Package: simplextree (via r-universe)

August 30, 2024

Type Package

Title Provides Tools for Working with General Simplicial Complexes

Version 1.0.2

Date 2021-07-21

Depends R (>= 3.4.0)

Maintainer Matt Piekenbrock <matt.piekenbrock@gmail.com>

Description Provides an interface to a Simplex Tree data structure, which is a data structure aimed at enabling efficient manipulation of simplicial complexes of any dimension. The Simplex Tree data structure was originally introduced by Jean-Daniel Boissonnat and Clément Maria (2014) <doi:10.1007/s00453-014-9887-3>.

Language en-US

License MIT + file LICENSE

URL https://github.com/peekxc/simplextree

LinkingTo Rcpp

Imports Rcpp (>= 0.12.10), methods, magrittr

Encoding UTF-8

LazyData true

SystemRequirements C++11

RoxygenNote 7.2.0

Roxygen list(markdown = TRUE)

Suggests testthat, knitr, rmarkdown, covr

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

RemoteUrl https://github.com/peekxc/simplextree

RemoteRef HEAD

RemoteSha 525024fe5cb14515e154202caf1e8a6487f20d5a

Contents

simplextree-package *'simplextree' package*

Description

Provides an R/Rcpp implementation of a Simplex Tree data structure and its related tools.

Details

This package provides a lightweight implementation of a Simplex Tree data structure, exported as an Rcpp Module. The current implementation provides a limited API and a subset of the functionality described in the paper.

Author(s)

Matt Piekenbrock

adjacent *Adjacent vertices*

Description

Returns a vector of vertex ids that are immediately adjacent to a given vertex.

Usage

```
adjacent(st, vertices = NULL)
```

Arguments

st	a simplex tree.
vertices	a numeric vector of vertex ids.

Value

a list of double vectors of vertices adjacent to each of `vertices` in `st` (or `numeric(0)` for vertices not in `st`), unlisted to a single vector if `length(vertices == 1`.

See Also

Other vertex-level operations: `degree()`

Examples

```
st <- simplex_tree(1:3)
st %>% adjacent(2)
# 1 3
```

clear *Clear a simplex tree*

Description

Removes all simplices from the simplex tree, except the root node.

Usage

```
clear(st)
```

Arguments

st	a simplex tree object.

Value

the simplex tree st with simplices removed, invisibly.

See Also

Other complex-level operations: contract(), expand(), threshold()

Examples

```
st <- simplex_tree()
st %>% insert(1:3)
print(st) ## Simplex Tree with (3, 3, 1) (0, 1, 2)-simplices
st %>% clear()
print(st) ## < empty simplex tree >
```

clone *Clone a simplex tree*

Description

Performs a deep-copy on the supplied simplicial complex.

Performs a deep-copy on the supplied simplicial complex.

Usage

```
clone(st)
```

```
clone(st)
```

Arguments

st	a simplex tree.

Details

A clone is produced by serializing the input simplex tree st and deserializing it into a new simplex tree that is then returned.

Value

a new simplex tree copied from st.

See Also

Other serialization methods: deserialize(), serialize()

---

cofaces *Coface traversal*

Description

Generates a coface traversal on the simplex tree.

Usage

```
cofaces(st, sigma)
```

Arguments

st	the simplex tree to traverse.
sigma	simplex to start the traversal at.

Value

a traversal (object of class "st_traversal").

See Also

Other traversals: coface_roots(), faces(), k_simplices(), k_skeleton(), level_order(), link(), maximal(), preorder(), traverse()

coface_roots *Coface roots traversal*

Description

Generates a coface roots traversal on the simplex tree.

The coface roots of a given simplex `sigma` are the roots of subtrees in the trie whose descendents (including the roots themselves) are cofaces of `sigma`. This traversal is more useful when used in conjunction with other traversals, e.g. a *preorder* or *level order* traversal at the roots enumerates the cofaces of `sigma`.

Usage

```
coface_roots(st, sigma)
```

Arguments

`st`	the simplex tree to traverse.
`sigma`	simplex to start the traversal at.

Value

a traversal (object of class `"st_traversal"`).

See Also

Other traversals: `cofaces()`, `faces()`, `k_simplices()`, `k_skeleton()`, `level_order()`, `link()`, `maximal()`, `preorder()`, `traverse()`

collapse *Elementary collapse*

Description

Performs an elementary collapse.

Usage

```
collapse(st, pair, w = NULL)
```

Arguments

`st`	a simplex tree.
`pair`	list of simplices to collapse.
`w`	vertex to collapse to, if performing a vertex collapse.

Details

This function provides two types of *elementary collapses*.

The first type of collapse is in the sense described by (1), which is summarized here. A simplex σ is said to be collapsible through one of its faces τ if σ is the only coface of τ (excluding τ itself). This function checks whether its possible to collapse σ through τ, (if τ has σ as its only coface), and if so, both simplices are removed. `tau` and `sigma` are sorted before comparison. To perform this kind of elementary collapse, call `collapse` with two simplices as arguments, i.e. `tau` before `sigma`.

Alternatively, this method supports another type of elementary collapse, also called a *vertex collapse*, as described in (2). This type of collapse maps a pair of vertices into a single vertex. To use this collapse, specify three vertex ids, the first two representing the free pair, and the last representing the target vertex to collapse to.

Value

boolean indicating whether the collapse was performed.

References

Boissonnat, Jean-Daniel, and Clement Maria. "The simplex tree: An efficient data structure for general simplicial complexes." Algorithmica 70.3 (2014): 406-427.

Dey, Tamal K., Fengtao Fan, and Yusu Wang. "Computing topological persistence for simplicial maps." Proceedings of the Thirtieth Annual Symposium on Computational Geometry. ACM, 2014.

Examples

```
st <- simplextree::simplex_tree(1:3)
st %>% print_simplices()
# 1, 2, 3, 1 2, 1 3, 2 3, 1 2 3
st %>% collapse(list(1:2, 1:3))
# 1, 2, 3, 1 3, 2 3=

st %>% insert(list(1:3, 2:5))
st %>% print_simplices("column")
# 1 2 3 4 5 1 1 2 2 2 3 3 4 1 2 2 2 3 2
#           2 3 3 4 5 4 5 5 2 3 3 4 4 3
#                           3 4 5 5 5 4
#                                     5

st %>% collapse(list(2:4, 2:5))
st %>% print_simplices("column")
# 1 2 3 4 5 1 1 2 2 2 3 3 4 1 2 2 3
#           2 3 3 4 5 4 5 5 2 3 4 4
#                           3 5 5 5
```

`combinadic` *k-combinations and binomial coefficients*

Description

Map between k-combinations of n and their lexicographic positions, and recover binomial coefficient numerators.

Usage

```
nat_to_sub(i, n, k)

sub_to_nat(s, n)

inverse.choose(x, k)
```

Arguments

`i`	vector of integers in the range `c(1L, choose(n, k))`.
`n`	integer numerator of the binomial coefficient.
`k`	integer denominator of the binomial coefficient.
`s`	matrix whose columns represent k-combinations.
`x`	a binomial coefficient (n, k).

Details

`nat_to_sub` computes the `i`^th (`n` choose `k`) combination in the lexicographic order. `sub_to_nat` computes the position of a combination `s` out of all lexicographically-ordered (`n` choose `k`) combinations. The values are calculated via a lexicographically-ordered combinadic mapping.

In general, `nat_to_sub` is *not* intended to be used to *generate* all k-combinations in the combinadic mapping. For that, use combn.

`inverse.choose` inverts the binomial coefficient for general (n, k). That is, given the denominator `k` and `x = choose(n, k)`, find `n`.

Value

an integer matrix whose columns give the combinadics of `i` (`nat_to_sub`), an integer vector of the positions of the combinations `s` (`sub_to_nat`), or the integer numerator of the binomial coefficient `x` (`inverse.choose`).

References

McCaffrey, J. D. "Generating the mth lexicographical element of a mathematical combination." MSDN Library (2004).

Examples

```
library(simplextree)
all(nat_to_sub(seq(choose(100,2)), n = 100, k = 2) == combn(100,2))

## Generating pairwise combinadics is particularly fast
## Below: test to generate ~ 45k combinadics
## (note: better to use microbenchmark)
system.time({
  x <- seq(choose(300,2))
  nat_to_sub(x, n = 300, k = 2L)
})

## Compare with generating raw combinations
system.time(combn(300,2))

100 == inverse.choose(choose(100,2), k = 2)
# TRUE
12345 == inverse.choose(choose(12345, 5), k = 5)
# TRUE
```

`contract` *Edge contraction*

Description

Performs an edge contraction.

Usage

```
contract(st, edge)
```

Arguments

`st`	a simplex tree.
`edge`	an edge to contract, as a 2-length vector.

Details

This function performs an *edge contraction* in the sense described by (1), which is summarized here. Given an edge (v, w), w is contracted to v if w is removed from the complex and the link of v is augmented with the link of w. This may be thought as applying the mapping:

$$f(u) = v$$

if $u = w$ and identity otherwise, to all simplices in the complex.

`edge` is **not** sorted prior to contraction: the second vertex of the edge is always contracted to the first. Note that edge contraction is not symmetric.

Value

the contracted simplex tree `st`, invisibly.

References

Boissonnat, Jean-Daniel, and Clement Maria. "The simplex tree: An efficient data structure for general simplicial complexes." Algorithmica 70.3 (2014): 406-427.

See Also

Other complex-level operations: `clear()`, `expand()`, `threshold()`

Examples

```
st <- simplex_tree(1:3)
st %>% print_simplices()
# 1, 2, 3, 1 2, 1 3, 2 3, 1 2 3
st %>% contract(c(1, 3)) %>% print_simplices()
# 1, 2, 1 2
```

`degree` *Vertex degree*

Description

Returns a list of vertex ids that are immediately adjacent to a given vertex. If a given vertex does not have any adjacencies, a vector of length 0 is returned.

Usage

```
degree(st, vertices = NULL)
```

Arguments

`st`	a simplex tree.
`vertices`	a numeric vector of vertex ids.

Value

an integer vector of degrees of `vertices` in `st` (taken to be `0` for vertices not in `st`).

See Also

Other vertex-level operations: `adjacent()`

deserialize *Deserialize the simplex tree*

Description

Provides a compressed serialization interface for the simplex tree.

Usage

```
deserialize(complex, st = NULL)
```

Arguments

complex	The result of `serialize()`.
st	optionally, the simplex tree to insert into. Otherwise a new one is created.

Details

The serialize/deserialize commands can be used to compress/uncompress the complex into smaller form amenable for e.g. storing on disk (see `base::saveRDS()`) or saving for later use.

See Also

Other serialization methods: `clone()`, `serialize()`

empty_face *Empty faces*

Description

Alias to the empty integer vector (`integer(0L)`). Used to indicate the empty face of the tree.

Usage

```
empty_face
```

Format

An object of class `integer` of length 0.

See Also

traverse

enclosing_radius *Enclosing radius of a set of distances*

Description

Computes the enclosing radius of a set of distances.

Usage

```
enclosing_radius(d)
```

Arguments

d	a `stats::dist()` object.

Details

The enclosing radius is useful as an upper bound of the scale parameter for the rips filtration. Scales above the enclosing radius render the Vietoris–Rips complex as a simplicial cone, beyond which the homology is trivial.

Value

a numeric scalar.

expand *k-expansion*

Description

Performs a k-expansion on the 1-skeleton of the complex, adding k-simplices if all combinations of edges are included. Because this operation uses the edges alone to infer the existence of higher order simplices, the expansion assumes the underlying complex is a flag complex.

Usage

```
expand(st, k = 2)
```

Arguments

st	a simplex tree.
k	the target dimension of the expansion.

Value

the expanded simplex tree `st`, invisibly.

See Also

Other complex-level operations: clear(), contract(), threshold()

---

faces *Face traversal*

Description

Generates a face traversal on the simplex tree.

Usage

```
faces(st, sigma)
```

Arguments

st	the simplex tree to traverse.
sigma	simplex to start the traversal at.

Value

a traversal (object of class "st_traversal").

See Also

Other traversals: coface_roots(), cofaces(), k_simplices(), k_skeleton(), level_order(), link(), maximal(), preorder(), traverse()

---

find *Find simplices*

Description

Returns whether supplied simplices exist in the complex.

Usage

```
find(st, simplices)
```

Arguments

st	a simplex tree.
simplices	simplices to insert, either as a vector, a list of vectors, or a column-matrix. See details.

Details

Traverses the simplex tree looking for `simplices`, returning whether or not it exists. `simplices` can be specified as vector to represent a single simplex, and a list to represent a set of simplices. Each `simplices` is sorted before traversing the trie.

If `simplices` is a vector, it's assumed to be a simplex. If a list, its assumed each element in the list represents a simplex (as vectors). If the simplices to insert are all of the same dimension, you can also optionally use a matrix, where each column is assumed to be a simplex.

Value

a boolean vector indicating whether each simplex in `simplices` exists in `st`.

Usage

st %>% find(simplices)

See Also

Other simplex-level operations: `generate_ids()`, `insert()`, `remove()`

`flag` *Flag complexes*

Description

Creates a filtration of flag complexes

Creates a filtration of flag complexes

Usage

```
flag(st, d)
```

```
flag(st, d)
```

Arguments

`st`	a simplex tree. See details.
`d`	a vector of edge weights, or a 'dist' object.

Details

A flag complex is a simplicial complex whose k-simplices for $k >= 2$ are completely determined by edges/graph of the complex. This function creates filtered simplicial complex using the supplied edge weights. The resulting complex is a simplex tree object endowed with additional structure; see. Vertices have their weights set to 0, and k-simplices w/ $k >= 2$ have their weights set to the maximum weight of any of its edges.

A flag complex is a simplicial complex whose k-simplices for k >= 2 are completely determined by edges/graph of the complex. This function creates filtered simplicial complex using the supplied edge weights. The resulting complex is a simplex tree object endowed with additional structure; see. Vertices have their weights set to 0, and k-simplices w/ k >= 2 have their weights set to the maximum weight of any of its edges.

Value

a simplicial filtration (object of class `"Rcpp_Filtration"`).

See Also

Other simplicial complex constructors: `nerve()`, `rips()`, `simplex_tree()`

`generate_ids` *Generate vertex ids*

Description

Generates vertex ids representing 0-simplices not in the tree.

Usage

```
generate_ids(st, n)
```

Arguments

`st`	a simplex tree.
`n`	the number of ids to generate.

Details

This function generates new vertex ids for use in situations which involve generating new new 0-simplices, e.g. insertions, contractions, collapses, etc. There are two 'policies' which designate the generating mechanism of these ids: 'compressed' and 'unique'. 'compressed' generates vertex ids sequentially, starting at 0. 'unique' tracks an incremental internal counter, which is updated on every call to `generate_ids()`. The new ids under the 'unique' policy generates the first sequential `n` ids that are strictly greater `max(`*counter*`,` *max vertex id*`)`.

Value

a double vector of the `n` smallest natural numbers (starting at `0`) that are not vertex ids of `st`.

See Also

Other simplex-level operations: `find()`, `insert()`, `remove()`

Examples

```
st <- simplex_tree()
print(st$id_policy)
## "compressed"
st %>% generate_ids(3)
## 0 1 2
st %>% generate_ids(3)
## 0 1 2
st %>% insert(list(1,2,3))
print(st$vertices)
## 1 2 3
st %>% insert(as.list(st %>% generate_ids(2)))
st %>% print_simplices()
# 0, 1, 2, 3, 4
st %>% remove(4)
st %>% generate_ids(1)
# 4
```

---

insert *Insert simplices*

Description

Inserts simplices into the simplex tree. Individual simplices are specified as vectors, and a set of simplices as a list of vectors.

Usage

```
insert(st, simplices)
```

Arguments

`st`	a simplex tree.
`simplices`	simplices to insert, either as a vector, a list of vectors, or a column-matrix. See details.

Details

This function allows insertion of arbitrary order simplices. If the simplex already exists in the tree, no insertion is made, and the tree is not modified. `simplices` is sorted before traversing the trie. Faces of `simplices` not in the simplex tree are inserted as needed.

If `simplices` is a vector, it's assumed to be a simplex. If a list, its assumed each element in the list represents a simplex (as vectors). If the simplices to insert are all of the same dimension, you can also optionally use a matrix, where each column is assumed to be a simplex.

Value

the simplex tree `st` with the simplices `simplices` inserted, invisibly.

See Also

Other simplex-level operations: `find()`, `generate_ids()`, `remove()`

Examples

```
st <- simplex_tree()
st %>% insert(1:3) ## inserts the 2-simplex { 1, 2, 3 }
st %>% insert(list(4:5, 6)) ## inserts a 1-simplex { 4, 5 } and a 0-simplex { 6 }.
st %>% insert(combn(5,3)) ## inserts all the 2-faces of a 4-simplex
```

`is_face` *Face test*

Description

Checks whether a simplex is a face of another simplex and is in the complex.

Usage

```
is_face(st, tau, sigma)
```

Arguments

`st`	a simplex tree.
`tau`	a simplex which may contain `sigma` as a coface.
`sigma`	a simplex which may contain `tau` as a face.

Details

A simplex τ is a face of σ if the vertices of τ are vertices of σ. This function checks whether that is true. `tau` and `sigma` are sorted before comparison.

Value

boolean indicating whether `tau` is a face of `sigma`.

See Also

std::includes

Examples

```
st <- simplex_tree()
st %>% insert(1:3)
st %>% is_face(2:3, 1:3)
st %>% is_face(1:3, 2:3)
```

is_tree *Tree (acyclicity) test*

Description

This function performs a breadth-first search on the simplicial complex, checking if the complex is acyclic.

Usage

```
is_tree(st)
```

Arguments

st	a simplex tree.

Value

a boolean indicating whether `st` is acyclic.

Examples

```
st <- simplex_tree()
st %>% insert(list(1:2, 2:3))
st %>% is_tree() # true
st %>% insert(c(1, 3))
st %>% is_tree() # false
```

k_simplices *k-Simplex traversal*

Description

Generates a traversal on the k-simplices of the simplex tree.

Usage

```
k_simplices(st, k, sigma = NULL)
```

Arguments

st	the simplex tree to traverse.
k	the dimension of the skeleton to include.
sigma	simplex to start the traversal at.

Value

a traversal (object of class "st_traversal").

See Also

Other traversals: coface_roots(), cofaces(), faces(), k_skeleton(), level_order(), link(), maximal(), preorder(), traverse()

---

k_skeleton *k-Skeleton traversal*

Description

Generates a k-skeleton traversal on the simplex tree.

Usage

```
k_skeleton(st, k, sigma = NULL)
```

Arguments

st	the simplex tree to traverse.
k	the dimension of the skeleton to include.
sigma	simplex to start the traversal at.

Value

a traversal (object of class "st_traversal").

See Also

Other traversals: coface_roots(), cofaces(), faces(), k_simplices(), level_order(), link(), maximal(), preorder(), traverse()

level_order *Level order traversal*

Description

Generates a level order traversal on the simplex tree.

Usage

```
level_order(st, sigma = NULL)
```

Arguments

st	the simplex tree to traverse.
sigma	simplex to start the traversal at.

Value

a traversal (object of class "st_traversal").

See Also

Other traversals: coface_roots(), cofaces(), faces(), k_simplices(), k_skeleton(), link(), maximal(), preorder(), traverse()

link *Link traversal*

Description

Generates a traversal on the link of a given simplex in the simplex tree.

Usage

```
link(st, sigma)
```

Arguments

st	the simplex tree to traverse.
sigma	simplex to start the traversal at.

Value

a traversal (object of class "st_traversal").

See Also

Other traversals: coface_roots(), cofaces(), faces(), k_simplices(), k_skeleton(), level_order(), maximal(), preorder(), traverse()

`maximal` *Maximal traversal*

Description

Generates a traversal on the maximal of the simplex tree.

Usage

```
maximal(st, sigma = NULL)
```

Arguments

`st`	the simplex tree to traverse.
`sigma`	simplex to start the traversal at.

Value

a traversal (object of class `"st_traversal"`).

See Also

Other traversals: coface_roots(), cofaces(), faces(), k_simplices(), k_skeleton(), level_order(), link(), preorder(), traverse()

`nerve` *Nerve of a cover*

Description

Compute the nerve of a given cover.

Usage

```
nerve(st, cover, k = st$dimension, threshold = 1L, neighborhood = NULL)
```

Arguments

st	a simplex tree.
cover	list of integers indicating set membership. See details.
k	max simplex dimension to consider.
threshold	the number of elements in common for k sets to be considered intersecting. Defaults to 1.
neighborhood	which combinations of sets to check. See details.

Details

This computes the nerve of a given cover, adding a k-simplex for each combination of $k+1$ sets in the given cover that have at least threshold elements in their common intersection.

If neighborhood is supplied, it can be either (1) a matrix, (2) a list, or (3) a function. Each type parameterizes which sets in the cover need be checked for to see if they have at least threshold elements in their common intersection. If a matrix is supplied, the columns should indicate the indices of the cover to check (e.g if neighborhood = matrix(c(1,2), nrow = 2), then only the first two sets of cover are tested.). Similarly, if a list is supplied, each element in the list should give the indices to test.

The most flexible option is supplying a function to neighborhood. If a function is passed, it's assumed to accept an integer vector of k indices (of the cover) and return a boolean indicating whether or not to *test* if they have at least threshold elements in their common intersection. This can be used to filter out subsets of the cover the user knows are The indices are generated using the same code that performs expand().

Value

a simplicial complex (object of class "Rcpp_SimplexTree").

See Also

Other simplicial complex constructors: flag(), rips(), simplex_tree()

---

plot_simplextree *Plots the simplex tree*

Description

Plots the simplex tree

Usage

```
## S3 method for class 'Rcpp_SimplexTree'
plot(
  x,
  coords = NULL,
  vertex_opt = NULL,
  text_opt = NULL,
  edge_opt = NULL,
  polygon_opt = NULL,
  color_pal = NULL,
  maximal = TRUE,
  by_dim = TRUE,
  add = FALSE,
  ...
)

## S3 method for class 'Rcpp_Filtration'
plot(...)
```

Arguments

x	a simplex tree.
coords	Optional (n x 2) matrix of coordinates, where n is the number of 0-simplices.
vertex_opt	Optional parameters to modify default vertex plotting options. Passed to points.
text_opt	Optional parameters to modify default vertex text plotting options. Passed to text.
edge_opt	Optional parameters to modify default edge plotting options. Passed to segments.
polygon_opt	Optional parameters to modify default k-simplex plotting options for k > 1. Passed to polygon.
color_pal	Optional vector of colors. See details.
maximal	Whether to draw only the maximal faces of the complex. Defaults to true.
by_dim	Whether to apply (and recycle or truncate) the color palette to the dimensions rather than to the individual simplices. Defaults to true.
add	Whether to add to the plot or redraw. Defaults to false. See details.
...	unused

Details

This function allows generic plotting of simplicial complexes using base graphics.
If not (x,y) coordinates are supplied via coords, a default layout is generated via phyllotaxis arrangement. This layout is not in general does not optimize the embedding towards any usual visualization criteria e.g. it doesn't try to separate connected components, minimize the number of crossings, etc. For those, the user is recommended to look in existing code graph drawing libraries, e.g. igraphs 'layout.auto' function, etc. The primary benefit of the default phyllotaxis arrangement is that it is deterministic and fast to generate.

All parameters passed via list to `vertex_opt`, `text_opt`, `edge_opt`, `polygon_opt` override default parameters and are passed to `points`, `text`, `segments`, and `polygon`, respectively.

If `add` is true, the plot is not redrawn.

If `maximal` is true, only the maximal simplices are drawn.

The `color_pal` argument controls how the simplicial complex is colored. It can be specified in multiple ways.

1. A vector of colors of length *dim+1*, where *dim*=`x$dimension`
2. A vector of colors of length *n*, where *n*=`sum(x$n_simplices)`
3. A named list of colors

Option (1) assigns every simplex a color based on its dimension.

Option (2) assigns each individual simplex a color. The vector must be specified in level-order (see `ltraverse` or examples below).

Option (3) allows specifying individual simplices to draw. It expects a named list, where the names must correspond to simplices in `x` as comma-separated strings and whose values are colors. If option (3) is specified, this method will *only* draw the simplices given in `color_pal`.

If `length(color_pal)` does not match the dimension or the number of simplices in the complex, the color palette is recyled and simplices are as such.

Functions

- `plot.Rcpp_Filtration`: family of plotting methods.

Examples

```
## Simple 3-simplex
st <- simplex_tree() %>% insert(list(1:4))

## Default is categorical colors w/ diminishing opacity
plot(st)

## If supplied colors have alpha defined, use that
vpal <- rainbow(st$dimension + 1)
plot(st, color_pal = vpal)

## If alpha not supplied, decreasing opacity applied
plot(st, color_pal = substring(vpal, first=1, last=7))

## Bigger example; observe only maximal faces (+vertices and edges) are drawn
st <- simplex_tree(list(1:3, 2:5, 5:9, 7:8, 10))
plot(st, color_pal = rainbow(st$dimension + 1))

## If maximal == FALSE, every simplex is drawn (even on top of each other)
vpal <- rainbow(st$dimension + 1)[c(1,2,5,4,3)]
```

```
pal_alpha <- c(1, 1, 0.2, 0.35, 0.35)
vpal <- sapply(seq_along(vpal), function(i) adjustcolor(vpal[i], alpha.f = pal_alpha[i]))
plot(st, color_pal = vpal, maximal = FALSE)

## You can also color each simplex individually by supplying a vector
## of the same length as the number of simplices.
plot(st, color_pal = sample(rainbow(sum(st$n_simplices))))

## The order is assumed to follow the level order traversal (first 0-simplices, 1-, etc.)
## This example colors simplices on a rainbow gradient based on the sum of their labels
si_sum <- straverse(st %>% level_order, sum)
rbw_pal <- rev(rainbow(50, start=0,end=4/6))
plot(st, color_pal=rbw_pal[cut(si_sum, breaks=50, labels = FALSE)])

## This also makes highlighting simplicial operations fairly trivial
four_cofaces <- as.list(cofaces(st, 4))
coface_pal <- straverse(level_order(st), function(simplex){
    ifelse(list(simplex) %in% four_cofaces, "orange", "blue")
})
plot(st, color_pal=unlist(coface_pal))

## You can also give a named list to draw individual simplices.
## **Only the maximal simplices in the list are drawn**
blue_vertices <- structure(as.list(rep("blue", 5)), names=as.character(seq(5, 9)))
plot(st, color_pal=append(blue_vertices, list("5,6,7,8,9"="red")))
```

`preorder` *Preorder traversal*

Description

Generate a preorder traversal on the simplex tree.

Usage

```
preorder(st, sigma = NULL)
```

Arguments

`st`	the simplex tree to traverse.
`sigma`	simplex to start the traversal at.

Value

a traversal (object of class `"st_traversal"`).

See Also

Other traversals: `coface_roots()`, `cofaces()`, `faces()`, `k_simplices()`, `k_skeleton()`, `level_order()`, `link()`, `maximal()`, `traverse()`

print_simplices *Print simplices to the console*

Description

Prints a traversal, a simplex tree, or a list of simplices to the R console, with options to customize how the simplices are printed. The `format` must be one of "summary", "tree", "cousins", "short", "column", or "row", with the default being "short". In general, the "tree" and "cousins" format give more details on the structure of the trie, whereas the other formats just change how the given set of simplices are formatted.

The "tree" method prints the nodes grouped by the same last label and indexed by depth. The printed format is:

`[vertex] (h = [subtree height]): [subtree depth]([subtree])`

Where each lists the top node (*vertex*) and its corresponding subtree. The *subtree height* displays the highest order k-simplex in that subtree. Each level in the subtree tree is a set of sibling k-simplices whose order is given by the number of dots (`'.'`) proceeding the print level.

The "cousin" format prints the simplex relations used by various algorithms to speed up finding adjacencies in the complex. The cousins are grouped by label and depth.

The format looks like:

`(last=[label], depth=[depth of label]): [simplex]`

This function is useful for understanding how the simplex tree is stored, and for debugging purposes.

Usage

```
print_simplices(
  st,
  format = c("summary", "tree", "cousins", "short", "column", "row")
)
```

Arguments

`st`	a simplex tree.
`format`	the choice of how to format the printing. See details.

reindex *reindexes vertex ids*

Description

This function allows one to 'reorder' or 'reindex' vertex ids. All higher order simplices are renamed accordingly. This operation does not change the structure of the complex; the original and relabeled complexes are isomorphic.

Usage

```
reindex(st, ids)
```

Arguments

st	a simplex tree.
ids	vector of new vertex ids. See details.

Details

The ids parameter must be a sorted integer vector of new ids with length matching the number of vertices. The simplex tree is modified to replace the vertex label at index i with ids[[i]]. See examples.

Examples

```
st <- simplex_tree()
st %>% insert(1:3) %>% print_simplices("tree")
# 1 (h = 2): .( 2 3 )..( 3 )
# 2 (h = 1): .( 3 )
# 3 (h = 0):
st %>% reindex(4:6) %>% print_simplices("tree")
# 4 (h = 2): .( 5 6 )..( 6 )
# 5 (h = 1): .( 6 )
# 6 (h = 0):
```

---

remove *Remove simplices*

Description

Removes simplices from the simplex tree. Individual simplices are specified as vectors, and a set of simplices as a list of vectors.

Usage

```
remove(st, simplices)
```

Arguments

st	a simplex tree.
simplices	simplices to insert, either as a vector, a list of vectors, or a column-matrix. See details.

Details

This function allows removal of a arbitrary order simplices. If `simplices` already exists in the tree, it is removed, otherwise the tree is not modified. `simplices` is sorted before traversing the trie. Cofaces of `simplices` are also removed.

If `simplices` is a vector, it's assumed to be a simplex. If a list, its assumed each element in the list represents a simplex (as vectors). If the simplices to insert are all of the same dimension, you can also optionally use a matrix, where each column is assumed to be a simplex.

Value

the simplex tree `st` with the simplices `simplices` removed, invisibly.

See Also

Other simplex-level operations: `find()`, `generate_ids()`, `insert()`

---

rips *Vietoris–Rips complex*

Description

Constructs the Vietoris–Rips complex.

Usage

```
rips(d, eps = enclosing_radius(d), dim = 1L, filtered = FALSE)
```

Arguments

`d`	a numeric 'dist' vector.
`eps`	diameter parameter.
`dim`	maximum dimension to construct up to. Defaults to 1 (edges only).
`filtered`	whether to construct the filtration. Defaults to false. See details.

Value

a simplicial complex (object of class `"Rcpp_SimplexTree"`).

See Also

Other simplicial complex constructors: `flag()`, `nerve()`, `simplex_tree()`

`sample-abstract` *Sample random abstract simplicial complexes*

Description

Generate random simplicial complexes following the models of Meshulam and Wallach (2009), Kahle (2009), and Costa and Farber (2016).

Usage

```
sample_abstract(
  n,
  prob,
  dimension = NULL,
  method = c("costa_farber", "linial_meshulam_wallach", "kahle", "erdos_renyi")
)
```

Arguments

`n`	an integer number of starting vertices.
`prob`	a numeric simplex insertion probability (Linial-Meshulam-Wallach, Kahle) or a vector of probabilities for all dimensions (Costa-Farber). The dimension of a Costa-Farber random simplicial complex will be at most `length(prob) - 1L`.
`dimension`	an integer dimension at which to randomly insert simplices.
`method`	a character string indicating the model to use; matched to `"erdos_renyi"`, `"kahle"`, `"linial_meshulam_wallach"`, and `"costa_farber"`, allowing for spaces in place of underscores.

Details

The random graph model $G(n, p)$ of Erdős and Rényi (1959) powers parts of other models and is exported for convenience.

The random clique complex model of Kahle (2009) samples an Erdős-Rényi random graph, then uses `[expand()]` to insert all complete subgraphs.

The random simplicial complex model of Costa and Farber (2016) begins with a finite number of vertices n (`n`) and proceeds as follows, based on the $d + 1$-dimensional vector of probabilities $p_0, \ldots, p_d$ (`prob`):

- Delete each vertex with probability $1 - p_0$.
- Insert an edge on each pair of vertices with probability p_1.
- Insert a 2-simplex on each triangle with probability p_2.
- For $k = 3, \ldots, d$, insert a k-simplex on each subcomplex that forms a k-simplex boundary with probability p_k.

The model of Meshulam and Wallach (2009), generalized from that of Linial and Meshulam (2006), is a special case in which $p_k = 1$ for $0 \leq k \leq d - 1$; the only parameters are n (`n`) and p_d (`prob`).

References

Erdős P. and Rényi A. (1959) On Random Graphs I. Publicationes Mathematicae 6: 290–297.

Linial N. and Meshulam R. (2006) Homological Connectivity of Random 2-Complexes. Combinatorica 26, 4, 475–487. doi:10.1007/s00493-006-0027-9

Meshulam, R. and Wallach, N. (2009) Homological Connectivity of Random k-Dimensional Complexes. Random Struct. Alg., 34: 408–417. doi:10.1002/rsa.20238

Kahle, M. (2009) Topology of Random Clique Complexes. Discrete Math., 309(6): 1658–1671. doi:10.1016/j.disc.2008.02.037

Costa A. and Farber M. (2016) Random Simplicial Complexes. In: Callegaro F., Cohen F., De Concini C., Feichtner E., Gaiffi G., Salvetti M. (eds) Configuration Spaces. Springer INdAM Series, vol 14. Springer, Cham. doi:10.1007/978-3-319-31580-5_6

Examples

```
set.seed(1)
## Generate Erdos-Renyi random graphs
plot(sample_abstract(n = 12L, prob = .2, method = "erdos_renyi"))
plot(sample_abstract(n = 12L, prob = .5, method = "erdos_renyi"))
plot(sample_abstract(n = 12L, prob = .8, method = "erdos_renyi"))
## Generate Kahle random clique complexes
sample_abstract(n = 6L, prob = .2, method = "kahle")
sample_abstract(n = 6L, prob = .5, method = "kahle")
sample_abstract(n = 6L, prob = .8, method = "kahle")
## Generate Linial-Meshulam random simplicial complexes
sample_abstract(n = 6L, dimension = 0L, prob = .6,
                method = "linial_meshulam_wallach")
sample_abstract(n = 6L, dimension = 1L, prob = .6,
                method = "linial_meshulam_wallach")
sample_abstract(n = 6L, dimension = 2L, prob = .6,
                method = "linial_meshulam_wallach")
sample_abstract(n = 6L, dimension = 3L, prob = .6,
                method = "linial_meshulam_wallach")
## Generate Costa-Farber random simplicial complexes
plot(sample_abstract(n = 12L, prob = c(.5, .5, .5), method = "costa_farber"))
plot(sample_abstract(n = 12L, prob = c(.5, .5, .5), method = "costa_farber"))
plot(sample_abstract(n = 12L, prob = c(.5, .5, .5), method = "costa_farber"))
## Construct a complete complex of a given size and dimension
sample_abstract(n = 6L, dimension = 4L, prob = 0,
                method = "linial_meshulam_wallach")
sample_abstract(n = 6L, prob = rep(1, 4L), method = "costa_farber")
## Construct the clique complex of a random 1-skeleton
plot(sample_abstract(n = 10L, prob = c(.7, .6, rep(1, 11L)),
                     method = "costa_farber"))
```

`sample-geometric` *Sample random geometric simplicial complexes*

Description

Generate Vietoris–Rips complexes on random point clouds.

Usage

```
make_geometric(
  d,
  radius = NULL,
  dimension = 1L,
  method = c("vietoris_rips"),
  coords = FALSE,
  ...
)

sample_unit(n, torus = FALSE, coords = FALSE)

sample_geometric(
  n,
  torus = FALSE,
  radius = NULL,
  dimension = 1L,
  method = c("vietoris_rips"),
  coords = FALSE,
  ...
)
```

Arguments

`d`	a distance matrix (`"dist"` class) object, or a numeric matrix of (row) coordinates of points (which will be transformed into a distance matrix).
`radius`	a numeric distance within which subsets of points will form simplices.
`dimension`	an integer maximum dimension of simplices to form.
`method`	a character string indicating the model to use; matched only to `"vietoris_rips"`, allowing for spaces in place of underscores, anticipating future additional methods like `"cech"`.
`coords`	a logical instruction to retain the coordinates from a numeric matrix `d` as an attribute of the simplicial complex.
`...`	additional parameters passed to the constructor indicated by `method`.
`n`	an integer number of starting points.
`torus`	a logical instruction to identify opposite faces of the sampling region.

Details

The geometric random graph model (see Penrose, 2003) begins with a random sample of points from a distribution on a manifold (usually Euclidean space), which are taken to be vertices, and introduces edges between vertices within a fixed distance of each other.

The geometric random simplicial complex model extends this model by constructing a Vietoris–Rips or Čech complex on the sample. See Kahle (2011) and Bobrowski and Weinberger (2017) for key results and Kahle (2017) for a review.

Value

A `[simplex_tree()]` (`*_geometric()`) or a `"dist"` object or coordinate matrix (`sample_unit()`).

References

Penrose M. (2003) Random Geometric Graphs. Oxford University Press. doi:10.1093/acprof:oso/9780198506263.001.0001/a 9780198506263

Kahle M. (2011) Random Geometric Complexes. Discrete Comput. Geom. 45, 553–573. doi:10.1007/s00454-010-9319-3

Bobrowski O. and Weinberger S. (2017) On the vanishing of homology in random Čech complexes. Random Struct. Alg., 51: 14–51. doi:10.1002/rsa.20697

Kahle M. (2017) In: J.E. Goodman, J. O'Rourke, and C.D. Tóth (eds) Handbook of Discrete and Computational Geometry, 3rd edition. CRC Press, Boca Raton, FL.

Examples

```
set.seed(1)
## Construct geometric simplicial complexes from a sample point cloud
theta <- stats::runif(n = 24L, min = 0, max = 2*pi)
x <- cbind(x = cos(theta), y = sin(theta))
plot(x)
make_geometric(x, radius = .03, dimension = 2L)
make_geometric(x, radius = .3, dimension = 2L)
## Check distance ranges for square and toroidal samples
sqrt(2)
range(sample_unit(n = 1e3L))
sqrt(2)/2
range(sample_unit(n = 1e3L, torus = TRUE))
## Construct random geometric simplicial complexes, square and toroidal
plot(sample_geometric(24L, radius = .1, dimension = 1L))
plot(sample_geometric(24L, radius = .1, dimension = 1L, torus = TRUE))
plot(sample_geometric(24L, radius = .1, dimension = 2L))
plot(sample_geometric(24L, radius = .1, dimension = 2L, torus = TRUE))
```

`serialize` *Serialize the simplex tree*

Description

Provides a compressed serialization interface for the simplex tree.

Usage

```
serialize(st)
```

Arguments

st	a simplex tree.

Details

The serialize/deserialize commands can be used to compress/uncompress the complex into smaller form amenable for e.g. storing on disk (see base::saveRDS()) or saving for later use. The serialization.

See Also

Other serialization methods: clone(), deserialize()

Examples

```
st <- simplex_tree(list(1:5, 7:9))
st2 <- deserialize(serialize(st))
all.equal(as.list(preorder(st)), as.list(preorder(st2)))
# TRUE

set.seed(1234)
R <- rips(dist(replicate(2, rnorm(100))), eps = pnorm(0.10), dim = 2)
print(R$n_simplices)
# 100 384 851

## Approx. size of the full complex
print(utils::object.size(as.list(preorder(R))), units = "Kb")
# 106.4 Kb

## Approx. size of serialized version
print(utils::object.size(serialize(R)), units = "Kb")
# 5.4 Kb
## You can save these to disk via e.g. saveRDS(serialize(R), ...)
```

---

simplex_tree *Simplex Tree*

Description

Simplex tree class exposed as an Rcpp Module.

Usage

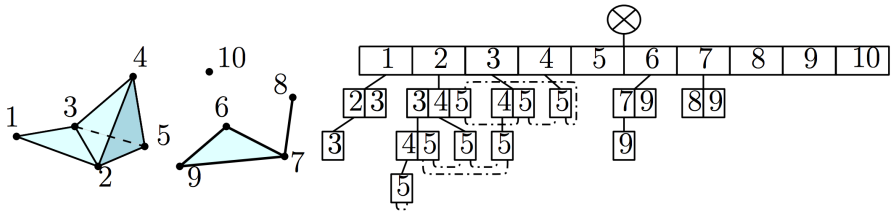
```
simplex_tree(simplices = NULL)
```

Arguments

simplices	optional simplices to initialize the simplex tree with. See insert().

Details

A simplex tree is an ordered trie-like structure specialized for storing and doing general computation simplicial complexes. Here is figure of a simplex tree, taken from the original paper (see Boissonnat and Maria, 2014):



The current implementation provides a subset of the functionality described in the paper.

Value

A queryable simplex tree, as an object (Rcpp module) of class `"Rcpp_SimplexTree"`.

Fields

`n_simplices` A vector, where each index k denotes the number $(k-1)$-simplices.

`dimension` The dimension of the simplicial complex.

Properties

Properties are actively bound shortcuts to various methods of the simplex tree that may be thought of as fields. Unlike fields, however, properties are not explicitly stored: they are generated on access.

`$id_policy` The policy used to generate new vertex ids. May be assigned `"compressed"` or `"unique"`. See `generate_ids()`.

`$vertices` The 0-simplices of the simplicial complex, as a matrix.

`$edges` The 1-simplices of the simplicial complex, as a matrix.

`$triangles` The 2-simplices of the simplicial complex, as a matrix.

`$quads` The 3-simplices of the simplicial complex, as a matrix.

`$connected_components` The connected components of the simplicial complex.

Methods

`$as_XPtr()` Creates an external pointer.

`$generate_ids()` Generates new vertex ids according to the set policy.

`$insert()` Inserts a simplex into the trie.

`$remove()` Removes a simplex from the trie.

`$find()` Returns whether a simplex exists in the trie.

`$degree()` Returns the degree of each given vertex.

`$adjacent()` Returns vertices adjacent to a given vertex.

`$clear()` Clears the simplex tree.

`$expand()` Performs an k-expansion.

`$collapse()` Performs an elementary collapse.

`$contract()` Performs an edge contraction.

`$traverse()` Traverses a subset of the simplex tree, applying a function to each simplex.

`$ltraverse()` Traverses a subset of the simplex tree, applying a function to each simplex and returning the result as a list.

`$is_face()` Checks for faces.

`$is_tree()` Checks if the simplicial complex is a tree.

`$as_list()` Converts the simplicial complex to a list.

`$as_adjacency_matrix()` Converts the 1-skeleton to an adjacency matrix.

`$as_adjacency_list()` Converts the 1-skeleton to an adjacency list.

`$as_edgelist()` Converts the 1-skeleton to an edgelist.

Author(s)

Matt Piekenbrock

References

Boissonnat, Jean-Daniel, and Clement Maria. "The simplex tree: An efficient data structure for general simplicial complexes." Algorithmica 70.3 (2014): 406-427.

See Also

Other simplicial complex constructors: `flag()`, `nerve()`, `rips()`

Examples

```
## Recreating simplex tree from figure.
st <- simplex_tree()
st %>% insert(list(1:3, 2:5, c(6, 7, 9), 7:8, 10))
plot(st)

## Example insertion
st <- simplex_tree(list(1:3, 4:5, 6)) ## Inserts one 2-simplex, one 1-simplex, and one 0-simplex
print(st)
# Simplex Tree with (6, 4, 1) (0, 1, 2)-simplices

## More detailed look at structure
print_simplices(st, "tree")
# 1 (h = 2): .( 2 3 )..( 3 )
# 2 (h = 1): .( 3 )
# 3 (h = 0):
# 4 (h = 1): .( 5 )
# 5 (h = 0):
# 6 (h = 0):
## Print the set of simplices making up the star of the simplex '2'
```

```
print_simplices(st %>% cofaces(2))
# 2, 2 3, 1 2, 1 2 3

## Retrieves list of all simplices in DFS order, starting with the empty face
dfs_list <- ltraverse(st %>% preorder(empty_face), identity)

## Get connected components
print(st$connected_components)
# [1] 1 1 1 4 4 5

## Use clone() to make copies of the complex (don't use the assignment `<-`)
new_st <- st %>% clone()

## Other more internal methods available via `$`
list_of_simplices <- st$as_list()
adj_matrix <- st$as_adjacency_matrix()
# ... see also as_adjacency_list(), as_edge_list(), etc
```

`threshold` *Filtered complex thresholding*

Description

Thresholds a given filtered simplicial complex.

Usage

```
threshold(st, index = NULL, value = NULL)
```

Arguments

`st`	simplex tree.
`index`	integer index to threshold to.
`value`	numeric index to threshold filtration.

Value

the thresholded simplex tree `st`, invisibly.

See Also

Other complex-level operations: `clear()`, `contract()`, `expand()`

traversals *Methods for traversal objects*

Description

Methods for traversal objects

Usage

```
## S3 method for class 'st_traversal'
print(x, ...)

## S3 method for class 'st_traversal'
as.list(x, ...)
```

Arguments

x	traversal object.
...	unused.

traverse *Apply a function along a traversal*

Description

Traverses specific subsets of a simplicial complex.

Usage

```
traverse(traversal, f, ...)

straverse(traversal, f, ...)

ltraverse(traversal, f, ...)
```

Arguments

traversal	The type of traversal to use.
f	An arbitrary function to apply to eac simplex of the traversal. See details.
...	unused.

Details

traverse() allows for traversing ordered subsets of the simplex tree. The specific subset and order are determined by the choice of *traversal*: examples include the preorder() traversal, the cofaces() traversal, etc. See the links below. Each simplex in the traversal is passed as the first and only argument to f, one per simplex in the traversal. traverse() does nothing with the result; if you want to collect the results of applying f to each simplex into a list, use ltraverse() (or straverse()), which are meant to be used like lapply() and sapply(), respectively.

Value

NULL; for list or vector-valued returns, use ltraverse() and straverse() respectively.

See Also

Other traversals: coface_roots(), cofaces(), faces(), k_simplices(), k_skeleton(), level_order(), link(), maximal(), preorder()

Examples

```
## Starter example complex
st <- simplex_tree()
st %>% insert(list(1:3, 2:5))

## Print out complex using depth-first traversal.
st %>% preorder() %>% traverse(print)

## Collect the last labels of each simplex in the tree.
last_labels <- st %>% preorder() %>% straverse(function(simplex){ tail(simplex, 1) })
```

---

union_find *Union-find*

Description

Union find structure exposed as an Rcpp Module.

Usage

```
union_find(n = 0L)
```

Arguments

n	Number of elements in the set.

Value

A disjoint set, as a Rcpp_UnionFind object (Rcpp module).

Methods

`$print.simplextree` S3 method to print a basic summary of the simplex tree.

Author(s)

Matt Piekenbrock

Index