshapes = c(
    "(6);",
    "((3),(3));",
    "(1,(1,(1,(1,(2)))));"
  ),
  labels = LETTERS[1:6]
)
```

---

permute_all_uncertainties

Permute all possible uncertainties for a given set of states

Description

Given a set of discrete states, will permute all possible uncertainity combinations of those states.

Usage

```
permute_all_uncertainties(single_states)
```

Arguments

`single_states`	A vector of single states (e.g., 0, 1, 2 etc.).

Details

This function solves a simple phylogenetic combinatorics problem - what are all the possible outcomes for a character to be in given uncertainties are allowed?

For example, for three states (0, 1, 2) there are four possible uncertainties: 0/1, 0/2, 1/2 and 0/1/2.

If the user is instead only interested in the size of this state space, this is simply given by 2^N - N - 1, where N is the number of single states. Thus, the first several outcomes are:

```
----------------------------------
| N states | N possible outcomes |
----------------------------------
|     2    |          1          |
|     3    |          4          |
|     4    |          11         |
|     5    |          26         |
|     6    |          57         |
|     7    |          120        |
|     8    |          247        |
|     9    |          502        |
|    10    |          1,013      |
|    11    |          2,036      |
|    12    |          4,083      |
|    13    |          8,178      |
|    14    |          16,369     |
----------------------------------
```

Note that this function is really designed for internal use, but may have value to some users and so is available "visibly" here.

Value

A vector of all possible uncertainty states.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

See Also

make_costmatrix and permute_all_polymorphisms

Examples

```
# Get all possible states for the character 0, 1, and 2:
permute_all_uncertainties(single_states = 0:2)
```

---

permute_connected_graphs

Permute all connected graphs

Description

Given a vertex count, permutes all connected graphs.

Usage

```
permute_connected_graphs(n_vertices)
```

Arguments

`n_vertices`	The number of vertices to connect.

Details

For the two vertex case there is only a single connected graph:

```
A---B
```

(The labels A and B here simply indicate the two vertices and are not a true labelling.)

If we add a third vertex, there are two connected graphs:

```
A---B
 \ /
  C
```

And:

```
A---B---C
```

This function permutes all such connected graphs for a given vertex count.

Note that the output is in the form of a matrix of edges. For the three vertex case above these would be:

```
     [,1] [,2]
[1,] "A"  "B"
[2,] "A"  "C"
[3,] "B"  "C"
```

And:

```
     [,1] [,2]
[1,] "A"  "B"
[2,] "B"  "C"
```

Again, it is important to note that the labels A, B, and C here are purely "dummy" labels and should not be considered a graph labelling. To use the second graph as an example there are multiple labellings of this graph:

```
A---B---C
```

And:

```
B---A---C
```

And:

```
A---C---B
```

However, these are all isomorphisms of the same unlabelled graph. Only the unique graphs themselves are returned here.

Value

A list of graphs (matrices of dummy labelled edges).

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Generate all connected graphs of four vertices:
permute_connected_graphs(n_vertices = 4)
```

---

permute_costmatrices *Permute costmatrices*

Description

Given vectors of states and costs, permutes all possible costmatrices.

Usage

```
permute_costmatrices(states = c("0", "1"), costs = c(1:3), symmetry = "both")
```

Arguments

states	A vector of character states, e.g., "0", "1", "2".
costs	A vector of numeric costs, e.g., 1, 2, Inf.
symmetry	Must be one of "symmetric", "asymmetric" or "both".

Details

Costmatrices define the cost of each state-to-state transition, but they are restricted in what these costs can be (see check_costMatrix). Nevertheless, strictly speaking there are infinite possible costmatrices - even where costs are restricted to integer values (as TNT does; Goloboff et al. 2008; Goloboff and Catalano 2016), i.e., "stepmatrices" (Swofford and Maddison 1992). Thus this function operates on a finite system by requiring the user to specify a restricted set of states and individual cost values, with the function permuting every possible combination of finite costs. Note that not *every* permutation will be returned as not all of these will be valid costmatrices (see check_costMatrix and fix_costmatrix). Others will not be returned because their cost *ratio* can be considered redundant. For example, for a binary character (states "0", and "1") the following two costmatrices would be mutually redundant as the ratio of their costs is identical:

```
  A B
A 0 1
B 2 0
```

```
  A B
A 0 2
B 4 0
```

(If the user does want to consider these kinds of alternatives then a better solution is to simply weight the first matrix by two, or any other value, in any downstream analys(es).)

For the function to work costs must be unique positive values. This includes infinity (`Inf` in R). Infinite costs can be used to denote a particular transition is impossible and allows defining (e.g.) irreversible characters, or those that force a particular root value.

Value

A list of unique costmatrices containing every possible combination of costs.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Goloboff, P. A. and Catalano, S. A., 2016. TNT version 1.5, including a full implementation of phylogenetic morphometrics/ *Cladistics*, **32**. 221-238

Goloboff, P., Farris, J. and Nixon, K., 2008. TNT, a free program for phylogenetic analysis. *Cladistics*, **24**, 774-786.

Swofford, D. L. and Maddison, W. P., 1992. Parsimony, character-state reconstructions, and evolutionary inferences. *In* R. L. Mayden (ed.) Systematics, Historical Ecology, and North American Freshwater Fishes. Stanford University Press, Stanford, p187-223.

Examples

```
# Permute all the ways to assign the costs 1 and 2 for a three state
# character:
permute_costmatrices(
  states = c("0", "1", "2"),
  costs = c(1, 2),
  symmetry = "both"
)
```

`permute_graph_splits` *Permute all ways to split a graph*

Description

Given a graph represented by an adjacency matrix, permutes all ways this graph could be split apart.

Usage

```
permute_graph_splits(adjacency_matrix)
```

Arguments

`adjacency_matrix`
: A labelled adjacency matrix where the diagonal is zeroes and the off-diagonal either ones (if the two vertices are directly connected) or zeroes (if not directly connected). Labels must match across row names and column names,

Details

This function takes a connected graph and considers all the ways it *could* be split by removing every possible combination of edges (inclusive of none and all).

Value

A vector of splits where connected vert(ices) are listed with "+" joining them and disconnected vert(ices) are separated by "|".

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Create a connected graph matrix:
adjacency_matrix <- matrix(
  data = c(
    0, 1, 0, 0, 1, 0,
    1, 0, 1, 0, 1, 0,
    0, 1, 0, 1, 0, 0,
    0, 0, 1, 0, 1, 1,
    1, 1, 0, 1, 0, 0,
    0, 0, 0, 1, 0, 0
  ),
  ncol = 6,
  byrow = TRUE,
  dimnames = list(
    LETTERS[1:6],
    LETTERS[1:6]
  )
)

# Check graph is connectedPermute all ways to split graph (remove edges):
permute_graph_splits(adjacency_matrix = adjacency_matrix)
```

---

permute_restricted_compositions

Permute all ways to place n items into m bins

Description

Given a positive integer, n, and a number of bins (m), permutes all possible compositions.

Usage

```
permute_restricted_compositions(n, m_labels, allow_zero = FALSE)
```

Arguments

n	A positive integer.
m_labels	A character vector of labels for m.
allow_zero	A logical indicating whether or not each bin should (TRUE) or should not (FALSE) be allowed to be zero.

Details

Every way that an integer (n) can be divided up into m bins can be permuted using a restricted version of the mathematical concept of compositions. In practice this function is designed to distribute the states for n tips across m states (e.g., with permute_tipstates), but many other uses are conceivable and hence this is included here as a general function.

This algorithm reuses code from the `multicool` (Curran et al. 2021) and `partitions` (Hankin 2006) packages.

The number of restricted compositions is given by the k-dimensional extension of triangular numbers (Baumann 2019):

- If `allow_zero = TRUE`, the binomial coefficient, n choose k, where n = `n + m - 1` and k = `m`.
- If `allow_zero = FALSE`, the binomial coefficient, n choose k, where n = `n - 1` and k = `m`.

Value

A matrix where each row is a unique restricted composition of n and each column is a labelled bin.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Baumann, M. H., 2019. Die k-dimensionale Champagnerpyramide. Mathematische Semesterberichte, 66, 89-100.

Curran, J., Williams, A., Kelleher, J. and Barber, D., 2021. multicool: Permutations of Multisets in Cool-Lex Order. R package version 0.1-12. https://CRAN.R-project.org/package=multicool.

Hankin, R. K. S., 2006. Additive integer partitions in R. *Journal of Statistical Software, Code Snippets*, **16**, 1.

Examples

```
# Permute all the ways eight can be assigned to four bins (A, C, G, T),
# with each bin assigned at least one:
permute_restricted_compositions(
  n = 8,
  m_labels = c("A", "C", "G", "T"),
  allow_zero = FALSE
)
```

`permute_tipstates` *Permute all tip states on a tree*

Description

Given a phylogenetic tree and a set of states, permutes all possible tip state combinations.

Usage

```
permute_tipstates(tree, states, all_states_present = TRUE)
```

Arguments

`tree` A phylogenetic tree in ape format.

`states` A vector of character states.

`all_states_present`
A logical indicating whether or not all states should appear at least once. (Defaults to `TRUE`.)

Details

When calculating g_max or otherwise determining the limits for a given character type it can be useful to generate (permute) all possible combinations of tip states for a given tree. For example, let's imagine we have a binary character (states 0 and 1) and four tips (labelled A-D). We could simply assign each possible state to each possible label to get the following sixteen permutations:

#	A	B	C	D
1	0	0	0	0
2	1	0	0	0
3	0	1	0	0
4	0	0	1	0
5	0	0	0	1
6	1	1	0	0
7	1	0	1	0
8	1	0	0	1
9	0	1	1	0
10	0	1	0	1
11	0	0	1	1
12	1	1	1	0
13	1	1	0	1
14	1	0	1	1
15	0	1	1	1
16	1	1	1	1

We can know there are sixteen permutations before we even check as logically for any N states we simply multiply T times (for T tips). I.e., the answer is given by N^T. As here N is 2 and T is 4 this is 2^4 = 2 x 2 x 2 x 2 = 16.

Technically this achieves our goal - we have generated all possible tip states, but there are at least two reasons this approach is suboptimal. Firstly, permutations 1 and 16 are invariant (all tip states are the same) and so are unlikely to be of interest. Or, to generalise this, we might not want to consider permutations where not all possible states are sampled - i.e., we may wish to stipulate that every state appear at least once. Secondly, this approach does not consider the structure of the phylogenetic tree of interest. It might not be immediately obvious why this matters, so let's consider another example, Sticking with our binary character and four tips let's consider the tree:

```
A   B   C   D
```

```
 \   \   \ /
  \   \   /
   \   \ /
    \   /
     \ /
```

And the following permutations (from above):

#	A	B	C	D
4	0	0	1	0
5	0	0	0	1
7	1	0	1	0
8	1	0	0	1
9	0	1	1	0
10	0	1	0	1
12	1	1	1	0
13	1	1	0	1

Here are the eight permutations where C and D are assigned different states. However, in practical terms there is no need to generate half of these permutations as the order CD/DC doesn't matter (due to the principal of free rotation). In other words, the "identity" (i.e., the label of a tip) is not as important as the structure of the tree.

Thus, there are two ways in which the simple ordered permutation method inefficiently generates permutations that are redundant with respect to a given tree. This is important as scaling the simple assignment of states to labels (N^T) can rapidly lead to very large numbers of permutations:

	Number of tips				
N States	5	10	20	50	100
2	32	1,024	1,048,576	1.13 x 10^15	1.27 x 10^30
3	243	59,049	3,486,784,401	7.18 x 10^23	5.15 x 10^47
4	1,024	1,048,576	1.10 x 10^12	1.27 x 10^30	1.61 x 10^60
5	3,125	9,765,625	9.54 x 10^13	8.88 x 10^34	7.89 x 10^69

Thus for many empirically realistic examples the computational expense is too great and a more complex, less redundant, permutation algorithm is desirable that directly considers both tree structure and the desire for full variance sampling.

As this is not a simple algorithm let's begin by consider a simple and limiting case - the star tree:

```
A   B   C   D
|   |   |   |
|   |   |   |
-------------
```

Here there is no "normal" tree structure, but in practical terms this means there is no reason to consider *which* label a tip state is assigned to. If we again consider our binary character from above this means by pruning invariant and redundant partitions we get just three permutations:

#	A	B	C	D
2	1	0	0	0
6	1	1	0	0
12	1	1	1	0

Put simply, these are the cases where there is one, two, or three of state 1, and three, two or one of state 0. Thus we have dramatically reduced the number of permutations, here by over 80

	Number of tips				
N States	5	10	20	50	100
2	4	9	19	49	99

However, things become a little more complex when we consider a third state. It also makes more sense to start tabulating these permutations by the number of times a state appears instead of arbitrarily assigning states to a particular label. So our binary example would look like this:

0	1
3	1
2	2
1	3

If we now add a third possible state (2) we can permute all the possible scenarios (excluding those where a state doesn't appear at all) as:

0	1	2
2	1	1
1	2	1
1	1	2

Thus for this example there are still only three options. This is because stipulating that every state appears at least once means there is only one "free" assignment that can be allocated as either state

0, state 1 or state 2 - accounting for the three permutations. In other words, the assignments here had to be 2, 1, and 1, the only distinction is the "order" of these assignments. Let's consider a fifth tip but stick with a star tree. Now the assignments can be 3, 1, and 1 or 2, 2, and 1:

0	1	2
3	1	1
1	3	1
1	1	3
2	2	1
2	1	2
1	2	2

Thus by increasing the number of tips by one in this instance we have also doubled the permutations. Note, though, that we do not need an additional column to represent this, just additional rows. Adding a sixth tip we get:

0	1	2
4	1	1
1	4	1
1	1	4
3	2	1
3	1	2
2	3	1
1	3	2
1	2	3
2	1	3
2	2	2

Now we have three possible assignments (4-1-1, 3-2-1 and 2-2-2), but find the ways to permute these are no longer equal (3, 6 and 1, respectively). This makes it harder to derive an equation that will allow us to scale the problem to any value of N or T. We can repeat our table from above though to get a general sense of scale. Remember that this is for the star tree with the additional stipulation that every state appears at least once:

	Number of tips				
N States	5	10	20	50	100
2	4	9	19	49	99
3	6	36	171	1,176	4,851
4	4	84	969	18,424	156,849
5	1	126	3,876	211,876	3,764,376

We see, then, that this scales much more reasonably than our initial approach, although this is the "best case" scenario of no tree structure. We can also relax things a little by allowing states to not always appear:

	Number of tips				
N States	5	10	20	50	100
2	6	11	21	51	101
3	21	66	231	1,326	5,151
4	56	286	1,771	23,426	176,851
5	126	1,001	10,626	316,251	4,598,126

These are larger, but not intolerably so.

In practice things can become more complex when the full variety of possible treeshapes are considered and in fact the present algorithm is not without the possibility of redundancies. Specifically, because trees can contain repeating "motifs" that share a common ancestor fewer permutations would still cover the full range of possibilities. For example, consider the tree:

```
A   B   C   D   E   F
 \   \ /     \   \ /
  \   /       \   /
   \ /         \ /
    \           /
     \         /
      \       /
       \     /
        \   /
         \ /
```

Here the motifs:

```
A   B   C          D   E   F
 \   \ /            \   \ /
  \   /     and      \   /
   \ /                \ /
```

Are the same as such so would be the permutations:

A	B	C	D	E	F
0	1	1	1	0	0
1	0	0	0	1	1

Currently the function does not account for these, nor is a closed form equation known that would generate the true smallest number of unique permtations.

Instead, it simply permutes each "berry" (a node where all descendants are terminals, an extension of the notion pf phylogenetic "cherries") as though it is the star tree, and permuting all other tips as simple labellings.

Value

A matrix where each row is a unique suite of states and each column a lebelled tip of the input tree.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Permute four tip states (A, C, G, T) on a six-taxon tree:
permute_tipstates(
  tree = ape::read.tree(text = "((A,B),(C,(D,(E,F))));"),
  states = c("A", "C", "G", "T"),
  all_states_present = TRUE
)

# Permute three tip states (0, 1, 2) on a ten-taxon star tree:
permute_tipstates(
  tree = ape::read.tree(text = "(A,B,C,D,E,F,G,H,I);"),
  states = c("0", "1", "2"),
  all_states_present = TRUE
)
```

---

permute_treeshapes *Permute all treeshapes of N tips*

Description

Given a number of tips, permutes all rooted unlabelled multifurcating trees (i.e., treeshapes).

Usage

```
permute_treeshapes(n_tips, sort_by_resolution = TRUE)
```

Arguments

`n_tips` The number of tips required. Note that it may be very slow or not run at all if this value is too large.

`sort_by_resolution` Whether or not to sort the output by number of internal nodes (from 1 to N - 1). Defaults to `TRUE`.

Details

A treeshape is essentially an unlabelled phylogenetic tree. Like other phylogenetic trees it has a root and tips, but (as you might expect) because it is unlabelled the tips have no specific identity. Thus the only information it contains is its' "shape" - the number of internal nodes and their descendants. This function permutes all *unique* treeshapes and allows for multifurcations.

Note that unique means it excludes alternative rotations of individual branch points. For example, the trees ((2),1); and (1,(2)); are identical in information content and this function would only permute one of them.

The algorithm used here is loosely based on the partitions approach suggested in Felsenstein (2004), although to the best of my knowledge nobody else has formally created an algorithm to do this. (Felsenstein also lays out the expected number of such treeshapes for each value of N.)

Here treeshapes are encoded and output in a pseudo-Newick style format where labels are replaced with the number of tips, e.g.:

(((3),1),(1,(2)));

Thus each pair of parentheses represents an internal node, each number the number of tips, each comma separates sets of tips, and the semicolon denotes the root clade.

Value

If `sort_by_resolution = TRUE` then returns a list of length N - 1, where each element is a character vector of treeshapes in Newick-style number format with that many internal nodes. I.e., the first value will always be the star tree. If `sort_by_resolution = FALSE` then will just be a character vector of treeshapes in Newick-style number format.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Felsenstein, J., 2004. *Inferring Phylogenies*. Sinauer Associates, Inc., Sunderland.

Examples

```
# Permute all treeshapes of six tips:
permute_treeshapes(n_tips = 6)
```

---

plot_changes_on_tree *Plots character changes on branches*

Description

Plots character changes in boxes on branches.

Usage

```
plot_changes_on_tree(character_changes, time_tree, label_size = 0.5)
```

Arguments

character_changes	A matrix of character changes.
time_tree	Tree on which character changes occur.
label_size	The size of the text for the barnch labels. Default is 0.5.

Details

Takes the character_changes output from test_rates and plots it on the tree used to generate it.

Value

A plot of character changes on a tree.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Set random seed:
set.seed(17)

# Get first MPT for the Michaux data set:
time_tree <- ape::read.tree(text = paste0("(Ancilla:31.6,(Turrancilla:102.7,",
  "(Ancillista:1,Amalda:63.5):1):1);"))

# Set root time for tree:
time_tree$root.time <- 103.7

# Generate two equal length time bins:
time_bins <- matrix(data = c(seq(time_tree$root.time, 0, length.out = 3)[1:2],
 seq(time_tree$root.time, 0, length.out = 3)[2:3]), ncol = 2, dimnames = list(LETTERS[1:2],
  c("fad", "lad")))

# Set class as timeBins:
class(time_bins) <- "timeBins"

# Get discrete character rates (includes changes):
out <- test_rates(
  time_tree = time_tree,
  cladistic_matrix = michaux_1989,
  time_bins = time_bins,
  branch_partitions = list(list(1)),
  alpha = 0.01
)
```

```
# Plot character changes on the tree:
plot_changes_on_tree(
  character_changes = out$inferred_character_changes,
  time_tree = time_tree
)
```

---

plot_chronophylomorphospace

Chronophylomorphospace Plot

Description

Plots a three-dimensional chronophylomorphospace.

Usage

```
plot_chronophylomorphospace(
  pcoa_input,
  x_axis = 1,
  y_axis = 2,
  taxon_groups = NULL,
  plot_tips = TRUE,
  plot_nodes = TRUE,
  plot_taxon_names = TRUE,
  plot_edges = TRUE,
  shadow = TRUE,
  plot_group_legend = TRUE,
  group_legend_position = "top_right",
  palette = "viridis"
)
```

Arguments

pcoa_input	Principal coordinate data in the format output by ordinate_cladistic_matrix that includes a tree and ancestral states.
x_axis	Which ordination axis to plot as the x-axis.
y_axis	Which ordination axis to plot as the y-axis.
taxon_groups	An object of class taxonGroups.
plot_tips	Whether or not to plot the tip nodes (defaults to TRUE).
plot_nodes	Whether or not to plot the internal nodes (defaults to TRUE).
plot_taxon_names	Whether or not to show the taxon nodes (defaults to TRUE).
plot_edges	Whether or not to plot the branches (defaults to TRUE).
shadow	Whether or not to plot a shadow (2D plot) on the bottom face of the 3D plot (defaults to TRUE).

`plot_group_legend`
: Whether or not to add a legend to identify the groups. Only relevant if using `"taxon_groups"`.

`group_legend_position`
: Position to plot the group legend. Must be one of `bottom_left`, `bottom_right`, `top_left`, or `top_right` (the default).

`palette`
: The palette to use for plotting each element of taxon_groups. See palette.

Details

Creates a manually repositionable three-dimensional (two ordination axes plus time) plot of a phylomorphospace.

This function aims to mimic the data visualisation of Sakamoto and Ruta (2012; their Video S1).

Author(s)

Emma Sherratt <emma.sherratt@gmail.com> and Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Sakamoto, M. and Ruta, M. 2012. Convergence and divergence in the evolution of cat skulls: temporal and spatial patterns of morphological diversity. *PLoS ONE*, **7**, e39752.

See Also

assign_taxa_to_bins, plot_morphospace_stack, plot_morphospace, plot_multi_morphospace, ordinate_cladistic_matrix

Examples

```
## Not run:
# Require rgl library to use:
require(rgl)

# Make time-scaled first MPT for Day 2016 data set:
time_tree <- ape::read.tree(text = paste0("(Biarmosuchus_tener:0.5,",
  "(((Hipposaurus_boonstrai:3.5,(Bullacephalus_jacksoni:0.75,",
  "Pachydectes_elsi:0.75):0.75):0.75,(Lemurosaurus_pricei:7.166666667,",
  "(Lobalopex_mordax:4.333333333,((Lophorhinus_willodenensis:3.666666667,",
  "(Proburnetia_viatkensis:0.8333333333,(Lende_chiweta:2,",
  "(Paraburnetia_sneeubergensis:1,Burnetia_mirabilis:2):1):1.833333333)",
  ":0.8333333333):0.8333333333,(BP_1_7098:2.25,Niuksenitia_sukhonensis:",
  "1.25):1.25):0.8333333333):0.8333333333):3.083333333):1.95,",
  "(Ictidorhinus_martinsi:15.9,(RC_20:11.6,(Herpetoskylax_hopsoni:11.3,",
  "Lycaenodon_longiceps:0.3):0.3):0.3):0.3):0.3);"))

# Add root age to tree:
time_tree$root.time <- 269.5

# Prune incomplete taxa from tree:
time_tree <- ape::drop.tip(phy = time_tree, tip = c("Lycaenodon_longiceps",
```

```
  "Niuksenitia_sukhonensis"))

# Prune incomplete taxa from cladistic matrix:
cladistic_matrix <- prune_cladistic_matrix(cladistic_matrix = day_2016,
  taxa2prune = c("Lycaenodon_longiceps", "Niuksenitia_sukhonensis"))

# Perform a phylogenetic Principal Coordinates Analysis:
pcoa_input <- ordinate_cladistic_matrix(
  cladistic_matrix = cladistic_matrix,
  time_tree = time_tree
)

# Define some simple taxon groups for the data as a named list:
taxon_groups <- list(nonBurnetiamorpha = c("Biarmosuchus_tener",
  "Hipposaurus_boonstrai", "Bullacephalus_jacksoni", "Pachydectes_elsi",
  "Niuksenitia_sukhonensis", "Ictidorhinus_martinsi", "RC_20",
  "Herpetoskylax_hopsoni"),
  Burnetiamorpha = c("Lemurosaurus_pricei", "Lobalopex_mordax",
  "Lophorhinus_willodenensis", "Proburnetia_viatkensis", "Lende_chiweta",
  "Paraburnetia_sneeubergensis", "Burnetia_mirabilis", "BP_1_7098"))

# Set class as taxonGroups:
class(taxon_groups) <- "taxonGroups"

# Plot a chronophylomorphospace:
plot_chronophylomorphospace(
  pcoa_input = pcoa_input,
  taxon_groups = taxon_groups,
)

## End(Not run)
```

`plot_morphospace` *Plot Morphopace*

Description

Plots a morphospace using the output from ordinate_cladistic_matrix.

Usage

```
plot_morphospace(
  pcoa_input,
  x_axis = 1,
  y_axis = 2,
  z_axis = NULL,
  taxon_groups = NULL,
  plot_taxon_names = FALSE,
  plot_convex_hulls = FALSE,
```

```
  plot_internal_nodes = FALSE,
  plot_edges = TRUE,
  plot_root = TRUE,
  root_colour = "red",
  palette = "viridis",
  plot_group_legend = TRUE,
  group_legend_position = "top_right",
  plot_z_legend = TRUE,
  z_legend_position = "bottom_right",
  inform = TRUE,
  x_limits = NULL,
  y_limits = NULL,
  plot_size_landscape = FALSE,
  size_variable = NULL,
  n_x_tiles = 20,
  n_y_tiles = 20,
  landscape_colour = "green",
  landscape_transparency = 0.5,
  landscape_weight = 0.1,
  x_tile_apron = 1.2,
  y_tile_apron = 1.2
)
```

Arguments

`pcoa_input` The main input in the format output from ordinate_cladistic_matrix.

`x_axis` Which ordination axis to plot as the x-axis (defaults to 1).

`y_axis` Which ordination axis to plot as the y-axis (defaults to 2).

`z_axis` Which ordination axis to plot as the z-axis (defaults to NULL, i.e., is not plotted).

`taxon_groups` An object of class `taxonGroups`.

`plot_taxon_names` Logical indicating whether to plot the names of the taxa (defaults to FALSE).

`plot_convex_hulls` Logical indicating whether to plot convex hulls around any taxon_groups (if used).

`plot_internal_nodes` Logical indicating whether to plot the internal nodes of the tree (if included in `pcoa_input`) (defaults to FALSE).

`plot_edges` Logical indicating whether to plot the branches of the tree (if included in `pcoa_input`) (defaults to TRUE).

`plot_root` Logical indicating whether to plot the root separately (defaults to FALSE).

`root_colour` If plotting the root separately (previous option) sets the root colour.

`palette` The palette to use for plotting each element of taxon_groups. See palette.

`plot_group_legend` Logical indicating whether to plot a legend for taxon_groups. (Default is TRUE.)

group_legend_position
: Position to plot the group legend. Must be one of `bottom_left`, `bottom_right`, `top_left`, or `top_right` (the default).

plot_z_legend
: Logical indicating whether to plot a legend for the z-axis. (Default is TRUE.)

z_legend_position
: Position to plot the group legend. Must be one of `bottom_left`, `bottom_right` (the default), `top_left`, or `top_right`.

inform
: Logical indicating whether to inform the user of any taxon pruning. (Default is TRUE.)

x_limits
: Plot limits to use for x-axis. Only intended for use by plot_multi_morphospace.

y_limits
: Plot limits to use for y-axis. Only intended for use by plot_multi_morphospace.

plot_size_landscape
: Logical indicating whether or not to plot a body size "landscape". If `TRUE` then `size_variable` must be set.

size_variable
: A numeric vector with taxon names matching `pcoa_input`. Size can be measured anyway the user wishes (length, mass etc.).

n_x_tiles
: The number of horizontal "tiles" to plot a size landscape with.

n_y_tiles
: The number of vertical "tiles" to plot a size landscape with.

landscape_colour
: The colour of the size landscape. Must be one of `"blue"`, `"green"` or `"red"`.

landscape_transparency
: The transparency value to use for the size landscape. Must be on a zero to one scale. Default is `0.5`.

landscape_weight
: The "weight" to use for interpolating the colour of each size landscape tile. This is used to vary how much proximity to the tile is taken into account and can be any positive number (default is `0.1`).

x_tile_apron
: How far to extend the size landscape horizontally beyond the data points. Should be a number greater than 1 (default is `1.2`). By 50% would be 1.5.

y_tile_apron
: How far to extend the size landscape vertically beyond the data points. Should be a number greater than 1 (default is `1.2`). By 50% would be 1.5.

Details

Uses output from ordinate_cladistic_matrix to make morphospace plots.

Allows plotting of a third axis using the technique of Wills et al. (1994; their Figures 4 and 8; Wills 1998; his Figure 4), where solid and open indicate positive and negative values respectively, and the size of points their magnitudes.

Will automatically generate phylomorphospaces if a tree was included in the ordination.

Can also plot groups of points - whether they represent taxonomic, ecological, temporal, or spatial groupings - in different colours as well as plot translucent convex hulls around these groups, by using the `taxon_groups` and `plot_convex_hulls = TRUE` options, respectively. Note that `taxon_groups` should be in the form of a named list (see example below for how these should be formatted).

Various other options allow toggling of particular features on or off. For example, the taxon names can be shown with `plot_taxon_names = TRUE`.

Note that some features will generate legends that may initially appear to disappear off the sides of the plot, but simple resizing of the plot window (or increasing the width:height ratio if outputting to a file) should fix this.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com> and Emma Sherratt <emma.sherratt@gmail.com>

References

Wills, M. A., 1998. Cambrian and Recent disparity: the picture from priapulids. *Paleobiology*, **24**, 177-199.

Wills, M. A., Briggs, D. E. G. and Fortey, R. A., 1994. Disparity as an evolutionary index: a comparison of Cambrian and Recent arthropods. *Paleobiology*, **20**, 93-130.

See Also

assign_taxa_to_bins, plot_chronophylomorphospace, plot_morphospace_stack, plot_multi_morphospace, ordinate_cladistic_matrix

Examples

```
# Perform a PCoA ordination on the day_2016 data set:
pcoa_input <- ordinate_cladistic_matrix(cladistic_matrix = day_2016)

# Plot this as a simple bivarate morphospace:
plot_morphospace(pcoa_input = pcoa_input)

# Use the Wills technique to add a third axis (PC3):
plot_morphospace(pcoa_input = pcoa_input, z_axis = 3)

# You may need to resize the plot to see the legend for the z-axis

# Add taxon names as well:
plot_morphospace(pcoa_input = pcoa_input, z_axis = 3, plot_taxon_names = TRUE)

# Define some simple taxon groups for the data as a named list:
taxon_groups <- list(nonBurnetiamorpha = c("Biarmosuchus_tener",
  "Hipposaurus_boonstrai", "Bullacephalus_jacksoni", "Pachydectes_elsi",
  "Ictidorhinus_martinsi", "RC_20", "Herpetoskylax_hopsoni"),
  Burnetiamorpha = c("Lemurosaurus_pricei", "Lobalopex_mordax",
  "Lophorhinus_willodenensis", "Proburnetia_viatkensis", "Lende_chiweta",
  "Paraburnetia_sneeubergensis", "Burnetia_mirabilis", "BP_1_7098"))

# Set class as taxonGroups:
class(taxon_groups) <- "taxonGroups"

# Plot taxon groups including convex hulls:
plot_morphospace(pcoa_input = pcoa_input, z_axis = 3, plot_taxon_names = TRUE,
  taxon_groups = taxon_groups, plot_convex_hulls = TRUE)
```

```
# Make time-scaled first MPT for Day 2016 data set:
time_tree <- ape::read.tree(text = paste0("(Biarmosuchus_tener:0.5,",
  "(((Hipposaurus_boonstrai:3.5,(Bullacephalus_jacksoni:0.75,",
  "Pachydectes_elsi:0.75):0.75):0.75,(Lemurosaurus_pricei:7.166666667,",
  "(Lobalopex_mordax:4.333333333,((Lophorhinus_willodenensis:3.666666667,",
  "(Proburnetia_viatkensis:0.8333333333,(Lende_chiweta:2,",
  "(Paraburnetia_sneeubergensis:1,Burnetia_mirabilis:2):1):1.833333333)",
  ":0.8333333333):0.8333333333,(BP_1_7098:2.25,Niuksenitia_sukhonensis:",
  "1.25):1.25):0.8333333333):0.8333333333):3.083333333):1.95,",
  "(Ictidorhinus_martinsi:15.9,(RC_20:11.6,(Herpetoskylax_hopsoni:11.3,",
  "Lycaenodon_longiceps:0.3):0.3):0.3):0.3):0.3);"))

# Add root age to tree:
time_tree$root.time <- 269.5

# Prune incomplete taxa from tree:
time_tree <- ape::drop.tip(phy = time_tree, tip = c("Lycaenodon_longiceps",
  "Niuksenitia_sukhonensis"))

# Prune incomplete taxa from cladistic matrix:
cladistic_matrix <- prune_cladistic_matrix(cladistic_matrix = day_2016,
  taxa2prune = c("Lycaenodon_longiceps", "Niuksenitia_sukhonensis"))

# Note: the above pruning is simply to run this example and should not be
# done manually as a matter of course as the functions will automatically
# prune tips and nodes as required.

# Make new ordination with tree included (enabling phylomorphospace):
pcoa_input <- ordinate_cladistic_matrix(cladistic_matrix = cladistic_matrix,
  time_tree = time_tree)

# Plot this as a simple bivarate phylomorphospace:
plot_morphospace(pcoa_input = pcoa_input)

# Use the Wills technique to add a third axis (PC3):
plot_morphospace(pcoa_input = pcoa_input, z_axis = 3)

# You may need to resize the plot to see the legend for the z-axis

# Add taxon names as well:
plot_morphospace(pcoa_input = pcoa_input, z_axis = 3, plot_taxon_names = TRUE)

# Add taxon groups including convex hulls:
plot_morphospace(pcoa_input = pcoa_input, z_axis = 3, plot_taxon_names = TRUE,
  taxon_groups = taxon_groups, plot_convex_hulls = TRUE)
```

---

plot_morphospace_stack

Plot stacked ordination spaces

Description

Plots a stack of ordination spaces representing multiple time-slices.

Usage

```
plot_morphospace_stack(
  pcoa_input,
  taxon_ages,
  taxon_groups,
  time_bins,
  shear = 0.2,
  x_axis = 1,
  y_axis = 2,
  palette = "viridis",
  plot_cushion = 0.3,
  platform_size = 0.95,
  plot_pillars = TRUE,
  plot_crosshair = TRUE,
  plot_grid_cells = TRUE,
  plot_convex_hulls = TRUE,
  plot_timebin_names = TRUE,
  plot_tickmarks = TRUE,
  plot_group_legend = TRUE,
  group_legend_position = "bottom_right",
  point_size = 1.5
)
```

Arguments

`pcoa_input`	The main input in the format output from ordinate_cladistic_matrix.
`taxon_ages`	A two-column matrix of the first and last apperance dates (columns; `"fad"` and `"lad"`) for the taxa (as rownames) from `pcoa_input`.
`taxon_groups`	An object of class `taxonGroups`.
`time_bins`	An object of class `timeBins`.
`shear`	A single value (between 0 and 1) that determines the "sheared" visual appearance of the platforms.
`x_axis`	The ordination axis to plot on the x-axis.
`y_axis`	The ordination axis to plot nn the y-axis.
`palette`	The palette to use for plotting each element of `taxon_groups`. See palette.
`plot_cushion`	A number determining the "cushion" around the edge of each stack in which no data will be plotted. This should be larger than zero or points will "hang" over the edge. Additionally, if using a `platform_size` value in excess of one this will avoid points being hidden under overlying platforms. Note that this effectively adds empty plot space around the data, it does not remove anything.

platform_size The size of each platform as a proportion. Values of less than one will show slight gaps between platforms, whereas values in excess of one will mean platforms will appear to overlap.

plot_pillars Logical indicating whether or not to plot the pillars linking the corners of each platform.

plot_crosshair Logical indicating whether or not to plot the "crosshair" (i.e., the zero-zero lines that run through the origin of the morphospace).

plot_grid_cells
Logical indicating whether or not to plot grid cells that help visualise the distorted aspect ratio of the plot. Each cell is a square in the ordination space.

plot_convex_hulls
Logical indicating whether or not to plot convex hulls around the taxonomic groupings. Only relevant if taxon_groups is in use.

plot_timebin_names
Logical indicating whether or not to plot the names of each time bin next to each platform. I.e., the rownames from time_bins. Note if these are long they may disappear behind overlying platforms. To avoid this try using a smaller platform_size value, a larger shear value, or simply shorter or abbreviated names.

plot_tickmarks Logical indicating whether or not to plot tickmarks next to the bottom platform.

plot_group_legend
Logical indicating whether or not to plot a legend. Only relevant if using taxon_groups. Note this may obscure some points so use with caution and try picking different values for group_legend_position to avoid this.

group_legend_position
The position the group legend should be plotted. Only relevant if using taxon_groups and plot_group_legend = TRUE. Options are: "bottom_left", "bottom_right", "top_left", and "top_right".

point_size The size at which the points should be plotted. Note that here points are custom polygons and hence are not editable by normal plot options, e.g., pch or cex. At present all points are plotted as circles.

Details

This style of plot is taken from various papers by Michael Foote (Foote 1993; his Figures 2, 4, 6, 8, 10, 12, and 14; Foote 1994; his Figure 2; Foote 1995; his Figure 3; Foote 1999; his Figure 22), and can be seen elsewhere in the literature (e.g., Friedman and Coates 2006; their Figure 2c). Here multiple ordination (or morpho-) spaces are plotted in stratigraphic order (oldest at bottom) as a stacked series of "platforms" representing named time bins.

The user needs to supply three main pieces of information to use the function: 1) ordination data that includes rows (taxa) and columns (ordination axes), 2) the ages (first and last appearance dates) of the taxa sampled, and 3) the ages (first and last appearance dates) of the named time bins used.

Note that since version 0.6.1 this function has been completely rewritten to better reflect the usage of these type of figures (see citations above) as well as allow additional features. This was also done in part to standardise the function to fit the style of the other major disparity plotting functions in Claddis, such as plot_morphospace. This means the input data is now assumed to come directly

from ordinate_cladistic_matrix, but the user could easily still bring in data from elsewhere (the way the function worked previously) by reformatting it using something like:

`pcoa_input <- list(vectors = my_imported_data)`

Where my_imported_data has columns representing ordination axes (1 to N) and rownames corresponding to taxon names.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com> and Emma Sherratt <emma.sherratt@gmail.com>

References

Foote, M., 1993. Discordance and concordance between morphological and taxonomic diversity. *Paleobiology*, **19**, 185-204.

Foote, M., 1994. Morphological disparity in Ordovician-Devonian crinoids and the early saturation of morphological space. *Paleobiology*, **20**, 320-344.

Foote, M., 1995. Morphological diversification of Paleozoic crinoids. *Paleobiology*, **21**, 273-299.

Foote, M., 1999. Morphological diversity in the evolutionary radiation of Paleozoic and post-Paleozoic crinoids. *Paleobiology*, **25**, 1-115.

Friedman, M. and Coates, M. I., 2006. A newly recognized fossil coelacanth highlights the early morphological diversification of the clade. *Proceedings of the Royal Society of London B*, **273**, 245-250.

See Also

assign_taxa_to_bins, plot_chronophylomorphospace, plot_morphospace, plot_multi_morphospace, ordinate_cladistic_matrix

Examples

```
# Build taxon ages matrix for Day et al 2016 data:
taxon_ages <- matrix(data = c(269, 267, 263, 260, 265, 265, 265, 265, 257, 255, 259, 259, 258, 258,
  260, 257, 257, 255, 257, 257, 255, 252, 259, 259, 260, 258, 253, 252, 257, 255, 257, 255),
  ncol = 2, byrow = TRUE, dimnames = list(c("Biarmosuchus_tener", "Hipposaurus_boonstrai",
  "Bullacephalus_jacksoni", "Pachydectes_elsi", "Lemurosaurus_pricei", "Lobalopex_mordax",
   "Lophorhinus_willodenensis", "Proburnetia_viatkensis", "Lende_chiweta",
  "Paraburnetia_sneeubergensis", "Burnetia_mirabilis", "BP_1_7098", "Niuksenitia_sukhonensis",
   "Ictidorhinus_martinsi", "RC_20", "Herpetoskylax_hopsoni"), c("FAD", "LAD")))

# Ordinate Day et al 2016 data set:
pcoa_input <- ordinate_cladistic_matrix(cladistic_matrix = prune_cladistic_matrix(
  cladistic_matrix = day_2016,
  taxa2prune = "Lycaenodon_longiceps"))

# Build simple taxonomic groups for Day et al 2016 daat set:
taxon_groups <- list(nonBurnetiamorpha = c("Biarmosuchus_tener", "Hipposaurus_boonstrai",
  "Bullacephalus_jacksoni", "Pachydectes_elsi", "Niuksenitia_sukhonensis", "Ictidorhinus_martinsi",
  "RC_20", "Herpetoskylax_hopsoni"), Burnetiamorpha = c("Lemurosaurus_pricei", "Lobalopex_mordax",
   "Lophorhinus_willodenensis", "Proburnetia_viatkensis", "Lende_chiweta",
```

```
    "Paraburnetia_sneeubergensis", "Burnetia_mirabilis", "BP_1_7098"))

  # Set class as taxonGroups:
  class(taxon_groups) <- "taxonGroups"

  # Build a sequence of equally spaced time bins spanning Day et al. 2016 data:
  time_sequence <- seq(from = 270, to = 252, length.out = 6)

  # Reformat this sequence into named time bin matrix:
  time_bins <- matrix(
    data = c(time_sequence[1:(length(x = time_sequence) - 1)],
    time_sequence[2:length(x = time_sequence)]),
    ncol = 2,
    dimnames = list(c("Bin 1", "Bin 2", "Bin 3", "Bin 4", "Bin 5"), c("fad", "lad"))
  )

  # Set class as timeBins:
  class(time_bins) <- "timeBins"

  # Plot morphospace stack using named time bins:
  plot_morphospace_stack(
    pcoa_input = pcoa_input,
    taxon_ages = taxon_ages,
    taxon_groups = taxon_groups,
    time_bins = time_bins,
  )
```

---

plot_multi_morphospace

Plot Multiple Morphopaces

Description

Plots multiple morphospaces up to a given number of ordination axes.

Usage

```
plot_multi_morphospace(
  pcoa_input,
  n_axes = 4,
  taxon_groups = NULL,
  plot_taxon_names = FALSE,
  plot_convex_hulls = FALSE,
  plot_internal_nodes = FALSE,
  plot_edges = TRUE,
  plot_root = TRUE,
  root_colour = "red",
  palette = "viridis",
```

```
  plot_group_legend = TRUE
)
```

Arguments

`pcoa_input` The main input in the format outputted from ordinate_cladistic_matrix.

`n_axes` An integer indicating the total number of axes to plot (should minimally be three).

`taxon_groups` See plot_morphospace.

`plot_taxon_names` See plot_morphospace.

`plot_convex_hulls` See plot_morphospace.

`plot_internal_nodes` See plot_morphospace.

`plot_edges` See plot_morphospace.

`plot_root` See plot_morphospace.

`root_colour` See plot_morphospace.

`palette` See plot_morphospace.

`plot_group_legend` See plot_morphospace.

Details

Takes the output from ordinate_cladistic_matrix and uses plot_morphospace to plot the first N ordination axes.

This allows the user a better appreciation of how variance is distributed across multiple axes and all plots are scaled the saem way to further aid visualisation. Data will seem to "shrink" towards the centre of the space on higher axes as variance decreases.

Most of the options are simply passed to plot_morphospace, but the full range is not available as many will be inappropriate here (e.g., adding a z-axis).

Author(s)

Emma Sherratt <emma.sherratt@gmail.com> and Graeme T. Lloyd <graemetlloyd@gmail.com>

See Also

assign_taxa_to_bins, plot_chronophylomorphospace, plot_morphospace_stack, plot_morphospace, ordinate_cladistic_matrix

Examples

```
# Make PCoA for Day 2016 data set:
pcoa_input <- ordinate_cladistic_matrix(cladistic_matrix = day_2016)

# Define some simple taxon groups for the data as a named list:
taxon_groups <- list(nonBurnetiamorpha = c("Biarmosuchus_tener",
   "Hipposaurus_boonstrai", "Bullacephalus_jacksoni", "Pachydectes_elsi",
   "Niuksenitia_sukhonensis", "Ictidorhinus_martinsi", "RC_20",
   "Herpetoskylax_hopsoni", "Lycaenodon_longiceps"),
   Burnetiamorpha = c("Lemurosaurus_pricei", "Lobalopex_mordax",
   "Lophorhinus_willodenensis", "Proburnetia_viatkensis", "Lende_chiweta",
   "Paraburnetia_sneeubergensis", "Burnetia_mirabilis", "BP_1_7098"))

# Set class as taxonGroups:
class(taxon_groups) <- "taxonGroups"

# Plot taxon groups including convex hulls:
plot_multi_morphospace(pcoa_input, n_axes = 5, taxon_groups = taxon_groups,
  plot_convex_hulls = TRUE)

# Make time-scaled first MPT for Day 2016 data set:
time_tree <- ape::read.tree(text = paste0("(Biarmosuchus_tener:0.5,",
  "(((Hipposaurus_boonstrai:3.5,(Bullacephalus_jacksoni:0.75,",
  "Pachydectes_elsi:0.75):0.75):0.75,(Lemurosaurus_pricei:7.166666667,",
  "(Lobalopex_mordax:4.333333333,((Lophorhinus_willodenensis:3.666666667,",
  "(Proburnetia_viatkensis:0.8333333333,(Lende_chiweta:2,",
  "(Paraburnetia_sneeubergensis:1,Burnetia_mirabilis:2):1):1.833333333)",
  ":0.8333333333):0.8333333333,(BP_1_7098:2.25,Niuksenitia_sukhonensis:",
  "1.25):1.25):0.8333333333):0.8333333333):3.083333333):1.95,",
  "(Ictidorhinus_martinsi:15.9,(RC_20:11.6,(Herpetoskylax_hopsoni:11.3,",
  "Lycaenodon_longiceps:0.3):0.3):0.3):0.3):0.3):0.3);"))

# Add root age to tree:
time_tree$root.time <- 269.5

# Make same plot as before but with a phylogeny:
plot_multi_morphospace(
  pcoa_input = pcoa_input,
  n_axes = 5,
  taxon_groups = taxon_groups,
  plot_convex_hulls = TRUE
)
```

`plot_rates_character` *Visualize a rate test time series*

Description

Given the results from a rates test produces a time series visualization for a specific model.

Usage

```
plot_rates_character(test_rates_output, model_number, ...)
```

Arguments

`test_rates_output`	Rate output from test_rates.
`model_number`	The number of the model you wish to visualise from the rate output.
`...`	Other options to be passed to plot.

Details

The raw output from test_rates can be difficult to interpret without visualization and this function provides a means for doing that when the desired output is a time series (other functions will be added for other types of rate test).

The function will only work for a single model, but in practice the user may wish to produce multiple plots in which case they simply need to rn the function multiple times or setup a multipanel window first with layout, or similar.

Plots use the geoscale package to add a geologic time to the x-axis and interested users should consult the documentation there for a full list of options (passed via ...) in the function (see example below).

Calculated rates (changes per lineage million years) are plotted as filled circles and models are plotted as horizontal lines labelled by rate parameters (lambda 1, lmabda 2 etc.).

Value

Nothing is returned, but a plot is produced.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Make time-scaled first MPT for Day 2016 data set:
time_tree <- ape::read.tree(text = paste0("(Biarmosuchus_tener:0.5,",
  "(((Hipposaurus_boonstrai:3.5,(Bullacephalus_jacksoni:0.75,",
  "Pachydectes_elsi:0.75):0.75):0.75,(Lemurosaurus_pricei:7.166666667,",
  "(Lobalopex_mordax:4.333333333,((Lophorhinus_willodenensis:3.666666667,",
  "(Proburnetia_viatkensis:0.8333333333,(Lende_chiweta:2,",
  "(Paraburnetia_sneeubergensis:1,Burnetia_mirabilis:2):1):1.833333333)",
  ":0.8333333333):0.8333333333,(BP_1_7098:2.25,Niuksenitia_sukhonensis:",
  "1.25):1.25):0.8333333333):0.8333333333):3.083333333):1.95,",
  "(Ictidorhinus_martinsi:15.9,(RC_20:11.6,(Herpetoskylax_hopsoni:11.3,",
  "Lycaenodon_longiceps:0.3):0.3):0.3):0.3):0.3);"))

# Add root age to tree:
time_tree$root.time <- 269.5
```

```
# Prune continuous block from day 2016:
cladistic_matrix <- prune_cladistic_matrix(
  cladistic_matrix = day_2016,
  blocks2prune = 1
)

# Generate nine two million year time bins:
time_bins <- matrix(data = c(seq(from = 270, to = 252, length.out = 10)[1:9],
  seq(from = 270, to = 252, length.out = 10)[2:10]), ncol = 2,
  dimnames = list(LETTERS[1:9], c("fad", "lad")))

# Set class as timeBins:
class(time_bins) <- "timeBins"

# Run test rates function for two character partitions:
test_rates_output <- test_rates(
  time_tree = time_tree,
  cladistic_matrix = cladistic_matrix,
  character_partition = list(list(1:34), list(1:17, 18:34)),
  time_bins = time_bins
)

# Plot 2nd (arbitrary two-partition) character partition model:
plot_rates_character(
  test_rates_output = test_rates_output,
  model_number = 2
)
```

`plot_rates_time` *Visualize a rate test time series*

Description

Given the results from a rates test produces a time series visualization for a specific model.

Usage

```
plot_rates_time(test_rates_output, model_number, ...)
```

Arguments

`test_rates_output`	Rate output from test_rates.
`model_number`	The number of the model you wish to visualise from the rate output.
`...`	Other options to be passed to geoscalePlot.

Details

The raw output from test_rates can be difficult to interpret without visualization and this function provides a means for doing that when the desired output is a time series (other functions will be added for other types of rate test).

The function will only work for a single model, but in practice the user may wish to produce multiple plots in which case they simply need to rn the function multiple times or setup a multipanel window first with layout, or similar.

Plots use the geoscale package to add geologic time to the x-axis and interested users should consult the documentation tere for a full ist of options (passed via ...) in the function (see example below).

Calculated rates (changes per lineage million years) are plotted as filled circles and models are plotted as horizontal lines labelled by rate parameters (lambda_i).

Value

Nothing is returned, but a plot is produced.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Make time-scaled first MPT for Day 2016 data set:
time_tree <- ape::read.tree(text = paste0("(Biarmosuchus_tener:0.5,",
  "(((Hipposaurus_boonstrai:3.5,(Bullacephalus_jacksoni:0.75,",
  "Pachydectes_elsi:0.75):0.75):0.75,(Lemurosaurus_pricei:7.166666667,",
  "(Lobalopex_mordax:4.333333333,((Lophorhinus_willodenensis:3.666666667,",
  "(Proburnetia_viatkensis:0.8333333333,(Lende_chiweta:2,",
  "(Paraburnetia_sneeubergensis:1,Burnetia_mirabilis:2):1):1.833333333)",
  ":0.8333333333):0.8333333333,(BP_1_7098:2.25,Niuksenitia_sukhonensis:",
  "1.25):1.25):0.8333333333):0.8333333333):3.083333333):1.95,",
  "(Ictidorhinus_martinsi:15.9,(RC_20:11.6,(Herpetoskylax_hopsoni:11.3,",
  "Lycaenodon_longiceps:0.3):0.3):0.3):0.3):0.3);"))

# Add root age to tree:
time_tree$root.time <- 269.5

# Prune continuous block from day 2016:
cladistic_matrix <- prune_cladistic_matrix(
  cladistic_matrix = day_2016,
  blocks2prune = 1
)

# Generate nine two million year time bins:
time_bins <- matrix(data = c(seq(from = 270, to = 252, length.out = 10)[1:9],
  seq(from = 270, to = 252, length.out = 10)[2:10]), ncol = 2,
  dimnames = list(LETTERS[1:9], c("fad", "lad")))

# Set class as timeBins:
```

```
class(time_bins) <- "timeBins"

# Run test rates function for each time bin partition:
test_rates_output <- test_rates(
  time_tree = time_tree,
  cladistic_matrix = cladistic_matrix,
  time_partitions = partition_time_bins(n_time_bins = 9),
  time_bins = time_bins
)

# Plot 97th time bin partition model:
plot_rates_time(
  test_rates_output = test_rates_output,
  model_number = 97, units = "Stage", cex.ts = 1, cex.age = 1,
  abbrev = "Stage"
)
```

`plot_rates_tree` *Visualize a rate test time series*

Description

Given the results from a rates test produces a time series visualization for a specific model.

Usage

```
plot_rates_tree(test_rates_output, model_type, model_number, ...)
```

Arguments

`test_rates_output`	Rate output from test_rates.
`model_type`	The type of model to plot. Must be one of "branch" or "clade".
`model_number`	The number of the model you wish to visualise from the rate output.
`...`	Other options to be passed to plot.

Details

The raw output from test_rates can be difficult to interpret without visualization and this function provides a means for doing that when the desired output is a time series (other functions will be added for other types of rate test).

The function will only work for a single model, but in practice the user may wish to produce multiple plots in which case they simply need to rn the function multiple times or setup a multipanel window first with layout, or similar.

Plots use the geoscale package to add geologic time to the x-axis and interested users should consult the documentation tere for a full ist of options (passed via ...) in the function (see example below).

Calculated rates (changes per lineage million years) are plotted as filled circles and models are plotted as horizontal lines labelled by rate parameters (lambda_i).

Value

Nothing is returned, but a plot is produced.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Make time-scaled first MPT for Day 2016 data set:
time_tree <- ape::read.tree(text = paste0("(Biarmosuchus_tener:0.5,",
  "(((Hipposaurus_boonstrai:3.5,(Bullacephalus_jacksoni:0.75,",
  "Pachydectes_elsi:0.75):0.75):0.75,(Lemurosaurus_pricei:7.166666667,",
  "(Lobalopex_mordax:4.333333333,((Lophorhinus_willodenensis:3.666666667,",
  "(Proburnetia_viatkensis:0.8333333333,(Lende_chiweta:2,",
  "(Paraburnetia_sneeubergensis:1,Burnetia_mirabilis:2):1):1.833333333)",
  ":0.8333333333):0.8333333333,(BP_1_7098:2.25,Niuksenitia_sukhonensis:",
  "1.25):1.25):0.8333333333):0.8333333333):3.083333333):1.95,",
  "(Ictidorhinus_martinsi:15.9,(RC_20:11.6,(Herpetoskylax_hopsoni:11.3,",
  "Lycaenodon_longiceps:0.3):0.3):0.3):0.3):0.3);"))

# Add root age to tree:
time_tree$root.time <- 269.5

# Prune continuous block from day 2016:
cladistic_matrix <- prune_cladistic_matrix(
  cladistic_matrix = day_2016,
  blocks2prune = 1
)

# Generate nine two million year time bins:
time_bins <- matrix(data = c(seq(from = 270, to = 252, length.out = 10)[1:9],
  seq(from = 270, to = 252, length.out = 10)[2:10]), ncol = 2,
  dimnames = list(LETTERS[1:9], c("fad", "lad")))

# Set class as timeBins:
class(time_bins) <- "timeBins"

# Run test rates function for each clade partition:
test_rates_output <- test_rates(
  time_tree = time_tree,
  cladistic_matrix = cladistic_matrix,
  clade_partitions = as.list(x = seq(
    from = ape::Ntip(phy = time_tree) + 1,
    to = ape::Ntip(phy = time_tree) + ape::Nnode(time_tree), by = 1
  )),
  branch_partitions = lapply(X = as.list(x = seq(
    from = 1,
    to = length(x = time_tree$edge.length), by = 1
  )), as.list),
  time_bins = time_bins
```

```
)

# Plot ninth branch partition model (lowest AIC value):
plot_rates_tree(
  test_rates_output = test_rates_output,
  model_type = "branch", model_number = 9
)

# Plot third clade partition model (lowest AIC value):
plot_rates_tree(
  test_rates_output = test_rates_output,
  model_type = "clade", model_number = 3
)
```

---

print.cladisticMatrix *Compact display of a cladistic matrix*

Description

Displays a compact summary of the dimensions and nature of a cladistic matrix object.

Usage

```
## S3 method for class 'cladisticMatrix'
print(x, ...)
```

Arguments

x	An object of class "cladisticMatrix".
...	Further arguments passed to or from other methods.

Details

Displays some basic summary information on a cladistic matrix object, including number and type of characters, information about ordering, and whether variable weights are used.

Value

Nothing is directly returned, instead a text summary describing the dimensions and nature of an object of class "cladisticMatrix" is printed to the console.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

See Also

build_cladistic_matrix, compactify_cladistic_matrix, prune_cladistic_matrix, read_nexus_matrix, safe_taxonomic_reduction, write_nexus_matrix, write_tnt_matrix

Examples

```
# Show print.cladisticMatrix version of each included data sets:
print.cladisticMatrix(x = day_2016)
print.cladisticMatrix(x = gauthier_1986)
print.cladisticMatrix(x = michaux_1989)
```

`print.costMatrix` *Compact display of a costmatrix*

Description

Displays a compact summary of a costMatrix object.

Usage

```
## S3 method for class 'costMatrix'
print(x, ...)
```

Arguments

`x`	An object of class `"costMatrix"`.
`...`	Further arguments passed to or from other methods.

Details

Displays some basic summary information on a costmatrix object.

Value

Nothing is directly returned, instead a text summary describing a `"costMatrix"` object is printed to the console.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Make an unordered costmatrix:
example_costmatrix <- make_costmatrix(
  min_state = 0,
  max_state = 2,
  character_type = "unordered"
)

# Show print.costMatrix version:
print.costMatrix(x = example_costmatrix)
```

print.stateGraph *Compact display of a stategraph*

Description

Displays a compact summary of a stateGraph object.

Usage

```
## S3 method for class 'stateGraph'
print(x, ...)
```

Arguments

x	An object of class "stateGraph".
...	Further arguments passed to or from other methods.

Details

Displays some basic summary information on a stateGraph object.

Value

Nothing is directly returned, instead a text summary describing a "stateGraph" object is printed to the console.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Make an example stategraph:
example_stategraph <- list(
  n_vertices = 6,
  n_arcs = 12,
  n_states = 6,
  single_states = as.character(x = 0:5),
  type = "custom",
  arcs = data.frame(
    from = as.character(x = c(0, 1, 0, 2, 2, 5, 1, 4, 5, 4, 3, 4)),
    to = as.character(x = c(1, 0, 2, 0, 5, 2, 4, 1, 4, 5, 4, 3)),
    weight = c(1, 1, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1)
  ),
  vertices = data.frame(
    label = as.character(x = 0:5),
    in_degree = c(2, 2, 2, 1, 3, 2),
    out_degree = c(2, 2, 2, 1, 3, 2),
    eccentricity = c(3, 2, 3, 3, 2, 2),
```

```
    periphery = c(1, 0, 1, 1, 0, 0),
    centre = c(0, 1, 0, 0, 1, 1)
  ),
  radius = 2,
  diameter = 3,
  adjacency_matrix = matrix(
    data = c(
      0, 1, 1, 0, 0, 0,
      1, 0, 0, 0, 1, 0,
      1, 0, 0, 0, 0, 1,
      0, 0, 0, 0, 1, 0,
      0, 1, 0, 1, 0, 1,
      0, 0, 1, 0, 1, 0
    ),
    nrow = 6,
    byrow = TRUE,
    dimnames = list(0:5, 0:5)
  ),
  directed = FALSE,
  includes_polymorphisms = FALSE,
  polymorphism_costs = "additive",
  polymorphism_geometry = "simplex",
  polymorphism_distance = "euclidean",
  includes_uncertainties = FALSE,
  pruned = FALSE,
  dollo_penalty = 999,
  base_age = 100,
  weight = 1
)

# Set class as stateGraph:
class(x = example_stategraph) <- "stateGraph"

# Show print.stateGraph version:
print.stateGraph(x = example_stategraph)
```

`print.taxonGroups` *Compact display of taxon groups*

Description

Displays a compact summary of a taxonGroups object.

Usage

```
## S3 method for class 'taxonGroups'
print(x, ...)
```

Arguments

x	An object of class "taxonGroups".
...	Further arguments passed to or from other methods.

Details

Displays some basic summary information on a taxon groups object, including number of groups and their names and partial contents.

Value

Nothing is directly returned, instead a text summary describing a "taxonGroups" object is printed to the console.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Create a taxon groups object:
taxon_groups <- list(
  Group_A = c("Species_1", "Species_2", "Species_3"),
  Group_B = c("Species_3", "Species_4"),
  Group_C = c("Species_5", "Species_6", "Species_7", "Species_8")
)

# Set class as taxonGroups:
class(taxon_groups) <- "taxonGroups"

# Show print.taxonGroups version of each included data sets:
print.taxonGroups(x = taxon_groups)
```

---

print.timeBins *Compact display of time bins*

Description

Displays a compact summary of a timeBins object.

Usage

```
## S3 method for class 'timeBins'
print(x, ...)
```

Arguments

x	An object of class "timeBins".
...	Further arguments passed to or from other methods.

Details

Displays some basic summary information on a time bins object, including number of bins and their names and timespans.

Value

Nothing is directly returned, instead a text summary describing a "timeBins" object is printed to the console.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Create a time bins object:
time_bins <- matrix(
  data = c(99.6, 93.5, 93.5, 89.3, 89.3, 85.8, 85.8, 83.5, 83.5, 70.6, 70.6, 65.5),
  ncol = 2,
  byrow = TRUE,
  dimnames = list(
    c("Cenomanian", "Turonian", "Coniacian", "Santonian", "Campanian", "Maastrichtian"),
    c("fad", "lad")
  )
)

# Set class as timeBins:
class(time_bins) <- "timeBins"

# Show print.timeBins version of each included data sets:
print.timeBins(x = time_bins)
```

---

prune_cladistic_matrix

Prunes a character matrix of characters or taxa

Description

Prunes a character matrix of characters, taxa, or both.

Usage

```
prune_cladistic_matrix(
  cladistic_matrix,
  blocks2prune = c(),
  characters2prune = c(),
  taxa2prune = c(),
  remove_invariant = FALSE
)
```

Arguments

`cladistic_matrix`
: The cladistic matrix in the format imported by read_nexus_matrix.

`blocks2prune`
: A vector of number(s) of any blocks to prune.

`characters2prune`
: A vector of character numbers to prune.

`taxa2prune`
: A vector of taxon names to prune (these must be present in `rownames(x = cladistic_matrix$matrix)`).

`remove_invariant`
: A logical for whether invariant characters should (TRUE) or should not (FALSE, default) be pruned.

Details

Removing characters or taxa from a matrix imported using read_nexus_matrix is not simple due to associated vectors for ordering, character weights etc. To save repetitively pruning each part this function takes the matrix as input and vector(s) of either block numbers, character numbers, taxon names, or any combination thereof and returns a matrix with these items removed. Minimum and maximum values (used by calculate_morphological_distances) are also updated and the user has the option to remove constant characters this way as well (e.g, to reduce the memory required for a DNA matrix).

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

See Also

build_cladistic_matrix, compactify_cladistic_matrix, read_nexus_matrix, safe_taxonomic_reduction, write_nexus_matrix, write_tnt_matrix

Examples

```
# Remove the outgroup taxon and characters 11 and 53 from gauthier_1986:
prunedmatrix <- prune_cladistic_matrix(
  cladistic_matrix =
    gauthier_1986, characters2prune = c(11, 53), taxa2prune =
    c("Outgroup")
)

# Show priuned matrix:
prunedmatrix$matrix_1$matrix
```

`read_nexus_matrix` *Reads in a morphological #NEXUS data file*

Description

Reads in a morphological data file in #NEXUS format.

Usage

```
read_nexus_matrix(file_name, equalize_weights = FALSE)
```

Arguments

`file_name` The file name or path of the #NEXUS file.

`equalize_weights` Optional that overrides the weights specified in the file to make all characters truly equally weighted.

Details

Reads in a #NEXUS (Maddison et al. 1997) data file representing the distribution of characters (continuous, discrete, DNA etc.) in a set of taxa. Unlike read.nexus.data this function can handle polymorphisms (e.g., `(012)`).

Note that the function is generally intolerant to excursions from a standard format and it is recommended your data be formatted like the `morphmatrix.nex` example below. However, the function also produces informative error messages if (expected) excursions are discovered.

Previously all empty values (missing or inapplicable) were treated as NAs. But now anything coded as a "gap" now appears as an empty text string ("") in the matrix. Additionally, previously polymorphisms and uncertianties were both considered as polymorphisms with multiple states separated by an ampersand ("&"), but now polymorphisms use the ampersand ("&") and uncertainties use a slash ("/"), allowing for different treatment later and correct outputting when writing to #NEXUS format. (NB: TNT does not allow this distinction and so both polymorphisms and uncertainties will be output as polymorphisms.)

Value

`topper` Contains any header text or costmatrices and pertains to the entire file.

`matrix_N` One or more matrix blocks (numbered 1 to N) with associated information pertaining only to that matrix block. This includes the block name (if specificed, NA if not), the block datatype (one of "CONTINUOUS", "DNA", "NUCLEOTIDE", "PROTEIN", "RESTRICTION", "RNA", or "STANDARD"), the actual matrix (taxa as rows, names stored as rownames and characters as columns), the ordering type of each character (`"ordered"`, `"unordered"`), the character weights, the minimum and maximum values (used by Claddis' distance functions), and the original characters (symbols, missing, and gap values) used for writing out the data.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Maddison, D. R., Swofford, D. L. and Maddison, W. P., 1997. NEXUS: an extensible file format for systematic information. *Systematic Biology*, **46**, 590-621.

See Also

build_cladistic_matrix, compactify_cladistic_matrix, prune_cladistic_matrix, safe_taxonomic_reduction, write_nexus_matrix, write_tnt_matrix

Examples

```
# Create example matrix
example_matrix <- paste("#NEXUS", "", "BEGIN DATA;",
                        "\tDIMENSIONS  NTAX=5 NCHAR=5;",
                        "\tFORMAT SYMBOLS= \" 0 1 2\" MISSING=? GAP=- ;",
                        "MATRIX", "", "Taxon_1  010?0", "Taxon_2  021?0",
                        "Taxon_3  02111", "Taxon_4  011-1",
                        "Taxon_5  001-1", ";", "END;", "",
                        "BEGIN ASSUMPTIONS;",
                        "\tOPTIONS  DEFTYPE=unord PolyTcount=MINSTEPS ;",
                        "\tTYPESET * UNTITLED  = unord: 1 3-5, ord: 2;",
                        "\tWTSET * UNTITLED  = 1: 2, 2: 1 3-5;",
                        "END;", sep = "\n")

# Write example matrix to current working directory called
# "morphmatrix.nex":
cat(example_matrix, file = "morphmatrix.nex")

# Read in example matrix:
morph.matrix <- read_nexus_matrix("morphmatrix.nex")

# View example matrix in R:
morph.matrix

# Remove the generated data set:
file.remove("morphmatrix.nex")
```

---

reconstruct_ancestral_states

Determine maximum parsimony ancestral state reconstruction(s)

Description

Given a tree, or set of trees, and a cladistic matrix returns all most parsimonious reconstruction(s) for every character.

Usage

```
reconstruct_ancestral_states(
  trees,
  cladistic_matrix,
  estimate_all_nodes = FALSE,
  estimate_tip_values = FALSE,
  inapplicables_as_missing = FALSE,
  polymorphism_behaviour = "uncertainty",
  uncertainty_behaviour = "uncertainty",
  polymorphism_geometry,
  polymorphism_distance,
  state_ages,
  dollo_penalty
)
```

Arguments

`trees` A tree (phylo object) or set of trees (multiPhylo object).

`cladistic_matrix` A character-taxon matrix in the format imported by read_nexus_matrix. These should be discrete with labels matching the tip labels of `tree`.

`estimate_all_nodes` Logical that allows the user to make estimates for all ancestral values. The default (`FALSE`) will only make estimates for nodes that link coded terminals (recommended).

`estimate_tip_values` Logical that allows the user to make estimates for tip values. The default (`FALSE`) will only makes estimates for internal nodes (recommended).

`inapplicables_as_missing` Argument passed to calculate_tree_length.

`polymorphism_behaviour` Argument passed to calculate_tree_length.

`uncertainty_behaviour` Argument passed to calculate_tree_length.

`polymorphism_geometry` Argument passed to make_costmatrix.

`polymorphism_distance` Argument passed to make_costmatrix.

`state_ages` Argument passed to make_costmatrix.

`dollo_penalty` Argument passed to make_costmatrix.

Details

Although most phylogenetic inference considers character evolution on a tree, they rarely explicitly assign states to internal nodes in the process, and hence this is normally done *a posteriori* using some form of ancestral state estimation algorithm. This function uses the parsimony optimality

criterion (i.e., maximum parsimony, or the process that involves the fewest possible transitions, or, more accurately, the lowest total transition cost).

Algorithm

Like calculate_tree_length, this function is built around the generalised Swofford and Maddison (1992) algorithm, but adds the second pass as this is the one that explicitly generates *all* most parsimonious reconstructions. As it is a generalised approach it can work with the broadest range of character types and (rooted) trees of any degree of resolution - treating polytomies as hard.

Value

A list with multiple components, including:

ITEM	ITEM EXPLANATION.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Lloyd, G. T., 2018. Journeys through discrete-character morphospace: synthesizing phylogeny, tempo, and disparity. *Palaeontology*, **61**, 637-645.

Swofford, D. L. and Maddison, W. P., 1992. Parsimony, character-state reconstructions, and evolutionary inferences. *In* R. L. Mayden (ed.) Systematics, Historical Ecology, and North American Freshwater Fishes. Stanford University Press, Stanford, p187-223.

See Also

calculate_tree_length, estimate_ancestral_states

Examples

```
# Generate two trees:
trees <- ape::read.tree(text = c("(A,(B,(C,(D,E))));", "(A,((B,C),(D,E)));"))

# Build a simple 5-by-10 binary character matrix:
cladistic_matrix <- build_cladistic_matrix(
  character_taxon_matrix = matrix(
    data = sample(
      x = c("0", "1"), # ADD MISSING, POLYMORPHISM, INAPPLICABLE ETC. HERE LATER TO TEST
      size = 50,
      replace = TRUE
    ),
    nrow = 5,
    ncol = 10,
    dimnames = list(
      LETTERS[1:5],
      c()
    )
  )
```

```
)

# Reconstruct ancestral states (limiting output to all most parsimonious
# ancestral state reconstruction for every tree and character combination):
reconstruct_ancestral_states(
  trees = trees,
  cladistic_matrix = cladistic_matrix,
  estimate_all_nodes = FALSE,
  estimate_tip_values = FALSE,
  inapplicables_as_missing = FALSE,
  polymorphism_behaviour = "uncertainty",
  uncertainty_behaviour = "uncertainty",
  polymorphism_geometry = "simplex",
  polymorphism_distance = "euclidean",
  dollo_penalty = 999,
  state_ages = c(2, 1)
)$node_estimates
```

`safe_taxonomic_reduction`
Safe Taxonomic Reduction

Description

Performs Safe Taxonomic Reduction (STR) on a character-taxon matrix.

Usage

```
safe_taxonomic_reduction(cladistic_matrix)
```

Arguments

`cladistic_matrix`
: A character-taxon matrix in the format imported by read_nexus_matrix.

Details

Performs Safe Taxonomic Reduction (Wilkinson 1995).

If no taxa can be safely removed will print the text "No taxa can be safely removed", and the `str_taxa` and `removed_matrix` will have no rows.

NB: If your data contains inapplicable characters these will be treated as missing data, but this is inappropriate. Thus the user is advised to double check that any removed taxa make sense in the light of inapplicable states. (As far as I am aware this same behaviour occurs in the TAXEQ3 software.)

Value

str_taxa	A matrix listing the taxa that can be removed (junior), the taxa which they are equivalent to (senior) and the rule under which they can be safely removed (rule).
reduced_matrix	A character-taxon matrix excluding the taxa that can be safely removed.
removed_matrix	A character-taxon matrix of the taxa that can be safely removed.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Wilkinson, M., 1995. Coping with abundant missing entries in phylogenetic inference using parsimony. *Systematic Biology*, **44**, 501-514.

See Also

build_cladistic_matrix, compactify_cladistic_matrix, prune_cladistic_matrix, safe_taxonomic_reinsertion, read_nexus_matrix, write_nexus_matrix, write_tnt_matrix

Examples

```
# Performs STR on the Gauthier 1986 dataset used in Wilkinson (1995):
str_data <- safe_taxonomic_reduction(cladistic_matrix = gauthier_1986)

# View deleted taxa:
str_data$str_taxa

# View reduced matrix:
str_data$reduced_matrix

# View removed matrix:
str_data$removed_matrix
```

---

safe_taxonomic_reinsertion

Reinsert Safely Removed Taxa Into A Tree

Description

Safely reinsert taxa in a tree after they were removed from a matrix by Safe Taxonomic Reduction.

Usage

```
safe_taxonomic_reinsertion(
  input_filename,
  output_filename,
  str_taxa,
  multiple_placement_option = "exclude"
)
```

Arguments

`input_filename` A Newick-formatted tree file containing tree(s) without safely removed taxa.

`output_filename` A file name where the newly generated trees will be written out to (required).

`str_taxa` The safe taxonomic reduction table as generated by safe_taxonomic_reduction.

`multiple_placement_option` What to do with taxa that have more than one possible reinsertion position. Options are `"exclude"` (does not reinsert them; the default) or `"random"` (picks one of the possible positions and uses that - will vary stochastically if multiple trees exist).

Details

The problem with Safe Taxonomic Reduction (safe_taxonomic_reduction) is that it generates trees without the safely removed taxa, but typically the user will ultimately want to include these taxa and thus there is also a need to perform "Safe Taxonomic Reinsertion".

This function performs that task, given a Newick-formatted tree file and a list of the taxa that were safely removed and the senior taxon and rule used to do so (i.e., the `$str_taxa` part of the output from safe_taxonomic_reduction).

Note that this function operates on tree files rather than reading the trees directly into R (e.g., with ape's read.tree or read.nexus functions) as in practice this turned out to be impractically slow for the types of data sets this function is intended for (supertrees or metatrees). Importantly this means the function operates on raw Newick text strings and hence will only work on data where there is no extraneous information encoded in the Newick string, such as node labels or branch lengths.

Furthermore, in some cases safely removed taxa will have multiple taxa with which they can be safely placed. These come in two forms. Firstly, the multiple taxa can already form a clade, in which case the safely removed taxon will be reinserted in a polytomy with these taxa. In other words, the user should be aware that the function can result in non-bifurcating trees even if the input trees are all fully bifurcating. Secondly, the safely removed taxon can have multiple positions on the tree where it can be safely reinserted. As this generates ambiguity, by default (`multiple_placement_option = "exclude"`) these taxa will simply not be reinserted. However, the user may wish to still incorporate these taxa and so an additional option (`multiple_placement_option = "random"`) allows these taxa to be inserted at any of its' possible positions, chosen at random for each input topology (to give a realistic sense of phylognetic uncertainty. (Note that an exhaustive list of all possible combinations of positions is not implemented as, again, in practice this turned out to generate unfeasibly large numbers of topologies for the types of applications this function is intended for.)

Value

A vector of taxa which were not reinserted is returned (will be empty if all taxa have been reinserted) and a file is written to (`output_filename`).

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

See Also

safe_taxonomic_reduction

Examples

```
# Generate dummy four taxon trees (where taxa B, D and F were
# previously safely excluded):
trees <- ape::read.tree(text = c("(A,(C,(E,G)));", "(A,(E,(C,G)));"))

# Write trees to file:
ape::write.tree(phy = trees, file = "test_in.tre")

# Make dummy safe taxonomic reduction taxon list:
str_taxa <- matrix(data = c("B", "A", "rule_2b", "D", "C", "rule_2b",
  "F", "A", "rule_2b", "F", "C", "rule_2b"), byrow = TRUE, ncol = 3,
  dimnames = list(c(), c("junior", "senior", "rule")))

# Show that taxa B and D have a single possible resinsertion position,
# but that taxon F has two possible positions (with A or with C):
str_taxa

# Resinsert taxa safely (F will be excluded due to the ambiguity of
# its' position - multiple_placement_option = "exclude"):
safe_taxonomic_reinsertion(input_filename = "test_in.tre",
  output_filename = "test_out.tre", str_taxa = str_taxa,
  multiple_placement_option = "exclude")

# Read in trees with F excluded:
exclude_str_trees <- ape::read.tree(file = "test_out.tre")

# Show first tree with B and D reinserted:
ape::plot.phylo(x = exclude_str_trees[[1]])

# Repeat, but now with F also reinserted with its' position (with
# A or with C) chosen at random:
safe_taxonomic_reinsertion(input_filename = "test_in.tre",
  output_filename = "test_out.tre", str_taxa = str_taxa,
  multiple_placement_option = "random")

# Read in trees with F included:
random_str_trees <- ape::read.tree(file = "test_out.tre")

# Confirm F has now also been reinserted:
```

```
ape::plot.phylo(x = random_str_trees[[1]])

# Clean up example files:
file.remove(file1 = "test_in.tre", file2 = "test_out.tre")
```

---

split_out_subgraphs *Split adjacency matrix into subgraphs*

Description

Given a graph represented by an adjacency matrix splits into all connected subgraphs.

Usage

```
split_out_subgraphs(adjacency_matrix)
```

Arguments

adjacency_matrix
: An adjacency matrix where the diagonal is zeroes and the off-diagonal either ones (if the two vertices are directly connected) or zeroes (if not directly connected).

Details

This functions take any undirected graph (connected or unconnected) represented as an adjacency matrix and identifies all connected subgraphs and returns these as a list of adjacency matr(ices).

Value

A list of all connected subgraphs represented as adjacency matri(ces).

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Create an adjacency matrix representing an unconnected graph:
adjacency_matrix <- matrix(
  data = c(
    0, 0, 0, 1, 1, 0,
    0, 0, 1, 0, 0, 1,
    0, 1, 0, 0, 0, 1,
    1, 0, 0, 0, 0, 0,
    1, 0, 0, 0, 0, 0,
    0, 1, 1, 0, 0, 0
  ),
```

```
  ncol = 6,
  byrow = TRUE,
  dimnames = list(LETTERS[1:6], LETTERS[1:6])
)

# Check graph is connected:
split_out_subgraphs(adjacency_matrix = adjacency_matrix)
```

---

test_rates *Discrete character rates across trees, time, and character types*

Description

Given a tree and a cladistic-type matrix uses either likelihood ratio tests or the Akaike Information Criterion to compare rate models across branches, clades, time bins, or character partitions.

Usage

```
test_rates(
  time_tree,
  cladistic_matrix,
  time_bins,
  branch_partitions = NULL,
  character_partitions = NULL,
  clade_partitions = NULL,
  time_partitions = NULL,
  change_times = "random",
  test_type = "aic",
  alpha = 0.01,
  multiple_comparison_correction = "benjaminihochberg",
  polymorphism_state = "missing",
  uncertainty_state = "missing",
  inapplicable_state = "missing",
  time_binning_approach = "lloyd",
  all_weights_integers = FALSE,
  estimate_all_nodes = FALSE,
  estimate_tip_values = FALSE,
  inapplicables_as_missing = FALSE,
  polymorphism_behaviour = "equalp",
  uncertainty_behaviour = "equalp",
  threshold = 0.01,
  all_missing_allowed = FALSE
)
```

Arguments

`time_tree` A tree (phylo object) with branch durations that represents the relationships of the taxa in `cladistic_matrix`.

`cladistic_matrix` A character-taxon matrix in the format imported by read_nexus_matrix.

`time_bins` An object of class `timeBins`.

`branch_partitions` A list of branch(es) (edge number) partitions to test as N-rate parameter model (where N is the total number of partitions). If NULL (the default) then no partition test(s) will be made.

`character_partitions` A list of character partition(s) (character numbers) to test as N-rate parameter model (where N is the total number of partitions). If NULL (the default) then no partition test(s) will be made.

`clade_partitions` A list of clade partition(s) (node numbers) to test as N-rate parameter model (where N is the total number of partitions). If NULL (the default) then no partition test(s) will be made.

`time_partitions` A list of time bin partition(s) (numbered 1 to N) to test as N-rate parameter model (where N is the total number of partitions). If NULL (the default) then no partition test(s) will be made.

`change_times` The time at which to record the character changes. One of `"midpoint"` (changes occur at the midpoint of the branch), `"spaced"` (changes equally spaced along branch), or `"random"` (change times drawn at random from a uniform distribution; the default and recommended option). Note: this is only meaningful if testing for time bin partitions.

`test_type` Whether to apply an Akaike Information Criterion (`"aic"`; the default) or likelihood ratio test (`"lrt"`).

`alpha` The alpha value to be used for the significance tests. The default is 0.01. Thsi is only relevant if using likelihood ratio tests.

`multiple_comparison_correction` Current options are: 1. `"benjaminihochberg"` (the Benjamini and Hochberg 1995 false discovery rate approach; default and recommended), or 2. `"bonferroni"` (the Bonferroni correction). This is only relevant if using likelihood ratio tests.

`polymorphism_state` Current options are: 1. `"missing"` (converts polymorphic values to NA; the default), or 2. `"random"` (picks one of the possible polymorphic states at random).

`uncertainty_state` Current options are: 1. `"missing"` (converts uncertain values to NA; the default), or 2. `"random"` (picks one of the possible uncertain states at random).

`inapplicable_state` The only current option is `"missing"` (converts value to NA).

`time_binning_approach` One of `"close"` or `"lloyd"` (the latter is the default and recommended option).

`all_weights_integers`
: Logical for whether (`TRUE`) to reweight non-integer weights until all weights are integers or to leave them as they are (`FALSE`; the default).

`estimate_all_nodes`
: Option passed to estimate_ancestral_states.

`estimate_tip_values`
: Option passed to estimate_ancestral_states.

`inapplicables_as_missing`
: Option passed to estimate_ancestral_states.

`polymorphism_behaviour`
: Option passed to estimate_ancestral_states.

`uncertainty_behaviour`
: Option passed to estimate_ancestral_states.

`threshold`
: Option passed to estimate_ancestral_states.

`all_missing_allowed`
: Option passed to estimate_ancestral_states.

Details

Introduction

Morphological change can be captured by discrete characters and their evolution modelled as occurring along the branches of a phylogenetic tree. This function takes as primary input a character-taxon matrix of discrete characters (in the format imported by read_nexus_matrix) and a time-scaled phylogenetic tree (in the format of **paleotree** or **strap**) and begins by inferring ancestral states at the tree's internal nodes using the estimate_ancestral_states function. From here changes along individual branches can be estimated (only the minimum number of changes are inferred; see `map_stochastic_changes` for an alternative but unfinished approach) and hence rates can be calculated.

A discrete character rate can be expressed as the mean number of changes per million years (users may wish to normalise this by the number of characters for interpretation) and can be calculated for a branch (edge) of the tree, a clade (a mean rate for the edges descended from a single node), a character partition (the mean rate for a subset of the characters across all edges), or, most complex (see Lloyd 2016), the mean rate across the edges (or parts of edges) present in a time bin (defined by two values denoting the beginning and end of the time bin). In an ideal scenario these rates could be compared at face value, but that would require a large number of characters and very minimal (or zero) missing data. I.e., at an extreme of missing data if only one character can be observed along a branch it will either change (the maximum possible inferrable rate of evolution) or it will not (the minimum possible inferrable rate of evolution). In such cases it would be unwise to consider either outcome as being a significant departure from the mean rate.

Because of these complications Lloyd et al. (2012) introduced tests by which the significance of an edge (or other partitioning of the data, i.e., a clade, time bin etc.) could be considered to be significantly high or low in comparison to the mean rate for the whole tree (i.e., whether a two-rate model could be considered more likely than a one-rate model). This is achieved through a likelihood ratio test:

$$LR = valueoflikehoodfunctionunderthenull(one-rate)hypothesis/maximumpossiblevalueoflikehoodfunctionu$$

Typically we might expect the two hypotheses to be well defined a priori. E.g., an expectation that a specific branch of the tree might have a higher or lower rate than background due to some evolutionary shift. However, Lloyd et al. (2012) instead provided an exploratory approach whereby every possible one edge value was compared with the rate for the rest of the tree (and the equivalent with clades and time bins). This was the default in Claddis up to version 0.2, but this has now been replaced (since version 0.3) with a more customisable set of options that allows different types of hypotheses (e.g., partitioning the data by character), as well as more complex hypotheses (e.g., a three-rate model), to be tested. Since version 0.4 the option to replace likelihood ratio tests with the Akaike Information Criterion has also been added.

The four types of rate hypothesis

Following Cloutier (1991), Lloyd (2016) extended the two main types of rate hypotheses to four:

1. A branch rate (available here with the `branch_partitions` option).
2. A clade rate (available here with the `clade_partitions` option).
3. A time bin rate (available here with the `time_partitions` option).
4. A character partition rate (available here with the `character_partitions` option).

In Claddis (>=0.3) these partitions are defined as a list of lists of vectors where only the first N - 1 partitions need be defined. E.g., if comparing the first edge value (based on **ape** numbering, i.e., `plot(tree); edgelabels()`) to the rest of the tree then the user only needs to define the value "1" and the function will automatically add a second partition containing all other edges. This can be set with the option `branch_partitions = list(list(1))`. Similarly, to do what Lloyd et al. (2012) did and repeat the test for every edge in the tree (and assuming this variable is already named "tree") you could use, `branch_partitions = lapply(X = as.list(x = 1:nrow(tree$edge)), as.list)`.

Because of the flexibility of this function the user can define any set of edges. For example, they could test whether terminal branches have a different rate from internal branches with `branch_partitions = list(list(match(1:ape::Ntip(phy = tree), tree$edge[, 2])))`. The `clade_partitions` is really just a special subset of this type of hypothesis, but with edges being defined as descending from a specific internal node in the tree. Once again, an exploratory approach like that of Lloyd et al. (2012) can be used with: `clade_partitions = lapply(X = as.list(x = ape::Ntip(phy = tree) + (2:Nnode(tree))), as.list)`. Note that this excludes the root node as this would define a single partition and hence would represent the null hypothesis (a single rate model for the whole tree). (If using `test_type = "aic"` then the user typically *will* want a value for a single partition.) More generally clades must be defined by the node numbers they correspond to. In R an easy way to identify these is with: `plot(tree); nodelabels()`.

Time bin partitions are defined in a similar way, but are numbered 1:N starting from the oldest time bin. So if wanting to do an exploratory test of single bin partitions (and only four time bins were specified) you could use: `time_partitions = lapply(X = as.list(x = 1:4), as.list)`. Bins can be combined too, just as edges are above. For example, time bins 1 and 2 could form a single partition with: `time_partitions = list(list(1:2))`. Or if looking to test a model where each bin has its' own rate value you could use: `time_partitions = list(as.list(x = 1:3))`. Note, as before we do not need to specify the fourth bin as this will be automatically done by the function, however, `time_partitions = list(as.list(x = 1:4))` will also work. Some caution needs to be applied with N-rate models (where N is three or larger) and `test_type = "lrt"` as a result favouring such models does not necessarily endorse N-separate rates. I.e., it could simply be that one bin has

such a large excursion that overall the N-rate model fits better than the 1-rate model, but some 2-rate models might be better still. It is up to the user to check this themselves by exploring smaller combinations of bins and more genrally if exploring partitions of three or more use of the Akaike Information Criterion (`test_type = "aic"`) is recommended.

Finally, character partitions allow the user to explore whether rates vary across different character types (numbers), e.g., skeletal characters versus soft tissue characters, or cranial characters versus postcranial characters. Here characters are simply numbered 1:N (across all blocks of a matrix), but single character partitions are less likely to be of interest. As an example of use lets say the first ten characters are what we are interested in as a partition (the second partition being the remaining characters), we could use: `character_partitions = list(list(1:10))` to test for a two-rate model with `test_type = "lrt"`.

Note that the list of lists structure is critical to defining partitions as it allows them to be of different sizes and numbers. For example, one partition of three and another of six, or one set of two partitions and another set of four partitions - structures not easily realizable using vectors or matrices. However, it may not be intuitive to some users so it is recommended that the user refers to the examples above as a guide.

Additionally, it should be noted that the user can test multiple types of hypotheses simultaneously with the function. For example, performing several branch tests whilst also performing clade tests. However, it is not necessary to perform all types simultaneously (as was the case up to version 0.2) and unused partition types can be set to NULL, the default in each case.

AIC vs LRT

Since Claddis version 0.4 the option to use the Akaike Information Criterion (AIC) instead of likelihood ratio tests (LRTs) has been added (although it was not properly functional until version 0.4.2). Practically speaking the AIC uses something similar to the denominator term from the LRT (see equation above) and adds a penalty for the number of parameters (partitions). However, it also fundamentally alters the comparative framework applied and hence needs more careful attention from the user to be applied correctly. Specifically, the LRT is by its nature comparative, always comparing an N-rate partition with a one-rate partition. By contrast the AIC does not directly apply any comparison and so the user must logically supply multiple partitionings of the data in order for the results to be meaningful. It might be assumed that a user will always want to apply a single partition that pools all the data for each type of test, whether this is all the edges (branches), time bins, or characters. This will thus be the obvious comparator for any multiple partition supplied, ensuring that any more complex partitioning is minimally superior to this. Additionally, it is also logical to consider each possible way of joining partitions simpler than the most complex partition being considered. E.g., if considering a four-partition model then the user should also consider all possible three-partition and two-partition combinations of that four-partition model. This can obviously lead to some complexity in supplying partitions on the user's part and so some automating of this process is planned in future (but is not fully available yet). For an example, see partition_time_bins.

Additionally, AIC values are not simple to compare as there is no direct equivalent of the alpha value from the LRT. Instead the user can modify the AIC values returned themselves to get delta-AIC or Akaike weight values (e.g., with `geiger::aicw`). (NB: I will not explain these here as there are better explanations online.) Furthermore, since version 0.4.2 sample-size corrected AIC (AICc) is also available in the output. Note that some caution should be used in applying this if the number of partitions is equal to the sample size or only one fewer. E.g., if you have ten time bins and ten partitions you can get a negative value due to the denominator term in the AICc calculation. Thus it is advised to use the raw AIC values as first approximation and be wary if the AICc flips the

preferred model to a more complex model or models (i.e., those with more partitions) as this is the opposite of the intent of the AICc.

High versus low rates

Prior to Claddis version 0.3, rate results were classified as significantly high or significntly low as part of the output. This was done simply on whether the estimated p-value fell above or below the corrected alpha level (the significance threshold) and whether the first part of the two-rate partition had a higher or lower rate than the second part (the pooling of all other values). This simple interpretation is no longer valid here as the function can consider more than two partitions (high versus low is meaningless) and the allowing of AIC values means a significance test need not be performed either. Although the same interepretation can still be applied manually when only using two-partition tests and the LRT, other partition sizes and use of the AIC complicate this interpretation (in the same way an ANOVA is more complex than a two-sample t-test). This will also affect visualisation of the data (i.e. the simple pie chart coluring of non-significant, significntly high or significantly low rates seeen since Lloyd et al. 2012 will no longer apply). Instead the user should isolate the best model(s) and attempt to visualise these, perhaps using something like a heat map, with the mean rate for each partition being represented by an appropriate colour.

Other options

Since Claddis version 0.3 this function has allowed the user greater control with many more options than were offered previously and these should be considered carefully before running any tests.

Firstly, the user can pick an option for `change_times` which sets the times character changes are inferred to occur. This is only relevant when the user is performing time bin partition test(s) as this requires some inference to be made about when changes occur on branches that may span multiple time bins. The current options are: `"midpoint"` (all changes are inferred to occur midway along the branch, effectively mimicking the approach of Ruta et al. 2006), `"spaced"` (all changes are inferred to occur equally spaced along the branch, with changes occurring in character number order), or `"random"` (changes are assigned a random time by drawing from a uniform distribution between the beginning and end of each branch). The first of these is likely to lead to unrealistically "clumped" changes and by extension implies completely correlated character change that would violate the assumptions of the Poisson distribution that underlies the significance tests here (Lloyd et al. 2012). At the other extreme, the equally spaced option will likely unrealistically smooth out time series and potentially make it harder to reject the single-rate null (leading to Type II errors). For these reasons, the random option is recommended and is set as the default. However, because it is random this makes the function stochastic (the answer can vary each time it is run) and so the user should therefore run the function multiple times if using this option (i.e., by using a for loop) and aggregating the results at the end (e.g., as was done by previous authors; Lloyd et al. 2012; Close et al. 2015).

Secondly, the `alpha` value sets the significance threshold by which the likelihood ratio test's resulting p-value is compared (i.e., it is only reelevant when `test_type = "lrt"`. Following Lloyd et al. (2012) this is set lower (0.01) than the standard 0.05 value by default as those authors found rates to be highly heterogenous in their data set (fossil lungfish). However, this should not be adopted as a "standard" value without question (just as 0.05 shouldn't). Note that the function also corrects for multiple comparisons (using the `multiple_comparison_correction` option) to avoid Type I errors (false positives). It does so (following Lloyd et al. 2012) using the Benjamini-Hochberg (Benjamini and Hochberg 1995) False Discovery Rate approach (see Lloyd et al. 2012 for a discussion of why), but the Bonferroni correction is also offered here (albeit not recommended).

Thirdly, polymorphisms and uncertainities create complications for assessing character changes along branches. These can occur at the tips (true polymorphisms or uncertainties in sampled taxa)

and internal nodes (uncertainty over the estimated ancestral state). There are two options presented here, and applicable to both `polymorphism_state` and `uncertainty_state` (allowing these to be set separately). These are to convert such values to missing (NA) or to pick one of the possible states at random. Using missing values will increase overall uncertainty and potentially lead to Type II errors (false negatives), but represents a conservative solution. The random option is an attempt to avoid Type II errors, but can be considered unrealistic, especially if there are true polymorphisms. Additionally, the random option will again make the function stochastic meaning the user should run it multiple times amd aggregate the results. Note that if there are no polymorphisms or uncertainties in the character-taxon matrix the latter can still occur with ancestral state estimates, especially if the threshold value is set at a high value (see estimate_ancestral_states for details).

Fourthly, inapplicable characters can additionally complicate matters as they are not quite the same as missing data. I.e., they can mean that change in a particular character is not even possible along a branch. However, there is no easy way to deal with such characters at present so the user is not presented with a true option here - currently all inapplicable states are simply converted to missing values by the function. In future, though, other options may be available here. For now it is simply noted that users should be careful in making inferences if there are inapplicable characters in their data and should perhaps consider removing them with prune_cladistic_matrix to gauge their effect.

Fifthly, there are currenty two further options for assessing rates across time bins. As noted above a complication here is that character changes (the rate numerator) and character completeness (part of the rate denominator) are typically assessed on branches. However, branches will typically span time bin boundaries and hence many bins will contain only some portion of particular branches. The exact portion can be easily calculated for branch durations (the other part of the rate denominator) and the `change_times` option above is used to set the rate numerator, however, completeness remains complex to deal with. The first attempt to deal with this was made by Close et al. (2015) who simply used weighted mean completeness by calculating the proportion of a branch in each bin as the weight and multiplying this by each branch's completeness (the `"close"` option here). However, this may lead to unrealistic "smoothing" of the data and perhaps more importantly makes no account of which characters are known in a bin. Lloyd (2016) proposed an alternative "subtree" approach which assesses completeness by considering each character to be represented by a subtree where only branches that are complete are retained then branch durations in each bin are summed across subtrees such that the duration term automatically includes completeness (the `"lloyd"` option here). Here the latter is strongly recommended, for example, because this will lead to the same global rate across the whole tree as the branch, clade or character partitions, whereas the Close approach will not.

Sixthly, all character changes are weighted according to the weights provided in the input character-taxon matrix. In many cases these will simply all be one, although see the equalise weights option in read_nexus_matrix. However, when weights vary they can create some issues for the function. Specifically, changes are expected to be in the same (integer) units, but if weights vary then they have to be modelled accordingly. I.e., a character twice the weight of another may lead to a single change being counted as two changes. This is most problematic when the user has continuous characters which are automatically converted to gap-weighted (Thiele 1993) characters. However, this conversion creates drastically down-weighted characters and hence the user may wish to set the `all_weights_integers` option to TRUE. Note that reweighting will affect the results and hence shifting the weights of characters up or down will necessarily lead to shifts in the relative Type I and II errors. This is an unexplored aspect of such approaches, but is something the user should be aware of. More broadly it is recommended that continuous (or gap-weighted) characters be avoided when using this function.

Finally, the remaining options (`estimate_all_nodes`, `estimate_tip_values`, `inapplicables_as_missing`, `polymorphism_behaviour`, `uncertainty_behaviour`, and `threshold`) are all simply passed directly to estimate_ancestral_states for estimating the ancestral states and users should consult the help file for that function for further details.

Note that currently the function cannot deal with costmatrices and that the terminal versus internal option from Brusatte et al. (2014) is yet to be implemented.

Output

The output for each LRT test (i.e., the components of the `branch_test_results`, `character_test_results`, `clade_test_results` and `time_test_results` parts of the output) includes three main parts:

1. Rates.
2. p_value.
3. CorrectedAlpha.

Or for each AIC test there are:

1. Rates.
2. AIC.
3. AICc.

For each rate test the `Rates` part of the output is a vector of the absolute rate (number of changes per million years) for each partition in the test (in the order they were supplied to the function). So, for example, a branch rate for the sixth edge in a tree would be the rate for the sixth edge followed by the pooled rate for all other edges. The length of the vector is the length of the number of partitions.

The p_value is a single value indicating the probability that the likelihood ratio (see above and Lloyd et al. 2012) is one, i.e., that the likelihoods of the one-rate and N-rate models are the same.

The CorrectedAlpha is the alpha-value that should be used to determine the significance of the current partition test (i.e., The p_value, above). If the p_value exceeds the CorrectedAlpha then the null (single-rate) hypothesis should be accepted, if lower then the null should be rejected in favour of the N-rate hypthesis. Note that the CorrectedAlpha will not typically be the same for each partition and will also typically be different from the input `alpha` value due to the `multiple_comparison_correction` option used.

The AIC is the Akaike Information Criterion, and is relatively meaningless on its own and can only really be used to compare with the AIC values for other partitions of the data. The AICc is simply the sample-size corrected version of the AIC and is preferable when sample sizes are small.

Value

`time_bins_used` The time binning used (NB: May be slightly altered from the input values).

`inferred_character_changes` Matrix of inferred character changes.

`mean_rate` The global (mean) character rate in changes per million years. I.e, the average rate across all characters, branches and time bins, effectively the null hypothesis for any test performed.

`continuous_characters_discretized`
: Whether or not continuous characters were converted to discrete characters (important for handling the data in downstream analys(es)).

`branch_test_results`
: List of branch partition results (corresponding to `branch_partitions`. NULL if not requested.

`character_test_results`
: List of character partition results (corresponding to `character_partitions`. NULL if not requested.

`clade_test_results`
: List of clade partition results (corresponding to `clade_partitions`. NULL if not requested.

`time_test_results`
: List of time bin partition results (corresponding to `time_partitions`. NULL if not requested.

`branch_rates`
: Matrix showing calculated rates for each branch. NULL if `branch_partitions` is not requested.

`character_rates`
: Matrix showing calculated rates for each character. NULL if `character_partitions` is not requested.

`clade_rates`
: Matrix showing calculated rates for each clade. NULL if `clade_partitions` is not requested.

`time_rates`
: Matrix showing calculated rates for each time bin. NULL if `time_partitions` is not requested.

`time_tree`
: The time-scaled input tree used as input (provided as output for use with visualisation functions).

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com> and Steve C. Wang <scwang@swarthmore.edu>

References

Benjamini, Y. and Hochberg, Y., 1995. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society, Series B*, **57**, 289-300.

Brusatte, S. L., Lloyd, G. T., Wang, S. C. and Norell, M. A., 2014. Gradual assembly of avian body plan culminated in rapid rates of evolution across dinosaur-bird transition. *Current Biology*, **24**, 2386-2392.

Close, R. A., Friedman, M., Lloyd, G. T. and Benson, R. B. J., 2015. Evidence for a mid-Jurassic adaptive radiation in mammals. *Current Biology*, **25**, 2137-2142.

Cloutier, R., 1991. Patterns, trends, and rates of evolution within the Actinistia. *Environmental Biology of Fishes*, **32**, 23-58.

Lloyd, G. T., 2016. Estimating morphological diversity and tempo with discrete character-taxon matrices: implementation, challenges, progress, and future directions. *Biological Journal of the Linnean Society*, **118**, 131-151.

Lloyd, G. T., Wang, S. C. and Brusatte, S. L., 2012. Identifying heterogeneity in rates of morphological evolution: discrete character change in the evolution of lungfish (Sarcopterygii; Dipnoi). *Evolution*, **66**, 330-348.

Ruta, M., Wagner, P. J. and Coates, M. I., 2006. Evolutionary patterns in early tetrapods. I. Rapid initial diversification followed by decrease in rates of character change. *Proceedinsg of the Royal Society of London B*, **273**, 2107-2111.

Thiele, K.. 1993. The Holy Grail of the perfect character: the cladistic treatment of morphometric data. *Cladistics*, **9**, 275-304.

Examples

```
# Set random seed:
set.seed(seed = 17)

# Generate a random tree for the Michaux data set:
time_tree <- ape::rtree(n = nrow(michaux_1989$matrix_1$matrix))

# Update taxon names to match those in the data matrix:
time_tree$tip.label <- rownames(x = michaux_1989$matrix_1$matrix)

# Set root time by making youngest taxon extant:
time_tree$root.time <- max(diag(x = ape::vcv(phy = time_tree)))

# Set four equal ength time bins spanning range of tree:
time_bins <- matrix(data = c(seq(from = time_tree$root.time, to = 0,
  length.out = 5)[1:4], seq(from = time_tree$root.time, to = 0,
  length.out = 5)[2:5]), ncol = 2, dimnames = list(LETTERS[1:4],
  c("fad", "lad")))

# Set class as timeBins:
class(time_bins) <- "timeBins"

# Get discrete character rates:
x <- test_rates(
  time_tree = time_tree,
  cladistic_matrix = michaux_1989,
  time_bins = time_bins,
  branch_partitions = lapply(X = as.list(x = 1:nrow(time_tree$edge)), as.list),
  character_partitions = lapply(X = as.list(x = 1:3), as.list),
  clade_partitions = lapply(X = as.list(x = ape::Ntip(phy = time_tree) +
    (2:ape::Nnode(phy = time_tree))), as.list),
  time_partitions = lapply(X = as.list(x = 1:4), as.list),
  change_times = "random",
  alpha = 0.01,
  polymorphism_state = "missing",
  uncertainty_state = "missing",
  inapplicable_state = "missing",
  time_binning_approach = "lloyd"
)
```

trim_marginal_whitespace

Trims marginal whitespace

Description

Trims any marginal whitespace from a vector of character string(s).

Usage

```
trim_marginal_whitespace(x)
```

Arguments

x	A character string

Details

Trims any marginal whitespace (spaces or tabs) from a vector of character string(s).

Value

A vector of character string(s) with any leading or trailing whitespace removed.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

Examples

```
# Example string:
x <- "   \td s f\t s  "

# Trim only marginal whitespace:
trim_marginal_whitespace(x)
```

`trim_matrix` *Trims a morphological distance matrix*

Description

Trims a morphological distance matrix by removing objects that cause empty cells.

Usage

```
trim_matrix(distance_matrix, tree = NULL)
```

Arguments

`distance_matrix`
: A distance matrix in the format created by calculate_morphological_distances.

`tree`
: If the distance matrix includes ancestors this should be the tree (phylo object) used to estimate their states.

Details

Trims a morphological distance matrix by removing nodes (terminal or internal) that cause empty cells allowing it to be passed to an ordination function such as cmdscale.

Some distances are not calculable from cladistic matrices if there are taxa that have no coded characters in common. This algorithm iteratively removes the taxa responsible for the most empty cells until the matrix is complete (no empty cells).

If the matrix includes estimated ancestral states the user should also provide the tree used (as the `tree` argument). The function will then also remove the tips from the tree and where reconstructed ancestors also cause empty cells will prune the minimum number of descendants of that node. The function will then renumber the nodes in the distance matrix so they match the pruned tree.

Value

`distance_matrix`
: A complete distance matrix with all cells filled. If there were no empty cells will return original.

`tree`
: A tree (if supplied) with the removed taxa (see below) pruned. If no taxa are dropped will return the same tree as inputted. If no tree is supplied this is set to NULL.

`removed_taxa`
: A character vector listing the taxa removed. If none are removed this will be set to NULL.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

See Also

calculate_morphological_distances

Examples

```
# Get morphological distances for Michaux (1989) data set:
distances <- calculate_morphological_distances(cladistic_matrix = michaux_1989)

# Attempt to trim max.distance_matrix:
trim_matrix(distance_matrix = distances$distance_matrix)
```

---

write_nexus_matrix *Writes out a morphological #NEXUS data file*

Description

Writes out a morphological data file in #NEXUS format.

Usage

```
write_nexus_matrix(cladistic_matrix, file_name)
```

Arguments

`cladistic_matrix`	The cladistic matrix in the format imported by read_nexus_matrix.
`file_name`	The file name to write to. Should end in `.nex`.

Details

Writes out a #NEXUS (Maddison et al. 1997) data file representing the distribution of characters in a set of taxa. Data must be in the format created by importing data with read_nexus_matrix.

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Maddison, D. R., Swofford, D. L. and Maddison, W. P., 1997. NEXUS: an extensible file format for systematic information. *Systematic Biology*, **46**, 590-621.

See Also

write_tnt_matrix

build_cladistic_matrix, compactify_cladistic_matrix, prune_cladistic_matrix, read_nexus_matrix, safe_taxonomic_reduction, write_tnt_matrix

Examples

```
# Write out Michaux 1989 to current working directory:
write_nexus_matrix(cladistic_matrix = michaux_1989, file_name = "michaux_1989.nex")

# Remove file when finished:
file.remove(file1 = "michaux_1989.nex")
```

`write_tnt_matrix` *Writes out a morphological TNT data file*

Description

Writes out a morphological data file in Hennig86/TNT format.

Usage

```
write_tnt_matrix(cladistic_matrix, file_name, add_analysis_block = FALSE)
```

Arguments

`cladistic_matrix`
: A cladistic matrix in the format imported by read_nexus_matrix.

`file_name`
: The file name to write to. Should end in `.tnt`.

`add_analysis_block`
: Whether or not to add analysis block (i.e., tree search commands).

Details

Writes out a TNT (Goloboff et al. 2008; Goloboff and Catalano 2016) data file representing the distribution of discrete morphological characters in a set of taxa. Data must be in the format created by importing data with read_nexus_matrix.

Note that the format can currently deal with continuous characters, sequence (DNA) data, and combinations of these and discrete morphology, but not yet the morphometric format introduced in Goloboff and Catalano (2016).

Author(s)

Graeme T. Lloyd <graemetlloyd@gmail.com>

References

Goloboff, P. A. and Catalano, S. A., 2016. TNT version 1.5, including a full implementation of phylogenetic morphometrics. *Cladistics*, **32**, 221-238.

Goloboff, P., Farris, J. and Nixon, K., 2008. TNT, a free program for phylogenetic analysis. *Cladistics*, **24**, 774-786.

See Also

write_nexus_matrix

build_cladistic_matrix, compactify_cladistic_matrix, prune_cladistic_matrix, read_nexus_matrix, safe_taxonomic_reduction, write_nexus_matrix

Examples

```
# Write out Michaux 1989 to current working directory:
write_tnt_matrix(cladistic_matrix = michaux_1989, file_name = "michaux_1989.tnt")

# Remove file when finished:
file.remove(file1 = "michaux_1989.tnt")
```

Index

∗ **datasets**
day_2016, 78
gauthier_1986, 101
michaux_1989, 120

ace, *80*
add_polymorphisms_to_costmatrix, 5, *27*, *110*, *113*, *115*
add_uncertainties_to_costmatrix, 13, *27*, *115*
align_matrix_block, 16
ape, *79–81*, *89*, *100*, *122*, *135*, *174*
assign_taxa_to_bins, 18, *123*, *144*, *148*, *152*, *154*

bin_changes, 20
bin_character_completeness, 21
bin_edge_lengths, 23
build_cladistic_matrix, 24, *66*, *161*, *167*, *169*, *173*, *189*, *191*

calculate_g, 26, *33*
calculate_gmax, *28*, 32
calculate_kardashian_index, 37
calculate_morphological_distances, 39, *44*, *50*, *121*, *122*, *167*, *188*, *189*
calculate_MPD, 43
calculate_tree_length, 44, *170*, *171*
calculate_WMPD, 49
check_cladisticMatrix, 51
check_costMatrix, 51, *99*, *131*
check_stateGraph, 52
check_taxonGroups, 53
check_timeBins, 54
Claddis *(*Claddis-package*)*, 4
Claddis-package, 4
classify_costmatrix, 55
cmdscale, *188*
compactify_cladistic_matrix, *25*, 66, *161*, *167*, *169*, *173*, *189*, *191*
convert_adjacency_matrix_to_costmatrix, 67, *70*, *72*, *75*, *76*, *92*
convert_costmatrix_to_stategraph, 69, *72*, *86*, *94*
convert_state_tree_to_adjacency_matrix, *68*, 74, *109*
convert_stategraph_to_costmatrix, 71
count_cherries, 76

date_nodes, 77
day_2016, 78
drop.tip, *79*, *100*
drop_time_tip, 78, *100*

estimate_ancestral_states, 79, *122*, *171*, *179*, *183*, *184*
estimate_squared_change_ancestors, 82

find_costmatrix_minimum_span, 85
find_descendant_edges, 87
find_linked_edges, 88
find_minimum_spanning_edges, 89
find_mrca, 90
find_shortest_costmatrix_path, *86*, 91, *94*, *99*
find_stategraph_minimum_span, *86*, 93, *94*
find_time_bin_midpoints, 95
find_unique_trees, 97
fix_costmatrix, 98, *131*
fix_root_time, *79*, 100

gauthier_1986, 101
geoscale, *156*, *158*, *159*
geoscalePlot, *157*

is.cladisticMatrix, 101
is.costMatrix, 102
is.stateGraph, 103
is.taxonGroups, 104
is.timeBins, 105
is_graph_connected, 106

layout, *156*, *158, 159*
locate_bracket_positions, *68*, *76*, 108

make_costmatrix, *45, 46*, *48*, *92*, 109, *126*, *128*, *170*
make_labels, 117
map_dollo_changes, *112*, 118
match_tree_edges, 119
michaux_1989, 120
mst, *89*

ordinate_cladistic_matrix, *18*, 121, *143, 144*, *146–148*, *150*, *152*, *154*

palette, *144*, *146*, *150*
partition_time_bins, 124, *181*
pcoa, *122, 123*
permute_all_polymorphisms, *116*, 125, *128*
permute_all_treeshape_labellings, 126
permute_all_uncertainties, *126*, 127
permute_connected_graphs, 128
permute_costmatrices, 130
permute_graph_splits, 132
permute_restricted_compositions, 133
permute_tipstates, *133*, 134
permute_treeshapes, *126*, 140
phytools, *80, 81*
plot, *156*, *159*
plot_changes_on_tree, 141
plot_chronophylomorphospace, *18*, *123*, 143, *148*, *152*, *154*
plot_morphospace, *18*, *123*, *144*, 145, *151, 152*, *154*
plot_morphospace_stack, *18*, *123*, *144*, *148*, 149, *154*
plot_multi_morphospace, *18*, *123*, *144*, *147, 148*, *152*, 153
plot_rates_character, 155
plot_rates_time, 157
plot_rates_tree, 159
print.cladisticMatrix, 161
print.costMatrix, 162
print.stateGraph, 163
print.taxonGroups, 164
print.timeBins, 165
prune_cladistic_matrix, *25*, *66*, *161*, 166, *169*, *173*, *183*, *189*, *191*

read.nexus, *174*
read.nexus.data, *168*
read.tree, *174*
read_nexus_matrix, *21*, *25*, *39*, *45*, *66*, *78*, *80*, *101*, *121, 122*, *161*, *167*, 168, *170*, *172, 173*, *178, 179*, *183*, *189–191*
reconstruct_ancestral_states, *46–48*, 169
rerootingMethod, *80*

safe_taxonomic_reduction, *25*, *66*, *161*, *167*, *169*, 172, *174, 175*, *189*, *191*
safe_taxonomic_reinsertion, *173*, 173
split_out_subgraphs, 176

test_rates, *124*, *142*, *156–159*, 177
trim_marginal_whitespace, 187
trim_matrix, *90*, *122, 123*, 188

unique.multiPhylo, *97, 98*

write_nexus_matrix, *25*, *66*, *161*, *167*, *169*, *173*, 189, *191*
write_tnt_matrix, *25*, *66*, *161*, *167*, *169*, *173*, *189*, 190