Package 'quadtree'

August 29, 2023

Type Package

Title Region Quadtrees for Spatial Data

Version 0.1.14

Date 2023-08-26

Description Provides functionality for working with raster-like quadtrees (also called "region quadtrees"), which allow for variable-sized cells. The package allows for flexibility in the quadtree creation process. Several functions defining how to split and aggregate cells are provided, and custom functions can be written for both of these processes. In addition, quadtrees can be created using other quadtrees as "templates", so that the new quadtree's structure is identical to the template quadtree. The package also includes functionality for modifying quadtrees, querying values, saving quadtrees to a file, and calculating least-cost paths using the quadtree as a resistance surface.

License MIT + file LICENSE

URL https://github.com/dfriend21/quadtree/, https://dfriend21.github.io/quadtree/

BugReports https://github.com/dfriend21/quadtree/issues/

Depends R (>= 2.10)

Imports graphics, grDevices, methods, Rcpp (>= 1.0.5), terra, stats

Suggests raster, sf, knitr, rmarkdown, testthat (>= 3.0.0)

LinkingTo Rcpp

VignetteBuilder knitr

Config/testthat/edition 3

Encoding UTF-8

NeedsCompilation yes

RoxygenNote 7.2.3

Collate 'CppLcpFinder-class.R' 'CppNode-class.R' 'CppQuadtree-class.R' 'classes.R' 'generics.R' 'as_data_frame.R' 'as_foreign.R' 'as_raster.R' 'as_vector.R' 'copy.R' 'extent.R' 'extract.R' 'get_neighbors.R' 'lcp.R' 'n_cells.R' 'plot_LcpFinder.R' 'plot_Quadtree.R' 'projection.R' 'qtree-exports.R' 'quadtree-package.R' 'quadtree.R' 'read_write.R' 'set_values.R' 'summary_LcpFinder.R' 'summary_Quadtree.R' 'transform_values.R'

Author Derek Friend [aut, cre, cph] (<https://orcid.org/0000-0002-6909-8769>), Andrew Brown [ctb], Randolph Voorhies [cph] (Author of included 'cereal' library), Shane Grant [cph] (Author of included 'cereal' library), Juan Pedro Bolivar Puente [cph] (Author of included 'cereal' library)

Maintainer Derek Friend <dafriend.R@gmail.com>

Repository CRAN

Date/Publication 2023-08-29 09:10:02 UTC

R topics documented:

`quadtree-package` *Quadtree Representation of Rasters*

Description

This package provides functionality for working with raster-like quadtrees (also called "region quadtrees"), which allow for variable-sized cells. The package allows for flexibility in the quadtree creation process. Several functions defining how to split and aggregate cells are provided, and custom functions can be written for both of these processes. In addition, quadtrees can be created using other quadtrees as "templates", so that the new quadtree's structure is identical to the template quadtree. The package also includes functionality for modifying quadtrees, querying values, saving quadtrees to a file, and calculating least-cost paths using the quadtree as a resistance surface.

Vignettes are included that demonstrate the functionality contained in the package - these are intended to serve as an introduction to using the `quadtree` package. You can see the available vignettes by running `vignette(package = "quadtree")` and view individual vignettes using `vignette("vignette-name", package = "quadtree")`.

I'd recommend reading the vignettes in the following order:

1. `"quadtree-creation"`
2. `"quadtree-usage"`
3. `"quadtree-lcp"`

A fourth vignette called "quadtree-code" is also available. This briefly discusses the structure of the package. It is not necessary for using the package but may be useful for those who want more details about the code.

`add_legend` *Add a gradient legend to a plot*

Description

Adds a gradient legend to a plot.

Usage

```
add_legend(
  zlim,
  col,
  alpha = 1,
  lgd_box_col = NULL,
  lgd_x_pct = 0.5,
  lgd_y_pct = 0.5,
  lgd_wd_pct = 0.5,
  lgd_ht_pct = 0.5,
  bar_box_col = "black",
  bar_wd_pct = 0.2,
  bar_ht_pct = 1,
  text_cex = 1,
  text_col = NULL,
  text_font = NULL,
  text_x_pct = 1,
  ticks = NULL,
  ticks_n = 5
)
```

Arguments

zlim	two-element numeric vector; required; the min and max value of z
col	character vector; required; the colors that will be used in the legend.
alpha	numeric; transparency of the colors. Must be in the range 0-1, where 0 is fully transparent and 1 is fully opaque. Default is 1.
lgd_box_col	character; color of the box to draw around the entire legend. If NULL (the default), no box is drawn
lgd_x_pct	numeric; location of the center of the legend in the x-dimension, as a fraction (0 to 1) of the *right margin area*, **not** the entire width of the figure
lgd_y_pct	numeric; location of the center of the legend in the y-dimension, as a fraction (0 to 1). Unlike lgd_x_pct, this **is** relative to the entire figure height (since the right margin area spans the entire vertical dimension)
lgd_wd_pct	numeric; width of the entire legend, as a fraction (0 to 1) of the right margin width
lgd_ht_pct	numeric; height of the entire legend, as a fraction (0 to 1) of the figure height
bar_box_col	character; color of the box to draw around the color bar. If NULL, no box is drawn
bar_wd_pct	numeric; width of the color bar, as a fraction (0 to 1) of the width of the *legend area* (**not** the entire right margin width)
bar_ht_pct	numeric; height of the color bar, as a fraction (0 to 1) of the height of the *legend area* (**not** the entire right margin height)
text_cex	numeric; size of the legend text. Default is 1.
text_col	character; color of the legend text. Default is "black".

text_font	integer; specifies which font to use. See par() for more details.
text_x_pct	numeric; the x-placement of the legend text as a fraction (0 to 1) of the width of the legend area. This corresponds to the *right-most* part of the text - i.e. a value of 1 means the text will end exactly at the right border of the legend area. Default is 1.
ticks	numeric vector; the z-values at which to place tick marks. If NULL (the default), tick placement is automatically calculated
ticks_n	integer; the number of ticks desired - only used if ticks is NULL. Note that this is an *approximate* number - the pretty() function is used to generate "nice-looking" values, but it doesn't guarantee a set number of tick marks

Details

I took an HTML/CSS-like approach to determining the positioning - that is, each space is treated as -like space, and the position of objects within that space happens *relative to that space* rather than the entire space. The parameters prefixed by lgd are all relative to the right margin space and correspond to the box that contains the entire legend. The parameters prefixed by bar and ticks are relative to the space within the legend box.

This function is used within plot(), so the user shouldn't call this function to manually create the legend. Customizations to the legend can be done via the legend_args parameter of plot(). Using this function to plot the legend after using plot() raises the possibility of the legend not corresponding correctly with the plot, and thus should be avoided.

Value

no return value

Examples

```
library(terra)
library(quadtree)
habitat <- terra::rast(system.file("extdata", "habitat.tif", package="quadtree"))
qt <- quadtree(habitat, .2)

old_par <- par(mar = c(5, 4, 4, 5))
plot(qt, legend = FALSE)
leg <- terra::minmax(habitat)[1:2]
quadtree::add_legend(leg, rev(terrain.colors(100)))
par(old_par)
# this example simply illustrates how it COULD be used, but as stated in the
# 'Details' section, it shouldn't be called separately from 'plot()' - if
# customizations to the legend are desired, use the 'legend_args' parameter
# of 'plot()'.
```

as_data_frame *Convert a* Quadtree *to a data frame*

Description

Creates a data frame with information on each quadtree cell.

Usage

```
## S4 method for signature 'Quadtree'
as_data_frame(x, terminal_only = TRUE)
```

Arguments

x	a Quadtree
terminal_only	boolean; if TRUE (the default) only information on terminal cells is returned. If FALSE, information on all cells is returned.

Value

A data frame with one row for each quadtree cell. The columns are as follows:

- id: the id of the cell
- hasChildren: 1 if the cell has children, 0 otherwise
- level: integer; the depth of this cell/node in the quadtree, where the root of the quadtree is considered to be level 0
- xmin, xmax, ymin, ymax: the x and y limits of the cell
- value: the value of the cell
- smallestChildLength: the smallest cell length among all of this cell's descendants
- parentID: the ID of the cell's parent. The root, which has no parent, has a value of -1 for this column

See Also

as_vector() returns all the cell values as a numeric vector.

Examples

```
library(quadtree)

mat <- rbind(c(1, 1, 0, 1),
             c(1, 1, 1, 0),
             c(1, 0, 1, 1),
             c(0, 1, 1, 1))
qt <- quadtree(mat, .1)
plot(qt)
as_data_frame(qt)
```

as_raster *Create a raster from a* Quadtree

Description

Creates a SpatRaster from a Quadtree.

Usage

```
## S4 method for signature 'Quadtree'
as_raster(x, rast = NULL)
```

Arguments

x	a Quadtree
rast	a SpatRaster or RasterLayer; optional; this will be used as a template - the output raster will have the same extent and dimensions as this raster. If NULL (the default), a raster is automatically created, where the quadtree extent is used as the raster extent, and the size of smallest cell in the quadtree is used as the resolution of the raster.

Details

Note that the value of a raster cell is determined by the value of the quadtree cell located at the centroid of the raster cell - thus, if a raster cell overlaps several quadtree cells, whichever quadtree cell the centroid of the raster cell falls in will determine the raster cell's value. If no value is provided for the rast parameter, the raster's dimensions are automatically determined from the quadtree in such a way that the cells are guaranteed to line up with the quadtree cells with no overlap, thus avoiding the issue.

Value

a SpatRaster

Examples

```
library(quadtree)
habitat <- terra::rast(system.file("extdata", "habitat.tif", package="quadtree"))

# create a quadtree
qt <- quadtree(habitat, split_threshold = .1, split_method = "sd")

rst1 <- as_raster(qt) # use the default raster
rst2 <- as_raster(qt, habitat) # use another raster as a template

old_par <- par(mfrow = c(2, 2))
plot(habitat, main = "original raster")
plot(qt, main = "quadtree")
```

```
plot(rst1, main = "raster from quadtree")
plot(rst2, main = "raster from quadtree")
par(old_par)
```

`as_sf` *Convert to other R spatial objects*

Description

Convert to other R spatial objects

Usage

```
as_sf(x)

as_vect(x)

as_character(x)
```

Arguments

`x`	Quadtree object

Value

an object of class `sf` or `SpatVector`, or a Well-Known Text (WKT) `character` representation

Examples

```
library(quadtree)
habitat <- terra::rast(system.file("extdata", "habitat.tif", package="quadtree"))

qt <- quadtree(habitat, .1)
sf <- as(qt, "sf")
sr <- as(qt, "SpatRaster")
sv <- as(qt, "SpatVector")
ch <- as(qt, "character")
```

as_vector *Get all* Quadtree *cell values as a vector*

Description

Returns all cell values of a Quadtree as a numeric vector.

Usage

```
## S4 method for signature 'Quadtree'
as_vector(x, terminal_only = TRUE)
```

Arguments

x	a Quadtree
terminal_only	boolean; if TRUE (the default) only values of terminal cells are returned. If FALSE, all cell values are returned.

Value

a numeric vector

See Also

as_data_frame creates a data frame from a Quadtree that has all the cell values as well as details about each cell's size and extent.

Examples

```
library(quadtree)
habitat <- terra::rast(system.file("extdata", "habitat.tif", package="quadtree"))

qt <- quadtree(habitat, .2)
head(as_vector(qt), 20)
head(as_vector(qt, FALSE), 20)
```

copy *Create a deep copy of a* Quadtree

Description

Creates a *deep* copy of a Quadtree.

Usage

```
## S4 method for signature 'Quadtree'
copy(x)
```

Arguments

x	a Quadtree

Details

This function creates a *deep* copy of a Quadtree object. The Quadtree class contains a pointer to a CppQuadtree C++ object. If a copy is attempted by simply assigning the quadtree to a new variable, it will simply make a copy of the *pointer*, and both variables will point to the same CppQuadtree. Thus, changes made to one will also change the other. See "Examples" for a demonstration of this.

This function creates a deep copy by copying the entire quadtree, and should be used whenever a copy of a quadtree is desired.

Value

a Quadtree

Examples

```
library(quadtree)
habitat <- terra::rast(system.file("extdata", "habitat.tif", package="quadtree"))

# create a quadtree, then create a shallow copy and a deep copy
qt1 <- quadtree(habitat, split_threshold = .1)
plot(qt1)

qt2 <- qt1 # SHALLOW copy
qt3 <- copy(qt1) # DEEP copy

# change the values of qt1 so we can observe how this affects qt2 and qt3
transform_values(qt1, function(x) 1 - x)

# plot it out to see what happened
old_par <- par(mfrow = c(1, 3))
plot(qt1, main = "qt1", border_col = "transparent")
plot(qt2, main = "qt2", border_col = "transparent")
plot(qt3, main = "qt3", border_col = "transparent")
par(old_par)
# qt2 was modified but qt3 was not
```

CppLcpFinder-class CppLcpFinder: *C++ LCP finder*

Description

CppLcpFinder is a C++ class for finding least-cost paths (LCPs) using a Quadtree as a resistance surface. The average user should not need to interact with this class at all - all of the LCP functionality is made available through the LcpFinder S4 class.

Details

This class is defined in 'src/LcpFinderWrapper.h' and 'src/LcpFinderWrapper.cpp'. When made available to R, it is exposed as `CppLcpFinder` rather than `LcpFinderWrapper`. `LcpFinderWrapper` contains a pointer to a `LcpFinder` C++ object (defined in 'src/LcpFinder.h' and 'src/LcpFinder.cpp'). All of the core functionality is in the `LcpFinder` C++ class. `LcpFinderWrapper` is a wrapper class that adds the 'Rcpp' code required for it to be accessible from R.

Note that there is no constructor made accessible to R - a `CppLcpFinder` is created by using the `getLcpFinder` method of the `CppQuadtree` class.

Fields

`getAllPathsSummary` • **Description**: Returns a matrix summarizing all the LCPs calculated so far. `summarize_lcps()` is a wrapper for this function - see documentation of that function for more details.

- **Parameters**: none
- **Returns**: a matrix with one row per LCP. See documentation of `summarize_lcps()` for details.

`getLcp` • **Description**: Finds the LCP from the starting point to another point. `find_lcp` is a wrapper for this function - see its documentation for more details.

- **Parameters**:
  - `endPoint`: two-element numeric vector (x,y) - the point to find a shortest path to
- **Returns**: A matrix representing the least-cost path. See `find_lcp()` for details on the return matrix.

`getSearchLimits` • **Description**: Returns the x and y limits of the search area.

- **Parameters**: none
- **Returns**: four-element numeric vector, in this order: xmin, xmax, ymin, ymax

`getStartPoint` • **Description**: Returns the start point

- **Parameters**: none
- **Returns**: two-element numeric vector (x,y)

`makeNetworkAll` • **Description**: Calculates LCPs to all cells in the search area. This is used by `find_lcps` when `limit` is `NULL`. See documentation of that function for more details.

- **Parameters**: none
- **Returns**: void - no return value. Specific paths can be retrieved using `getLcp`, and `getAllPathsSummary` can be used to summarize all paths that have been found.

`makeNetworkCostDist` • **Description**: Calculates all LCPs whose cost-distance is less than a given threshold. This is used in `find_lcps` when `limit` is not `NULL`. See documentation of that function for more details.

- **Parameters**:
  - `constraint`: double; the maximum cost-distance allowed for a LCP
- **Returns**: void - no return value. Specific paths can be retrieved using `getLcp`, and `getAllPathsSummary` can be used to summarize all paths that have been found.

CppNode-class CppNode: *C++ quadtree node*

Description

The CppNode C++ class defines objects that represent a single node of a quadtree. This is used internally - end users should have no need to use any of the methods listed here.

Details

This class is defined in 'src/NodeWrapper.h' and 'src/NodeWrapper.cpp'. When made available to R, it is exposed as CppNode instead of NodeWrapper. NodeWrapper contains a pointer to a Node object (defined in 'src/Node.h' and 'src/Node.cpp'). All of the core functionality is in the Node class - NodeWrapper is a wrapper class that adds the 'Rcpp' code required for it to be accessible from R.

Fields

asVector
- **Description**: Returns a vector giving info about the node
- **Parameters**: none
- **Returns**: a numeric vector with the following named elements:
  - id
  - hasChidren
  - level
  - xmin
  - xmax
  - ymin
  - ymax
  - smallestChildLength

  as_data_frame makes use of this function to output info on each node - see the documentation of that function for details on what each column represents

getChildren
- **Description**: Returns a list of the child nodes
- **Parameters**: none
- **Returns**: a list of CppNode objects

getNeighborIds
- **Description**: Returns the IDs of the neighboring cells
- **Parameters**: none
- **Returns**: a numeric vector containing the neighbor IDs

getNeighborInfo
- **Description**: Returns a matrix with info on each of the neighboring cells
- **Parameters**: none
- **Returns**: a matrix. The getNeighborList() member function of CppQuadtree makes use of this function - see documentation of that function for details on the return matrix.

getNeighborVals
- **Description**: Returns the values of all neighboring cells
- **Parameters**: none
- **Returns**: a numeric vector

`getNeighbors`
- **Description**: Returns a list of the neighboring nodes
- **Parameters**: none
- **Returns**: a list of `CppNode` objects

`hasChildren`
- **Description**: Returns a boolean representing whether the node has children
- **Parameters**: none
- **Returns**: a boolean value - `TRUE` if it has children, `FALSE` otherwise

`id`
- **Description**: Returns the ID of this node
- **Parameters**: none
- **Returns**: an integer

`level`
- **Description**: Returns the 'level' (i.e. depth in the tree) of this node
- **Parameters**: none
- **Returns**: an integer

`smallestChildSideLength`
- **Description**: Returns the side length of the smallest descendant node
- **Parameters**: none
- **Returns**: a double

`value`
- **Description**: Returns the value of the node
- **Parameters**: none
- **Returns**: a double

`xLims`
- **Description**: Returns the x boundaries of the node
- **Parameters**: none
- **Returns**: two-element numeric vector (xmin, xmax)

`yLims`
- **Description**: Returns the y boundaries of the node
- **Parameters**: none
- **Returns**: two-element numeric vector (ymin, ymax)

`CppQuadtree-class` `CppQuadtree`*: C++ quadtree data structure*

Description

The `CppQuadtree` class is the underlying C++ data structure used by the `Quadtree` S4 class. Note that the average user should not need to use these functions - there are R wrapper functions that provide access to the many of the member functions.

Details

This class is defined in 'src/QuadtreeWrapper.h' and 'src/QuadtreeWrapper.cpp'. When made available to R, it is exposed as `CppQuadtree` rather than `QuadtreeWrapper`. `QuadtreeWrapper` contains a pointer to a `Quadtree` C++ object (defined in 'src/Quadtree.h' and 'src/Quadtree.cpp'). All of the core functionality is in the `Quadtree` C++ class - `QuadtreeWrapper` is a wrapper class that adds the 'Rcpp' code required for it to be accessible from R.

Fields

`constructor` • **Description**: Default constructor. Can be used as follows: `qt <- new(CppQuadtree)`
- **Parameters**: none
- **Returns**: an empty `CppQuadtree` object

`constructor` • **Description**: Constructor. Can be used as follows: `qt <- new(CppQuadtree, xlims, ylims, maxCellLength, minCellLength, splitAllNAs, splitAnyNAs)`. Used in `quadtree()`. The parameters for this constructor correspond with the similarly named parameters in `quadtree()` - see its documentation for more details on what the parameters signify. Note that the constructor does not "build" the quadtree structure - that is done by `createTree()`.
- **Parameters**:
  - `xlims`: two-element numeric vector (xmin, xmax)
  - `ylims`: two-element numeric vector (ymin, ymax)
  - `maxCellLength`: two-element numeric vector - first element is for the x dimension, second is for the y dimension
  - `minCellLength`: two-element numeric vector - first element is for the x dimension, second is for the y dimension
  - `splitAllNAs`: boolean
  - `splitAnyNAs`: boolean

`readQuadtree` • **Description**: Reads a quadtree from a file. Note that this is a static function, so it does not require an instance of `CppQuadtree` to be called. `read_quadtree()` is a wrapper for this function - see its documentation for more details.
- **Parameters**:
  - `filePath`: string; the file to read from
- **Returns**: a `CppQuadtree`

`asList` • **Description**: Outputs a list containing details about each cell. `as_data_frame()` is a wrapper for this function that `rbind`s the individual list elements into a data frame.
- **Parameters**: none
- **Returns**: a list of named numeric vectors. Each numeric vector provides information on a single cell. The elements returned are the same as the columns described in the documentation for `as_data_frame()` - see that help page for details.

`asVector` • **Description**: Returns cell values as a vector. `as_vector()` is a wrapper for this function.
- **Parameters**:
  - `terminalOnly`: boolean; if `TRUE`, returns only the values of the terminal cells. If `FALSE`, returns all cell values
- **Returns**: a numeric vector

`copy` • **Description**: Returns a deep copy of a quadtree. `copy()` is a wrapper for this function - see the documentation for that function for more details.
- **Parameters**: none
- **Returns**: a `CppQuadtree` object

`createTree` • **Description**: Constructs a quadtree from a matrix. `quadtree()` is a wrapper for this function and should be used to create quadtrees. The parameters correspond with the similarly named parameters in `quadtree()` - see the documentation of that function for details on the parameters.

- **Parameters**:
  - `mat`: matrix; data to be used to create the quadtree
  - `splitMethod`: string
  - `splitThreshold`: double
  - `splitFun`: function
  - `splitArgs`: list
  - `combineFun`: function
  - `combineArgs`: list
  - `templateQuadtree`: `CppQuadtree` object
- **Returns**: void - no return value

`extent`
- **Description**: Returns the extent of the quadtree. This is equivalent to `extent(qt, original = FALSE)`
- **Parameters**: none
- **Returns**: four-element numeric vector, in this order: xmin, xmax, ymin, ymax

`getCell`
- **Description**: Given the x and y coordinates of a point, returns the cell at that point.
- **Parameters**:
  - `pt`: two-element numeric vector (x,y)
- **Returns**: a `CppNode` object representing the cell that contains the point

`getCells`
- **Description**: Given x and y coordinates of points, returns a list of the cells at those points (as `CppNode` objects). It is the same as `getCell`, except that it allows users to get multiple cells at once instead of one at a time.
- **Parameters**:
  - `x`: numeric vector; the x coordinates
  - `y`: numeric vector; the y coordinates; must be the same length as `x`
- **Returns**: a list of `CppNode` objects corresponding to the x and y coordinates passed to the function

`getCellsDetails`
- **Description**: Given points defined by their x and y coordinates, returns a matrix giving details on the cells at each of the points. `extract(qt, extents = TRUE)` is a wrapper for this function.
- **Parameters**:
  - `x`: numeric vector; the x coordinates
  - `y`: numeric vector; the y coordinates; must be the same length as x
- **Returns**: A matrix with the cell details. See `extract()` for details about the matrix columns

`getLcpFinder`
- **Description**: Returns a `CppLcpFinder` object that can be used to find least-cost paths on the quadtree. `lcp_finder()` is a wrapper for this function. For details on the parameters see the documentation of the similarly named parameters in `lcp_finder()`.
- **Parameters**:
  - `startPoint`: two-element numeric vector
  - `xlim`: two-element numeric vector
  - `ylim`: two-element numeric vector
  - `searchByCentroid` : boolean
- **Returns**: a `CppLcpFinder` object

`getNeighborList`
- **Description**: Returns the neighbor relationships between all cells.
- **Parameters**: none
- **Returns**: a list of matrices. Each matrix corresponds to a single cell and has one line for each neighboring cell. "neighbor" includes diagonal adjacency. Each matrix has the following columns:
  - `id0`, `x0`, `y0`, `val0`, `hasChildren0`: the ID, x and y coordinates of the centroid, cell value, and whether the cell has children. This is for the cell of interest. Note that the values of these columns will be same across all rows because they refer to the same cell.
  - `id1`, `x1`, `y1`, `val1`, `hasChildren1`: the ID, x and y coordinates of the centroid, cell value, and whether the cell has children. This is for the neighbors of the cell of interest. (i.e. the cell represented by the columns suffixed with '0').

`getNeighbors`
- **Description**: Given a point, returns a matrix with info on the cells that neighbor the cell that the point falls in. `get_neighbors()` is a wrapper for this function.
- **Parameters**:
  - `pt`: two-element numeric vector (x,y)
- **Returns**: a six-column matrix with one row per neighboring cell. It has the following columns:
  - `id`
  - `xmin`
  - `xmax`
  - `ymin`
  - `ymax`
  - `value`

`getValues`
- **Description**: Given points defined by their x and y coordinates, returns a numeric vector of the values of the cells at each of the points. `extract(qt, extents = FALSE)` is a wrapper for this function.
- **Parameters**:
  - `x`: numeric vector; the x coordinates
  - `y`: numeric vector; the y coordinates; must be the same length as `x`
- **Returns**: a numeric vector of cell values corresponding with the x and y coordinates passed to the function

`maxCellDims`
- **Description**: Returns the maximum allowable cell length used when constructing the quadtree (i.e. the value passed to the `max_cell_length` parameter of `quadtree()`). Note that this does **not** return the maximum cell size in the quadtree - it returns the maximum *allowable* cell size. Also note that if no value was provided for `max_cell_length`, the max allowable cell length is set to the length and width of the total extent.
- **Parameters**: none
- **Returns**: A two-element numeric vector giving the maximum allowable side length for the x and y dimensions.

`minCellDims`
- **Description**: Returns the minimum allowable cell length used when constructing the quadtree (i.e. the value passed to the `min_cell_length` parameter of `quadtree()`). Note that this does **not** return the minimum cell size in the quadtree - it returns the minimum *allowable* cell size. Also note that if no value was provided for `min_cell_length`, the min allowable cell length is set to -1.

- **Parameters**: none
- **Returns**: A two-element numeric vector giving the minimum allowable side length for the x and y dimensions.

`nNodes`
- **Description**: Returns the total number of nodes in the quadtree. Note that this includes *all* nodes, not just terminal nodes.
- **Parameters**: none
- **Returns**: integer

`originalDim`
- **Description**: Returns the dimensions of the raster used to create the quadtree *before* its dimensions were adjusted.
- **Parameters**: none
- **Returns**: two-element numeric vector that gives the number of cells along the x and y dimensions.

`originalExtent`
- **Description**: Returns the extent of the raster used to create the quadtree *before* its dimensions/extent were adjusted. This is equivalent to `extent(qt, original = TRUE)`
- **Parameters**: none
- **Returns**: four-element numeric vector, in this order: xmin, xmax, ymin, ymax

`originalRes`
- **Description**: Returns the resolution of the raster used to create the quadtree *before* its dimensions/extent were adjusted.
- **Parameters**: none
- **Returns**: two-element numeric vector (x cell length, y cell length)

`print`
- **Description**: Returns a string that represents the quadtree.
- **Parameters**: none
- **Returns**: a string

`projection`
- **Description**: Returns the projection of the quadtree.
- **Parameters**: none
- **Returns**: a string

`root`
- **Description**: Returns the root node of the quadtree.
- **Parameters**: none
- **Returns**: a `CppNode` object

`setOriginalValues`
- **Description**: Sets the properties that record the extent and dimensions of the original raster used to create the quadtree
- **Parameters**:
  - `xmin`: double
  - `xmax`: double
  - `ymin`: double
  - `ymax`: double
  - `nX`: integer - number of cells along the x dimension
  - `nY`: integer - number of cells along the y dimension
- **Returns**: void - no return value

`setProjection`
- **Description**: Sets the the projection of the quadtree.
- **Parameters**:

  - projection: string
- **Returns**: void - no return value

`setValues`
- **Description**: Given points defined by their x and y coordinates and a vector of values, sets the values of the quadtree cells at each of the points. `set_values()` is a wrapper for this function - see its documentation page for more details.
- **Parameters**:
  - x: numeric vector; the x coordinates
  - y: numeric vector; the y coordinates; must be the same length as x
  - newVals: numeric vector; must be the same length as x and y
- **Returns**: void - no return value

`transformValues`
- **Description**: Uses a function to transform the values of all cells. `transform_values()` is a wrapper for this function - see its documentation page for more details.
- **Parameters**:
  - trasform_fun: function
- **Returns**: void - no return value

`writeQuadtree`
- **Description**: Writes a quadtree to a file. `write_quadtree()` is a wrapper for this function - see its documentation page for more details.
- **Parameters**:
  - filePath: string; the file to save the quadtree to
- **Returns**: void - no return value

`extent` *Get the extent of a* `Quadtree`

Description

Gets the extent of the `Quadtree` as an `Extent` object (from the raster package).

Usage

```
## S4 method for signature 'Quadtree'
extent(x, original = FALSE)
```

Arguments

x	a `Quadtree`
original	boolean; if `FALSE` (the default), it returns the total extent covered by the quadtree. If `TRUE`, the function returns the extent of the original raster used to create the quadtree, before the dimensions were adjusted by padding with `NA`s and/or the raster was resampled.

Value

an `Extent` object

Examples

```
library(quadtree)
habitat <- terra::rast(system.file("extdata", "habitat.tif", package="quadtree"))

# create a quadtree
qt <- quadtree(habitat, split_threshold = .1, adj_type = "expand")

# retrieve the extent and the original extent
ext <- extent(qt)
ext_orig <- extent(qt, original = TRUE)

ext
ext_orig

# plot them
plot(qt)
rect(ext[1], ext[3], ext[2], ext[4], border = "blue", lwd = 4)
rect(ext_orig[1], ext_orig[3], ext_orig[2], ext_orig[4],
     border = "red", lwd = 4)
```

---

extract *Extract* Quadtree *values*

Description

Extracts the cell values and optionally the cell extents at the given points.

Usage

```
## S4 method for signature 'Quadtree,ANY'
extract(x, y, extents = FALSE)
```

Arguments

x	a Quadtree
y	a two-column matrix representing point coordinates. First column contains the x-coordinates, second column contains the y-coordinates
extents	boolean; if FALSE (the default), a vector containing cell values is returned. If TRUE, a matrix is returned providing each cell's extent in addition to its value

Value

If extents = FALSE, returns a numeric vector corresponding to the values at the points represented by pts.

If extents = TRUE, returns a six-column numeric matrix providing the extent of each cell along with the cell's value and ID. The six columns are, in this order: id, xmin, xmax, ymin, ymax, value.

Examples

```
library(quadtree)
habitat <- terra::rast(system.file("extdata", "habitat.tif", package="quadtree"))

# create quadtree
qt1 <- quadtree(habitat, split_threshold = .1, adj_type = "expand")
plot(qt1)

# create points at which we'll extract values
coords <- seq(-1000, 40010, length.out = 10)
pts <- cbind(coords,coords)

# extract the cell values
vals <- extract(qt1, pts)

# plot the quadtree and the points
plot(qt1, border_col = "gray50", border_lwd = .4)
points(pts, pch = 16, cex = .6)
text(pts, labels = round(vals, 2), pos = 4)

# we can also extract the cell extents in addition to the values
extract(qt1, pts, extents = TRUE)
```

---

find_lcp *Find the LCP between two points on a* Quadtree

Description

Finds the least-cost path (LCP) from the start point (the point used to create the LcpFinder) to another point, using a Quadtree as a resistance surface.

Usage

```
## S4 method for signature 'Quadtree'
find_lcp(
  x,
  start_point,
  end_point,
  use_orig_points = TRUE,
  xlim = NULL,
  ylim = NULL,
  search_by_centroid = FALSE
)

## S4 method for signature 'LcpFinder'
find_lcp(x, end_point, allow_same_cell_path = FALSE)
```

Arguments

x	a LcpFinder or a Quadtree
start_point	two-element numeric vector; the x and y coordinates of the starting point. Not used if x is a LcpFinder since the start point is determined when the LcpFinder is created (using lcp_finder()).
end_point	two-element numeric vector; the x and y coordinates of the destination point
use_orig_points	boolean; if TRUE (the default), the path is calculated between start_point and end_point. If FALSE, the path is calculated between the centroids of the cells the points fall in.
xlim	two-element numeric vector (xmin, xmax); passed to lcp_finder(); constrains the nodes included in the network to those whose x limits fall in the range specified in xlim. If NULL the x limits of x are used
ylim	same as xlim, but for y
search_by_centroid	boolean; passed to lcp_finder(); determines which cells are considered to be "in" the box specified by xlim and ylim. If FALSE (the default) any cell that overlaps with the box is included. If TRUE, a cell is only included if its **centroid** falls inside the box.
allow_same_cell_path	boolean; default is FALSE; if TRUE, allows paths to be found between two points that fall in the same cell. See 'Details' for more.

Details

See the vignette 'quadtree-lcp' for more details and examples (i.e. run `vignette("quadtree-lcp", package = "quadtree")`)

Using `find_lcp(<Quadtree>)` rather than `find_lcp(<LcpFinder>)` is simply a matter of convenience - when a Quadtree is passed to `find_lcp()`, it automatically creates an LcpFinder and then uses `find_lcp(<LcpFinder>)` to get the path between the two points. This is convenient if you only want a single LCP. However, if you want to find multiple LCPs from a single start point, it is better to first create the LcpFinder object using `lcp_finder()` and then use `find_lcp(<LcpFinder>)` for finding LCPs. This is because an LcpFinder object saves state, so subsequent calls to `find_lcp(<LcpFinder>)` will run faster.

By default, if the end point falls in the same cell as the start point, the path will consist only of the point associated with the cell. When using `find_lcp` with a LcpFinder, setting `allow_same_cell_path` to TRUE allows for paths to be found within a single cell. In this case, if the start and end points fall in the same cell, the path will consist of two points - the point associated with the cell and `end_point`. If using `find_lcp` with a Quadtree, this will automatically be allowed if `use_orig_points` is TRUE.

Value

Returns a five column matrix representing the LCP. It has the following columns:

- x: x coordinate of this point (centroid of the cell)

- `y`: y coordinate of this point (centroid of the cell)
- `cost_tot`: the cumulative cost up to this point
- `dist_tot`: the cumulative distance up to this point - note that this is not straight-line distance, but instead the distance along the path
- `cost_cell`: the cost of the cell that contains this point
- `id`: the ID of the cell that contains this point

If no path is possible between the two points, a zero-row matrix with the previously described columns is returned.

See Also

`lcp_finder()` creates the LCP finder object used as input to this function. `find_lcps()` calculates all LCPs whose cost-distance is less than some value. `summarize_lcps()` outputs a summary matrix of all LCPs that have been calculated so far.

Examples

```
####### NOTE #######
# see the "quadtree-lcp" vignette for more details and examples:
# vignette("quadtree-lcp", package = "quadtree")
####################

library(quadtree)
habitat <- terra::rast(system.file("extdata", "habitat.tif", package="quadtree"))

# create a quadtree
qt <- quadtree(habitat, split_threshold = .1, adj_type = "expand")
plot(qt, crop = TRUE, na_col = NULL, border_lwd = .4)

# define our start and end points
start_pt <- c(6989, 34007)
end_pt <- c(33015, 38162)

# create the LCP finder object and find the LCP
lcpf <- lcp_finder(qt, start_pt)
path <- find_lcp(lcpf, end_pt)

# plot the LCP
plot(qt, crop = TRUE, na_col = NULL, border_col = "gray30", border_lwd = .4)
points(rbind(start_pt, end_pt), pch = 16, col = "red")
lines(path[, 1:2], col = "black")

# note that the above path can also be found as follows:
path <- find_lcp(qt, start_pt, end_pt)
```

find_lcps *Find LCPs to surrounding points*

Description

Calculates least-cost paths (LCPs) from the start point (the point used to create the LcpFinder) to surrounding points. A constraint can be placed on the LCPs so that only LCPs that are less than some specified cost-distance are returned.

Usage

```
## S4 method for signature 'LcpFinder'
find_lcps(x, limit = NULL, return_summary = TRUE)
```

Arguments

x	a LcpFinder
limit	numeric; the maximum cost-distance for the LCPs. If NULL (the default), no limit is applied and all possible LCPs (within the LcpFinder's search area) are found
return_summary	boolean; if TRUE (the default), summarize_lcps() is used to return a summary matrix of all paths found. If FALSE, no value is returned.

Details

Once the LCPs have been calculated, find_lcp() can be used to extract paths to individual points. No further calculation will be required to retrieve these paths so long as they were calculated when find_lcps() was run.

A very important note to make is that once the LCP tree is calculated, it never gets smaller. For example, we could use find_lcps() with limit = NULL to calculate all LCPs. If we then used find_lcps() on the same LcpFinder but this time used a limit, it would still return *all* of the LCPs, even those that are greater than the specified limit, since the tree never shrinks.

Value

If return_summary is TRUE, summarize_lcps() is used to return a matrix summarizing each LCP found. See the help page of that function for details on the return matrix. If return_summary is FALSE, no value is returned.

See Also

lcp_finder() creates the LcpFinder object used as input to this function. find_lcp() returns the LCP between the start point and another point. summarize_lcps() outputs a summary matrix of all LCPs that have been calculated so far.

Examples

```
####### NOTE #######
# see the "quadtree-lcp" vignette  for more details and examples:
# vignette("quadtree-lcp", package = "quadtree")
####################

library(quadtree)
habitat <- terra::rast(system.file("extdata", "habitat.tif", package="quadtree"))

qt <- quadtree(habitat, split_threshold = .1, adj_type = "expand")

start_pt <- c(19000, 25000)

# finds LCPs to all cells
lcpf1 <- lcp_finder(qt, start_pt)
paths1 <- find_lcps(lcpf1, limit = NULL)

# limit LCPs by cost-distance
lcpf2 <- lcp_finder(qt, start_pt)
paths2 <- find_lcps(lcpf2, limit = 5000)

# Now plot the reachable cells
plot(qt, crop = TRUE, na_col = NULL, border_lwd = .3)
points(lcpf1, col = "black", pch = 16, cex = 1)
points(lcpf2, col = "red", pch = 16, cex = .7)
points(start_pt[1], start_pt[2], bg = "skyblue", col = "black", pch = 24,
       cex = 1.5)
```

`get_neighbors` *Get the neighbors of a* `Quadtree` *cell*

Description

Returns a matrix with information about the neighbors of a quadtree cell.

Usage

```
## S4 method for signature 'Quadtree,numeric'
get_neighbors(x, y)
```

Arguments

x	Quadtree
y	two-element numeric vector; the x and y coordinates of a point - this is used to identify which quadtree cell to find neighbors for.

Value

A six-column matrix with one row per neighboring cell. It has the following columns:

- `id`: the ID of the cell
- `xmin`, `xmax`, `ymin`, `ymax`: the x and y limits of the cell
- `value`: the value of the cell

Note that this return matrix only includes terminal nodes/cells - that is, cells that have no children. Also note that cells that are diagonal from each other are considered to be neighbors.

Examples

```
library(quadtree)
habitat <- terra::rast(system.file("extdata", "habitat.tif", package="quadtree"))

# create a quadtree
qt <- quadtree(habitat, split_threshold = .1, adj_type = "expand")

# get the cell's neighbors
pt <- c(27000, 10000)
nbs <- get_neighbors(qt, pt)

# plot the neighbors
plot(qt, border_lwd = .3)
points(pt[1], pt[2], col = "black", bg = "lightblue", pch = 21)
with(data.frame(nbs),
   rect(xmin, ymin, xmax, ymax, col = "red", border = "black", lwd = 2))
```

`LcpFinder-class` *LcpFinder Class*

Description

This S4 class is a wrapper around a `CppLcpFinder` C++ object that is made available to R via the 'Rcpp' package. Instances of this class can be created from a `Quadtree` object using the `lcp_finder` function.

The methods of the C++ object (`CppLcpFinder`) can be accessed from R, but the typical end-user should have no need of these methods - they are meant for internal use. That being said, descriptions of the available methods can be found on the `CppLcpFinder` documentation page.

Details

Functions for creating a `LcpFinder` object:

- `lcp_finder()`

Methods:

- `find_lcp()`

- find_lcps()
- plot()
- show()
- summarize_lcps()
- summary()

Slots

ptr a C++ object of class CppLcpFinder

---

lcp_finder *Create a* LcpFinder

Description

Creates a LcpFinder object that can then be used by find_lcp and find_lcps to find least-cost paths (LCPs) using a Quadtree as a resistance surface.

Usage

```
## S4 method for signature 'Quadtree'
lcp_finder(
  x,
  start_point,
  xlim = NULL,
  ylim = NULL,
  new_points = matrix(nrow = 0, ncol = 2),
  search_by_centroid = FALSE
)
```

Arguments

x	a Quadtree to be used as a resistance surface
start_point	two-element numeric vector (x, y) - the x and y coordinates of the starting point
xlim	two-element numeric vector (xmin, xmax) - constrains the nodes included in the network to those whose x limits fall in the range specified in xlim. If NULL the x limits of x are used
ylim	same as xlim, but for y
new_points	a two-column matrix representing point coordinates. First column contains the x-coordinates, second column contains the y-coordinates. This matrix specifies point locations to use instead of the node centroids. See 'Details' for more.
search_by_centroid	boolean; determines which cells are considered to be "in" the box specified by xlim and ylim. If FALSE (the default) any cell that overlaps with the box is included. If TRUE, a cell is only included if its **centroid** falls inside the box.

Details

See the vignette 'quadtree-lcp' for more details and examples (i.e. run `vignette("quadtree-lcp", package = "quadtree")`)

To find a least-cost path, the cells are treated as points - by default, the cell centroids are used. This results in some degree of error, especially for large cells. The `new_points` parameter can be used to specify the points used to represent the cells - this is particularly useful for specifying the points to be used for the start and end cells. Each point in the matrix will be used as the point for the cell it falls in (if two points fall in the same cell, the first point is used). Note that this raises the possibility that a straight line between neighboring cells may pass through other cells as well, which complicates the calculation of the edge cost. To mitigate this, when a straight line between neighboring cells passes through a different cell, the path is adjusted so that it actually consists of two segments - the start point to the "corner point" where the two cells meet, and then from that point to the end point. See the "quadtree-lcp" vignette for a graphical example of this situation.

An `LcpFinder` saves state, so once the LCP tree is calculated, individual LCPs can be retrieved without further computation. This makes it efficient at calculating multiple LCPs from a single starting point. However, in the case where only a single LCP is needed, `find_lcp()` offers an interface for finding an LCP without needing to use `lcp_finder()` to create the `LcpFinder` object first.

Value

a `LcpFinder`

See Also

`find_lcp()` returns the LCP between the start point and another point. `find_lcps()` finds all LCPs whose cost-distance is less than some value. `summarize_lcps()` outputs a summary matrix of all LCPs that have been calculated so far. `points()` and `lines()` can be used to plot a `LcpFinder`.

Examples

```
####### NOTE #######
# see the "quadtree-lcp" vignette for more details and examples:
# vignette("quadtree-lcp", package = "quadtree")
####################

library(quadtree)

habitat <- terra::rast(system.file("extdata", "habitat.tif", package="quadtree"))
qt <- quadtree(habitat, split_threshold = .1, adj_type = "expand")

# find the LCP between two points
start_pt <- c(6989, 34007)
end_pt <- c(33015, 38162)

# create the LCP finder object and find the LCP
lcpf <- lcp_finder(qt, start_pt)
path <- find_lcp(lcpf, end_pt)
```

```
# plot the LCP
plot(qt, crop = TRUE, na_col = NULL, border_lwd = .3)
points(rbind(start_pt, end_pt), pch = 16, col = "red")
lines(path[, 1:2], col = "black")
```

---

n_cells *Get the number of cells in a* Quadtree

Description

Returns the number of nodes/cells in the quadtree.

Usage

```
## S4 method for signature 'Quadtree'
n_cells(x, terminal_only = TRUE)
```

Arguments

x	a Quadtree
terminal_only	boolean; if TRUE (the default) only the terminal nodes are counted. If FALSE, all nodes are counted, thereby giving the total number of nodes in the tree.

Value

a numeric

Examples

```
library(quadtree)
habitat <- terra::rast(system.file("extdata", "habitat.tif", package="quadtree"))

qt <- quadtree(habitat, .1)
n_cells(qt)
n_cells(qt, terminal_only = FALSE)
```

plot *Plot a* Quadtree

Description

Plots a Quadtree.

Usage

```
## S4 method for signature 'Quadtree,missing'
plot(
  x,
  add = FALSE,
  col = NULL,
  alpha = NULL,
  nb_line_col = NULL,
  border_col = "black",
  border_lwd = 0.4,
  xlim = NULL,
  ylim = NULL,
  zlim = NULL,
  crop = FALSE,
  na_col = "white",
  adj_mar_auto = 6,
  legend = TRUE,
  legend_args = list(),
  ...
)
```

Arguments

x	a Quadtree
add	boolean; if FALSE (the default) a new plot is created. If TRUE, the plot is added to the existing plot.
col	character vector; the colors that will be used to create the color ramp used in the plot. If no argument is provided, terrain.colors(100, rev = TRUE) is used.
alpha	numeric; transparency of the cell colors. Must be in the range 0-1, where 0 is fully transparent and 1 is fully opaque. If NULL (the default) it sets alpha to 1.
nb_line_col	character; the color of the lines drawn between neighboring cells. If NULL (the default), these lines are not plotted.
border_col	character; the color to use for the cell borders. Use "transparent" if you don't want borders to be shown. Default is "black".
border_lwd	numeric; the line width of the cell borders. Default is .4.
xlim	two-element numeric vector; defines the minimum and maximum values of the x axis. Note that this overrides the crop parameter.

ylim	two-element numeric vector; defines the minimum and maximum values of the y axis. Note that this overrides the crop parameter.
zlim	two-element numeric vector; defines how the colors are assigned to the cell values. The first color in col will correspond to zlim[1] and the last color in col will correspond to zlim[2]. If zlim does not encompass the entire range of cell values, cells that have values outside of the range specified by zlim will be treated as NA cells. If this value is NULL (the default), it uses the min and max cell values.
crop	boolean; if TRUE, only displays the extent of the original raster, thus ignoring any of the NA cells that were added to pad the raster before making the quadtree. Ignored if either xlim or ylim are non-NULL.
na_col	character; the color to use for NA cells. If NULL, NA cells are not plotted. Default is "white".
adj_mar_auto	numeric; checks the size of the right margin (par("mar")[4]) - if it is less than the provided value and legend is TRUE, then it sets it to be the provided value in order to make room for the legend (after plotting, it resets it to its original value). If NULL, the margin is not adjusted. Default is 6.
legend	boolean; if TRUE (the default) a legend is plotted in the right margin.
legend_args	named list; contains arguments that are sent to the add_legend() function. See the help page for add_legend() for the parameters. Note that zlim, cols, and alpha are supplied automatically, so if the list contains elements named zlim, cols, or alpha the user-provided values will be ignored.
...	arguments passed to the default plot() function

Details

See 'Examples' for demonstrations of how the various options can be used.

Value

no return value

Examples

```
library(quadtree)
habitat <- terra::rast(system.file("extdata", "habitat.tif", package="quadtree"))

# create quadtree
qt <- quadtree(habitat, split_threshold = .1, adj_type = "expand")

####################################
# DEFAULT
####################################

# default - no additional parameters provided
plot(qt)

####################################
```

```
# CHANGE PLOT EXTENT
####################################

# note that additional parameters like 'main', 'xlab', 'ylab', etc. will be
# passed to the default 'plot()' function

# crop extent to the original extent of the raster
plot(qt, crop = TRUE, main = "cropped")

# crop and don't plot NA cells
plot(qt, crop = TRUE, na_col = NULL, main = "cropped")

# use 'xlim' and 'ylim' to zoom in on an area
plot(qt, xlim = c(10000, 20000), ylim = c(20000, 30000), main = "zoomed in")

####################################
# COLORS AND BORDERS
####################################

# change border color and width
plot(qt, border_col = "transparent") # no borders
plot(qt, border_col = "gray60") # gray borders
plot(qt, border_lwd = .3) # change line thickness of borders

# change color palette
plot(qt, col = c("blue", "yellow", "red"))
plot(qt, col = hcl.colors(100))
plot(qt, col = c("black", "white"))

# change color transparency
plot(qt, alpha = .5)
plot(qt, col = c("blue", "yellow", "red"), alpha = .5)

# change color of NA cells
plot(qt, na_col = "lavender")

# don't plot NA cells at all
plot(qt, na_col = NULL)

# change 'zlim'
plot(qt, zlim = c(0, 5))
plot(qt, zlim = c(.2, .7))

####################################
# SHOW NEIGHBOR CONNECTIONS
####################################

# plot all neighbor connections
plot(qt, nb_line_col = "black", border_col = "gray60")

# don't plot connections to NA cells
plot(qt, nb_line_col = "black", border_col = "gray60", na_col = NULL)
```

```
###################################
# LEGEND
###################################

# no legend
plot(qt, legend = FALSE)

# increase right margin size
plot(qt, adj_mar_auto = 10)

# use 'legend_args' to customize the legend
plot(qt, adj_mar_auto = 10,
     legend_args = list(lgd_ht_pct = .8, bar_wd_pct = .4))
```

`plot.LcpFinder` *Plot a* `LcpFinder` *object*

Description

Plots a `LcpFinder` object.

Usage

```
## S4 method for signature 'LcpFinder'
points(x, add = TRUE, ...)

## S4 method for signature 'LcpFinder'
lines(x, add = TRUE, ...)
```

Arguments

x	a `LcpFinder`
add	boolean; if `TRUE` (the default), the plot is added to the existing plot. If `FALSE`, a new plot is created.
...	arguments passed to the default plotting functions

Details

`points()` plots points at the centroids of the cells to which a path has been found. `lines()` plots all of the LCPs found so far by the `LcpFinder` object.

Value

no return value

Examples

```
library(quadtree)
habitat <- terra::rast(system.file("extdata", "habitat.tif", package="quadtree"))

qt <- quadtree(habitat, .1)

start_point <- c(6989, 34007)
end_point <- c(12558, 27602)
lcpf <- lcp_finder(qt, start_point)
lcp <- find_lcp(lcpf, end_point)

plot(qt, crop = TRUE, border_lwd = .3, na_col = NULL)
points(lcpf, col = "red", pch = 16, cex = .4)
lines(lcpf)
```

---

projection *Retrieve the projection of a* Quadtree

Description

Retrieves the projection of a Quadtree.

Usage

```
## S4 method for signature 'Quadtree'
projection(x)

## S4 replacement method for signature 'Quadtree'
projection(x) <- value
```

Arguments

x	a Quadtree
value	character; the projection to assign to the Quadtree

Value

a string

Examples

```
library(quadtree)
habitat <- terra::rast(system.file("extdata", "habitat.tif", package="quadtree"))

qt <- quadtree(habitat, .1)
quadtree::projection(qt) <- "+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs"
quadtree::projection(qt)
```

quadtree *Create a* Quadtree *from a raster or matrix*

Description

Creates a Quadtree from a SpatRaster, RasterLayer or a matrix.

Usage

```
## S4 method for signature 'ANY'
quadtree(
  x,
  split_threshold = NULL,
  split_method = "range",
  split_fun = NULL,
  split_args = list(),
  split_if_any_na = TRUE,
  split_if_all_na = FALSE,
  combine_method = "mean",
  combine_fun = NULL,
  combine_args = list(),
  max_cell_length = NULL,
  min_cell_length = NULL,
  adj_type = "expand",
  resample_n_side = NULL,
  resample_pad_nas = TRUE,
  extent = NULL,
  projection = "",
  proj4string = NULL,
  template_quadtree = NULL
)
```

Arguments

x	a RasterLayer, SpatRaster, or matrix. If x is a matrix, the extent and projection parameters can be used to set the extent and projection of the quadtree. If x is a RasterLayer or SpatRaster, the extent and projection are derived from the raster.
split_threshold	numeric; the threshold value used by the split method (specified by split_method) to decide whether to split a quadrant. If the value for a quadrant is greater than this value, it is split into its four child cells. If split_method is "custom", this parameter is ignored.
split_method	character; one of "range" (the default), "sd" (standard deviation), "cv" (coefficient of variation) or "custom". Determines the method used for calculating the value used to determine whether or not to split a quadrant (this calculated

value is compared with split_threshold to decide whether to split a cell). If "custom", a function must be supplied to split_fun. See 'Details' for more.

split_fun function; function used on each quadrant to decide whether or not to split the quadrant. Only used when split_method is "custom". Must take two arguments, vals (a numeric vector of the cell values in a quadrant) and args (a named list of arguments used within the function), and must output TRUE if the quadrant is to be split and FALSE otherwise. It must be able to handle NA values - if NA is ever returned, an error will occur.

split_args list; named list that contains the arguments needed by split_fun. This list is given to the args parameter of split_fun.

split_if_any_na
boolean; if TRUE (the default), a quadrant is automatically split if any of the values within the quadrant are NA.

split_if_all_na
boolean; if FALSE (the default), a quadrant that contains only NA values is not split. If TRUE, quadrants that contain all NA values are split to the smallest possible cell size.

combine_method character; one of "mean", "median", "min", "max", or "custom". Determines the method used for aggregating the values of multiple cells into a single value for a larger, aggregated cell. Default is "mean". If "custom", a function must be supplied to combine_fun.

combine_fun function; function used to calculate the value of a quadrant. Only used when combine_method is "custom". Must take two arguments, vals (a numeric vector of the cell values in a quadrant) and args (a named list of arguments used within the function), and must output a single numeric value, which will be used as the cell value.

combine_args list; named list that contains the arguments needed by combine_fun. This list is given to the args parameter of combine_fun.

max_cell_length
numeric; the maximum side length allowed for a quadtree cell. Any quadrants larger than max_cell_length will automatically be split. If NULL (the default) no restrictions are placed on the maximum cell length.

min_cell_length
numeric; the minimum side length allowed for a quadtree cell. A quadrant will not be split if its children would be smaller than min_cell_length. If NULL (the default) no restrictions are placed on the minimum cell length.

adj_type character; one of "expand" (the default), "resample", or "none". Specifies the method used to adjust x so that its dimensions are suitable for quadtree creation (i.e. square and with the number of cells in each direction being a power of 2). See 'Details' for more on the two methods of adjustment.

resample_n_side
integer; if adj_type is 'resample', this number is used to determine the dimensions to resample the raster to.

resample_pad_nas
boolean; only applicable if adj_type is 'resample'. If TRUE (the default), NAs are added to the shorter side of the raster to make it square before resampling.

This ensures that the cells of the resulting quadtree will be square. If FALSE, no NAs are added - the cells in the quadtree will not be square.

`extent` Extent object or else a four-element numeric vector describing the extent of the data (in this order: xmin, xmax, ymin, ymax). Only used when x is a matrix - this parameter is ignored if x is a raster since the extent is derived directly from the raster. If no value is provided and x is a matrix, the extent is assumed to be `c(0,ncol(x),0,nrow(x))`.

`projection` character; string describing the projection of the data. Only used when x is a matrix - this parameter is ignored if x is a raster since the proj4ection of the raster is automatically used. If no value is provided and x is a matrix, the projection of the quadtree is set to NA.

`proj4string` deprecated. Use `projection` instead.

`template_quadtree`

Quadtree; if provided, the new quadtree will be created so that it has the exact same structure as the template quadtree. Thus, no split function is used because the decision about whether to split is pre-determined by the template quadtree. The raster used to create the template quadtree should have the exact same extent and dimensions as x. If `template_quadtree` is non-NULL, all `split_*` parameters are disregarded, as are `max_cell_length` and `min_cell_length`.

Details

The 'quadtree-creation' vignette contains detailed explanations and examples for all of the various creation options - run `vignette("quadtree-creation", package = "quadtree")` to view the vignette.

If `adj_type` is `"expand"`, NA cells are added to the raster in order to create an expanded raster whose dimensions are a power of two. The smallest number that is a power of two but greater than the larger dimension is used as the dimensions of the expanded raster. If `adj_type` is `"resample"`, the raster is resampled to a raster with `resample_n_side` rows and columns. If `resample_pad_nas` is TRUE, NA rows or columns are are added to the shorter dimension before resampling to make the raster square. This ensures that the quadtree cells will be square (assuming the original raster cells were square).

When `split_method` is `"range"`, the difference between the maximum and minimum cell values in a quadrant is calculated - if this value is greater than `split_threshold`, the quadrant is split. When `split_method` is `"sd"`, the standard deviation of the cell values in a quadrant is calculated - if this value is greater than `split_threshold`, the quadrant is split.

Value

a Quadtree

Examples

```
####### NOTE #######
# see the "quadtree-creation" vignette for more details and examples of all
# the different parameter options:
# vignette("quadtree-creation", package = "quadtree")
####################
```

```
library(quadtree)
habitat <- terra::rast(system.file("extdata", "habitat.tif", package="quadtree"))

qt <- quadtree(habitat, .15)
plot(qt)
# we can make it look nicer by customizing the plotting parameters
plot(qt, crop = TRUE, na_col = NULL, border_lwd = .3)

# try a different splitting method
qt <- quadtree(habitat, .05, "sd")
plot(qt)

# ---- using a custom split function ----

# split a cell if any of the values are below a given value
split_fun = function(vals, args) {
  if (any(is.na(vals))) { # check for NAs first
    return(TRUE) # if there are any NAs we'll split automatically
  } else {
    return(any(vals < args$threshold))
  }
}

qt <- quadtree(habitat, split_method = "custom", split_fun = split_fun,
               split_args = list(threshold = .8))
plot(qt)
```

`Quadtree-class` *Quadtree class*

Description

This S4 class is essentially a wrapper around a `CppQuadtree` C++ object. `Quadtree` has one slot, which is named `ptr` and contains a `CppQuadtree` object. Instances of this class can be created through the `quadtree()` function.

An important note to make is that functions that modify a `Quadtree` modify the existing object. For example, running `transform_values(qt, function(x) x+1)` modifies `qt`. This differs from the way R objects usually function - most functions that modify R objects return a modified copy of the object, thus preserving the original object. Note that the `copy()` function, which makes a deep copy of a `Quadtree`, can be used to preserve a copy of a `Quadtree` before modifying it.

The methods of the C++ object (`CppQuadtree`) stored in the `ptr` slot can be accessed from R, but the typical end-user should have no need of these methods - they are meant for internal use. That being said, descriptions of the available methods can be found on the `CppQuadtree` documentation page.

Details

Functions for creating a Quadtree object:

- quadtree()
- read_quadtree()

Methods:

- as_data_frame()
- as_raster()
- as_vector()
- copy()
- extent()
- extract()
- get_neighbors()
- lcp_finder()
- n_cells()
- projection()
- plot()
- set_values()
- show()
- summary()
- transform_values()
- write_quadtree()

Slots

ptr a C++ object of class CppQuadtree

---

read_quadtree *Read/write a* Quadtree

Description

Reads and writes a Quadtree.

Usage

```
## S4 method for signature 'character'
read_quadtree(x)

## S4 method for signature 'character,Quadtree'
write_quadtree(x, y)
```

Arguments

x	character; the filepath to read from or write to
y	a Quadtree

Details

To read/write a quadtree object, the C++ library `cereal` is used to serialize the quadtree and save it to a file. The file extension is unimportant - it can be anything (I've been using the extension '.qtree').

Value

`read_quadtree()` - returns a Quadtree

`write_quadtree()` - no return value

Examples

```
library(quadtree)
habitat <- terra::rast(system.file("extdata", "habitat.tif", package="quadtree"))

qt <- quadtree(habitat, .1)

path <- tempfile(fileext = "qtree")
write_quadtree(path, qt)
qt2 <- read_quadtree(path)
```

---

set_values *Change values of* Quadtree *cells*

Description

Given a Quadtree, a set of points, and a vector of new values, changes the value of the quadtree cells containing the points to the corresponding value.

Usage

```
## S4 method for signature 'Quadtree,ANY,numeric'
set_values(x, y, z)
```

Arguments

x	A Quadtree
y	A two-column matrix representing point coordinates. First column contains the x-coordinates, second column contains the y-coordinates.
z	A numeric vector the same length as the number of rows of y. The values of the cells containing y will be changed to the corresponding value in z.

Details

Note that it is entirely possible for y to contain multiple points that all fall within the same cell. The values are changed in the order given, so the cell will take on the *last* value given for that cell.

It's important to note that this modifies the original quadtree. If you wish to maintain a version of the original quadtree, use `copy` beforehand to make a copy of the quadtree.

Value

no return value

See Also

`transform_values()` can be used to transform the existing values of all cells using a function.

Examples

```
library(quadtree)
habitat <- terra::rast(system.file("extdata", "habitat.tif", package="quadtree"))

# create a quadtree
qt <- quadtree(habitat, split_threshold = .1)

# generate some random points, then change the values at those points
ext <- extent(qt)
pts <- cbind(runif(100, ext[1], ext[2]), runif(100, ext[3], ext[4]))
set_values(qt, pts, rep(10, 100))

# plot it out to see what happened
old_par <- par(mfrow = c(1, 2))
plot(qt, main = "original")
plot(qt, main = "after modification")
par(old_par)
```

---

summarize_lcps *Get a matrix summarizing all LCPs found by a* LcpFinder

---

Description

Given a `LcpFinder`, returns a matrix that summarizes all of the LCPs that have been calculated by the `LcpFinder`.

Usage

```
## S4 method for signature 'LcpFinder'
summarize_lcps(x)
```

Arguments

x	a `LcpFinder`

Details

Note that this function returns **all** of the paths that have been calculated. Finding one LCP likely involves finding other LCPs as well. Thus, even if the `LcpFinder` has been used to find one LCP, others have most likely been calculated. This function returns all of the LCPs that have been calculated so far.

Value

Returns a nine-column matrix with one row for each LCP (and therefore one row per destination cell). The columns are as follows:

- `id`: the ID of the destination cell
- `xmin`, `xmax`, `ymin`, `ymax`: the extent of the destination cell
- `value`: the value of the destination cell
- `area`: the area of the destination cell
- `lcp_cost`: the cumulative cost of the LCP to this cell
- `lcp_dist`: the cumulative distance of the LCP to this cell - note that this is not straight-line distance, but instead the distance along the path

See Also

`lcp_finder()` creates the `LcpFinder` object used as input to this function. `find_lcp()` returns the LCP between the start point and another point. `find_lcps()` calculates all LCPs whose cost-distance is less than some value.

Examples

```
library(quadtree)
habitat <- terra::rast(system.file("extdata", "habitat.tif", package="quadtree"))

qt <- quadtree(habitat, split_threshold = .1, adj_type = "expand")

start_pt <- c(19000, 25000)
end_pt <- c(33015, 38162)

# find LCP from 'start_pt' to 'end_pt'
lcpf <- lcp_finder(qt, start_pt)
lcp <- find_lcp(lcpf, end_pt)

# retrieve ALL the paths that have been calculated
paths <- summarize_lcps(lcpf)
head(paths)
```

summary.LcpFinder *Show a summary of a* LcpFinder

Description

Prints out information about the LcpFinder. Information shown is:

- class of object
- start point
- search limits
- number of paths found

Usage

```
## S4 method for signature 'LcpFinder'
summary(object)

## S4 method for signature 'LcpFinder'
show(object)
```

Arguments

object	a LcpFinder

Value

no return value

Examples

```
library(quadtree)
habitat <- terra::rast(system.file("extdata", "habitat.tif", package="quadtree"))

qt <- quadtree(habitat, .1)

start_point <- c(6989, 34007)
end_point <- c(33015, 38162)

lcpf <- lcp_finder(qt, start_point)
lcp <- find_lcp(lcpf, end_point)

summary(lcpf)
```

summary.Quadtree *Show a summary of a* Quadtree

Description

Prints out information about a Quadtree. Information shown is:

- class of object
- number of cells
- minimum cell size
- extent
- projection
- minimum and maximum values

Usage

```
## S4 method for signature 'Quadtree'
summary(object)

## S4 method for signature 'Quadtree'
show(object)
```

Arguments

object	a Quadtree object

Value

no return value

Examples

```
library(quadtree)
habitat <- terra::rast(system.file("extdata", "habitat.tif", package="quadtree"))

qt <- quadtree(habitat, .1)
summary(qt)
```

transform_values *Transform the values of all* Quadtree *cells*

Description

Uses a function to change all cell values of a Quadtree.

Usage

```
## S4 method for signature 'Quadtree,`function`'
transform_values(x, y)
```

Arguments

x	A Quadtree
y	function; function used on each cell to transform the value. Must accept a single numeric value and return a single numeric value. The function must also be able to handle NA values.

Details

This function applies a function to every single cell, which allows the user to do things like multiply by a scalar, invert the values, etc.

Since a quadtree may contain NA values, y must be able to handle NAs without throwing an error. For example, if y contains some control statement such as `if(x < .7)`, the function must have a separate statement before this to catch NA values, since having an NA in an if statement is not allowed. See 'Examples' for an example of this.

It's important to note that this modifies the original quadtree. If you wish to maintain a version of the original quadtree, use copy beforehand to make a copy of the quadtree (see 'Examples').

Value

no return value

See Also

set_values() can be used to set the values of cells to specified values (rather than transforming the existing values).

Examples

```
library(quadtree)
habitat <- terra::rast(system.file("extdata", "habitat.tif", package="quadtree"))

# create a quadtree
qt1 <- quadtree(habitat, split_threshold = .1)
```

```
# copy the quadtree so that we have a copy of the original (since using
#'transform_values' modifies the quadtree object)
qt2 <- copy(qt1)
qt3 <- copy(qt1)
qt4 <- copy(qt1)

transform_values(qt2, function(x) 1 - x)
transform_values(qt3, function(x) x^3)
transform_values(qt4, function(x) {
  if (is.na(x)) return(NA) # make sure to handle NA's
  if (x < .7) return(0)
  return(1)
})

old_par <- par(mfrow = c(2, 2))
plot(qt1, main = "original", crop = TRUE, na_col = NULL,
     border_lwd = .3, zlim = c(0, 1))
plot(qt2, main = "1 - value", crop = TRUE, na_col = NULL,
     border_lwd = .3, zlim = c(0, 1))
plot(qt3, main = "values cubed", crop = TRUE, na_col = NULL,
     border_lwd = .3, zlim = c(0, 1))
plot(qt4, main = "values converted to 0/1", crop = TRUE, na_col = NULL,
     border_lwd = .3, zlim = c(0, 1))
par(old_par)
```

`write_quadtree_ptr` *Read/write a* `Quadtree`

Description

This is for debugging only, and users should never need to use this function - use `write_quadtree()` instead. `write_quadtree()` serializes the `CppQuadtree` object (note that the underlying C++ object is actually called `QuadtreeWrapper`, but it is exposed to R as `CppQuadtree`) stored in the `ptr` slot of `Quadtree`.

This function, however, serializes only the `Quadtree` object contained by the `QuadtreeWrapper`.

Usage

```
## S4 method for signature 'character,Quadtree'
write_quadtree_ptr(x, y)
```

Arguments

x	character; the filepath to read from or write to
y	a `Quadtree`

Value

no return value

Examples

```
library(quadtree)
habitat <- terra::rast(system.file("extdata", "habitat.tif", package="quadtree"))

qt <- quadtree(habitat, .1)

path <- tempfile(fileext = "qtree")
write_quadtree_ptr(path, qt)
```

Index