Package 'quadtree'

August 12, 2021

Type Package

Title Quadtree Representation of Rasters

Version 0.0.0.9000

Date 2021-01-26

Description Provides a C++ implementation of a quadtree data structure. Functions are provided for creating a quadtree from a raster, with the level of 'coarseness' of the quadtree being determined by a user-supplied parameter. In addition, functions are provided to extract values of the quadtree at point locations. Functions for calculating least-cost paths between two points are also provided.

License GPL (>= 2)

Imports Rcpp (>= 1.0.5),
raster,
dplyr,
methods

LinkingTo Rcpp

RoxygenNote 7.1.1

NeedsCompilation yes

URL https://gitlab.com/dafriend/quadtree

Depends R (>= 2.10)

Suggests testthat (>= 3.0.0)

Config/testthat/edition 3

R topics documented:

`quadtree-package` *Quadtree Representation of Rasters*

Description

Provides a C++ implementation of a quadtree data structure. Functions are provided for creating a quadtree from a raster, with the level of "coarseness" of the quadtree being determined by a user-supplied parameter. In addition, functions are provided to extract values of the quadtree at point locations. Functions for calculating least-cost paths between two points are also provided.

Details

To get an understanding of the most important aspects of the package, read the following:

1. For details on how a quadtree is constructed, see `qt_create`.
2. For details on the least-cost path functionality, see `qt_lcp_finder`.

Function summary:

`qt_create`: create a quadtree from a raster

`qt_extent`: get the extent of a quadtree

`qt_extract`: extract values from the quadtree at point locations

`qt_lcp_finder`, `qt_find_lcp`: find the LCP between two points using the quadtree as a cost surface

`qt_plot`: plot a quadtree

`qt_proj4string`: get the proj4string of a quadtree

`qt_read`, `qt_write`: read and write a quadtree object to a file

Author(s)

Derek Friend

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

Description

Adds a gradient legent 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,
  ticks = NULL,
  ticks_n = 5,
  ticks_x_pct = 1
)
```

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
`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)

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 from grDevices is used to generate "nice-looking" values, but it doesn't guarantee a set number of tick marks
ticks_x_pct	numeric; the x-placement of the tick labels 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

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 then 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 bar and ticks are relative to the space within the legend box.

I obviously wrote this for plotting the quadtree, but there's nothing quadtree-specific about this particular function.

This function is used within qt_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 qt_plot(). Using this function to plot the legend after using qt_plot() raises the possibility of the legend not corresponding correctly with the plot, and thus should be avoided.

Examples

```
set.seed(23)
mat = matrix(runif(64,0,1), nrow=8)
qt = qt_create(mat, .75)

par(mar=c(5,4,4,5))
qt_plot(qt,legend=FALSE)
add_legend(range(mat), rev(terrain.colors(100)))
# this example simply illustrates how it COULD be used, but as stated in the
# 'Details' section, it shouldn't be called separately from 'qt_plot()' - if
# customizations to the legend are desired, use the 'legend_args' parameter
# of 'qt_plot()'.
```

---

get_coords *Get the extent of the figure area in plot units (for one dimension)*

Description

Given the coordinate range of a single dimension in user units (par("usr")) and the coordinates of that same coordinate range as a fraction of the current figure region (par("plt")), calculates the extent of the entire figure area in user units.

Usage

```
get_coords_axis(usr, plt)

get_coords(usr, plt)
```

Arguments

usr	two-element (get_coords_axis) or four-element (get_coords) numeric vector; specifies the user coordinates of the plot region. Can be retrieved using par("usr"), and subscripts can be used to get only one dimension (for get_coords_axis - i.e par("usr")[1:2])
plt	two-element (get_coords_axis) or four-element (get_coords) numeric vector; specifies the coordinates of the plot region as fractions of the figure region. Can be retrieved using par("plt"), and subscripts can be used to get only one dimension (for get_coords_axis - i.e par("plt")[1:2])

Details

get_coords_axis() is used to find the user coordinates of a single dimension of the figure area. In this case, usr and plt should both be two-element vectors corresponding to the same dimension (see examples). Both vectors need to be in the format c(max,min).

get_coords() is simply a wrapper for get_coords_axis that does both dimensions at once. In this case the output of par("usr") and par("plt") can be directly supplied to the usr and plt parameters, respectively. Note that for both parameters the vectors must have length 4 and be in this order: c(xmin,xmax,ymin,ymax).

These functions were written for use in add_legend. In order to properly place the legend, I needed to know the extent of the entire figure region in user coordinates. However, there's nothing about this function that is specific to that one application, and could be used in other situations as well.

Understanding what these functions do (and why they're necessary) requires an understanding of the graphical parameters, and in particular what usr and plt represent. See ?par for more on these parameters.

See Also

Run ?par for more details on the usr and plt parameters

Examples

```
p = par() # retrieve the graphical parameters as a list
get_coords_axis(p$usr[1:2], p$plt[1:2]) # x-axis
get_coords_axis(p$usr[3:4], p$plt[3:4]) # y-axis

get_coords(p$usr, p$plt) # both dimensions at once
get_coords(par("usr"), par("plt")) # this also works
```

---

habitat *Sample raster data for the 'quadtree' package*

Description

habitat is a raster containing habitat suitability values where each cell takes on a value between 0 and 1.

habitat_roads has the exact same footprint as 'habitat', but the values represent the presence/absence of roads in that cell. 1 indicates presence, while 0 indicates absence.

Usage

```
data("habitat")
data("habitat_roads")
```

Format

`RasterLayer` (from the 'raster' package)

Details

These rasters are included for two reasons: first, thet provide the datasets that are used for the examples. Second, they provide easy-to-access datasets for users to experiment with when learning about the 'quadtree' package.

Examples

```
data("habitat")
data("habitat_roads")

qt1 = qt_create(habitat, .1)
qt2 = qt_create(habitat_roads, .1)

qt_plot(qt1)
qt_plot(qt2)
```

`node-class` *C++ Quadtree Node*

Description

The 'node' class represents a single quadtree node.

Fields

`asVector`
- **Description**: Returns a vector giving info about the node
- **Parameters**: none
- **Returns**: a numeric vector with the following named elements:
  - `id`
  - `hasChdn`
  - `level`
  - `xMin`
  - `xMax`
  - `yMin`
  - `yMax`
  - `smSide`

  `qt_as_data_frame` makes use of this function to output info on each node - see the documentation of that function for details on the output

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

- **Returns**: a `list` of `Rcpp_node` objects

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

`getNeighborInfo`
- **Description**: Returns a matrix with info on each of the neighboring cells
- **Parameters**: none
- **Returns**: a matrix. `quadtree$getNbList()` 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 `Rcpp_node` 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**: 2-element numeric vector (xMin, xMax)

`yLims`
- **Description**: Returns the y boundaries of the node
- **Parameters**: none
- **Returns**: 2-element numeric vector (yMin, yMax)

qt_as_data_frame *convert a quadtree to a data frame*

Description

creates a data frame with information on each quadtree cell.

Usage

```
qt_as_data_frame(quadtree)
```

Arguments

quadtree	a quadtree object

Value

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

- id: the id of the cell
- hasChdn: 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
- smSide: the smallest cell length among all of this cells descendants
- parentID: the ID of the cell's parent. The root, which has no parent, has a value of -1 for this element

Examples

```
mat = matrix(rbind(c(1,1,0,1),c(1,1,1,0),c(1,0,1,1),c(0,1,1,1)))
qt = qt_create(mat,.1)
qt_plot(qt)
qt_as_data_frame(qt)
```

qt_copy *Create a deep copy of a quadtree object*

Description

Creates a *deep* copy of a quadtree object

Usage

```
qt_copy(quadtree)
```

Arguments

quadtree	The quadtree object to copy

Details

This function creates *deep* copy of a quadtree object. The quadtree class accessible from R uses pointers to a Quadtree C++ object. Thus, if a copy is attempted by simply assigning the quadtree to a new variable, it will only make a *shallow* copy, and both variables will refer to the same object. Thus, changes made to one will also change the other. For example, take the following code - assume that we already have a quadtree object in memory called qt, and that we have a matrix of points (pts) and values (vals):

```
qt_copy = qt
qt_set_values(qt,pts,vals)
```

Both qt **and** qt_copy will be changed by this operation.

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 object

Examples

```
data(habitat)

# create quadtree - then create a shallow copy and a deep copy for demonstration
qt1 = qt_create(habitat, split_threshold = .1)
qt_plot(qt1)

qt2 = qt1 # SHALLOW copy
qt3 = qt_copy(qt1) # DEEP copy

# change the values of qt1 so we can observe how this affects qt2 and qt3
ext = qt_extent(qt1)
pts = cbind(runif(100,ext[1], ext[2]), runif(100,ext[3], ext[4]))
qt_set_values(qt1, pts, rep(10,100))

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

---

qt_create *Create a quadtree from gridded data*

Description

Create a quadtree from gridded data

Usage

```
qt_create(
  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,
  proj4string = NULL,
  template_quadtree = NULL
)
```

Arguments

x	a raster or a matrix. If x is a matrix, the extent and proj4string parameters can be used to set the extent and projection of the quadtree. If x is a raster, 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 cell. If the value for a quadrant is greater than this value, the cell is split. If split_method is "custom", this parameter is ignored.
split_method	character; one of "range", "sd" (standard deviation), or "custom". Determines the method used for calculating the value used to determine whether or not to split a cell (this calculated value is compared with split_threshold to decide whether to split a cell). See 'Details' for more.
split_fun	function; function used on each quadrant to decide whether or not to split the cell. Only used when split_method is "custom". Must take two arguments, "vals" (a numeric vector) 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. See 'Details' and 'Examples' for more.
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"`.

`combine_fun` function; function used to calculate the value of a quadrant that consists of multiple cells. Only used when `combine_method` is "custom" Must take two arguments, `"vals"` (a numeric vector) 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 See 'Details' and 'Examples' for more.

`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. If `NULL` (the default) no restrictions are placed on the maximum cell length. See 'Details' for more.

`min_cell_length`
numeric; the minimum side length allowed for a quadtree cell. If `NULL` (the default) no restrictions are placed on the minimum cell length. See 'Details' for more.

`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), `NA`s 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 `NA`s 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))`.

`proj4string` character; proj4string describing the projection of the data. Only used when `x` is a matrix - this parameter is ignored if `x` is a raster since the proj4string of the raster is automatically used. If no value is provided and `x` is a matrix, the 'proj4string' of the quadtree is set to `NA`.

`template_quadtree`
quadtree object; 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

Overview of quadtree creation

A quadtree is created from a raster or a matrix by successively dividing the raster/matrix into smaller and smaller cells, with the decision on whether to divide a quadrant determined by a function that checks the cell values within each quadrant and returns TRUE if it should be split, and FALSE otherwise. Initially, all of the cells in the raster are considered. If the cell values meet the condition determined by the splitting function, the raster is divided into four quadrants - otherwise, the raster is not divided further and the value of this larger cell is calculated by applying a 'combine function' that aggregates the cell values into a single value (for example, mean and median). If the given cell is split, the process is repeated for each of those 'child' quadrants, and then for their children, and so on and so forth, until either the split function returns FALSE or the smallest possible cell size has been reached.

Pre-creation dimension adjustment

If a given quadrant has dimensions that are not divisible by 2 (for example, 5x5), then the process stops. Because of this, only rasters that have dimensions that are a power of 2 can be divided down to their smallest cell size. In addition, the rasters should be square. To create quadtrees from rasters that have dimensions that are not a power of two and are not square, two options are provided. The choice of method is determined by the `adj_type` parameter.

In the 'expand' method, NA cells are added to the raster in order to create an expanded raster whose dimensions are a power of 2. The smallest number that is a power of two but greater than the larger dimension is used as the dimensions of the expanded raster. For example, if a raster has dimensions 546 x 978, NA cells are added to the top and right of the raster in order to create a raster with dimensions 1024 x 1024 (as 1024 is the smallest power of 2 that is also greater than 978).

In the 'resample' method, the raster is resampled in order to create a square matrix with dimensions that are a power of two. If the data does not have same number of rows and columns, resampling the raster to have an equal number of rows and column will result in rectangular but non-square cells. If square cells are desired, an additional step is added to make the raster square by setting `resample_pad_NAs` to be TRUE (the default). This is done in a way similar to the method described above. The smaller dimension is padded with NA cells in order to equal the larger dimension. For example, if the raster has dimensions 546 x 978, NA rows are added in order to create a raster with dimensions 978 x 978. Then, regardless of whether `resample_pad_NAs` is TRUE or FALSE, the raster is resampled to a user-specified dimension (determined by the `resample_n_side` parameter). For example, the user could set `resample_n_side` to be 1024, which will resample the 978 x 978 raster to 1024 x 1024. This raster can then be used to create a quadtree. `resample_n_side` should be a power of 2 (see above for an explanation), although other numbers will be accepted (but will trigger a warning).

If `adj_type` is 'none', the provided matrix/raster is used 'as is', with no dimension adjustment.

Splitting and aggregating functions

The method used to determine whether or not to split a cell as well as the method used to aggregate cell values can be defined by the user. Simple methods are already provided, but custom functions can be defined. For splitting a cell, two methods are provided. "range" checks the difference between the minimum value and the maximum value within the quadrant - if this difference exceeds `split_threshold`, the quadrant is split. "sd" uses the standard deviation of the cell values within a quadrant - if the standard deviation exceeds `split_threshold`, the quadrant is split.

Four methods to aggregate cell values are provided - "mean", "median", "min", and "max" - the names are self-explanatory.

Custom functions can be written to apply more complex rules to splitting and combining. These functions *must* take two parameters: `vals` and `args`. `vals` is a numeric vector of the values of the cells within the current quadrant. `args` is a named list that contains the arguments need by the custom function. Any parameters needed for the function should be accessed through `args`. Note that even if no extra parameters are needed, the custom function still needs to take an `args` parameter - in that case it just won't be used by the function.

split_fun must return a boolean, where TRUE indicates that the quadrant should be split. An **important note** to make is that any custom function must be able to handle NA values. The function must *always* return either TRUE or FALSE - if NA is ever returned an error will occur.

For example, a simple splitting function that splits a quadrant when the variance exceeds a user-defined limit could be defined as follows:

```
splt_fun = function(vals, args) {
   if(any(is.na(vals))){
      return(TRUE);
   } else {
      return(sd(vals) > args$var_limit)
   }
}
```

Because the function makes use of an element of args named var_limit, the split_args parameter would need to contain an element called var_limit. For example:

```
qt = qt_create(rast,split_method="custom",split_fun=splt_fun,split_args=list(var_limit=.05))
```

combine_fun must return a single numeric value. Unlike split_fun, combine_fun is allowed to return NA values. See Examples for an example of a custom combine function.

Note that the provided splitting and combining functions are written in C++. So while we could define an R function to perform splitting based on the range, the C++ version will run much faster. Custom R functions will run slower than the provided C++ functions.

Creating a quadtree using a template

This function also allows users to create a quadtree using another quadtree as a "template" (via the template_quadtree parameter). The structure of the new quadtree will be identical to that of the template quadtree, but the values of the cells will be derived from the raster used to create the new quadtree. The rasters used to make the template quadtree and the new quadtree should have the exact same extent and dimensions - in addition the exact same 'expansion method' (i.e. the method specified by adj_type) should be used to create both quadtrees.

Other parameters

There are a few other parameters that control various aspects of the quadtree creation process.

The max_cell_length and min_cell_length parameters let the user specify the range of allowable cell sizes. If max_cell_length is not NULL, then the maximum cell size in the resulting quadtree will be max_cell_length. This essentially forces any quadrants larger than max_cell_length to split. The one exception is that a quadrant that contains entirely NA values will not be split. Similarly, the min_cell_length parameter can be used to define a minimum side length for all cells, such that a quadrant cannot be split if its children would be smaller than min_cell_length.

The split_if_any_NA and split_if_all_NA parameters control how NA values are handled. If split_if_any_NA is TRUE (the default), a quadrant will be split if any of the values are NA. This ensures that rasters with irregular shapes maintain their shape in the resulting quadtree representation. If FALSE, quadrants with NAs are not automatically split - note that this can produce unexpected results if the raster is irregularly shaped. split_if_all_NA controls what happens when a quadrant consists entirely of NA values. If FALSE (the default), these quadrants are not split. If TRUE, these quadrants are automatically split, which results in quadrants with all NA values being split to the smallest possible cell size.

Examples

```
library(raster)
```

```
# retrieve the sample data
data(habitat)
rast = habitat

####################################
# using 'adj_type'
####################################

# create quadtree using the 'expand' method - automatically adds NA cells to
# bring the dimensions to 128 x 128 before creating the quadtree
qt1 = qt_create(rast, split_threshold = .15, split_method = "range", adj_type="expand")
qt_plot(qt1) #plot the quadtree
qt_plot(qt1, crop=TRUE, na_col=NULL) #we can use 'crop=TRUE' and 'na_col=NULL'
# if we don't want to see the padded NA's

# create quadtree using the 'resample' method - we'll resample to 128 since it's a power of 2

# first we'll do it WITHOUT adding NAs to the shorter dimension, which
# will result in non-square cells
qt2 = qt_create(rast, split_threshold = .15, split_method = "range",
   adj_type="resample", resample_n_side = 128, resample_pad_NAs=FALSE)
qt_plot(qt2)
qt_plot(qt2, crop=TRUE, na_col=NULL)

# now we'll add 'padding' NAs so that the cells of the quadtree are square
qt3 = qt_create(rast, split_threshold = .15, split_method = "range",
   adj_type="resample", resample_n_side = 128)
qt_plot(qt3)
qt_plot(qt3, crop=TRUE, na_col=NULL)

####################################
# using 'max_cell_length' and 'min_cell_length'
####################################

# we can use the 'max_cell_length' and 'min_cell_length' parameters to control the
# maximum and minimum cell sizes
qt4 = qt_create(rast, split_threshold = .15, split_method = "range",
   max_cell_length = 1000, adj_type="expand")
qt5 = qt_create(rast, split_threshold = .15, split_method = "range",
   min_cell_length = 1000, adj_type="expand")

par(mfrow=c(1,3))
qt_plot(qt1, crop=TRUE, na_col=NULL, main="no cell length restrictions")
qt_plot(qt4, crop=TRUE, na_col=NULL, main="max cell length = 1000")
qt_plot(qt5, crop=TRUE, na_col=NULL, main="min cell length = 1000")
par(mfrow=c(1,1))

####################################
# using 'split_if_any_NA' and 'split_if_all_NA'
####################################

# split quadrants with all NA values - this will result in NA quadrants being split
# to the smallest possible cell size
qt6 = qt_create(rast, split_threshold=.15, split_method="range", split_if_all_NA=TRUE)
# don't force quadrants with NA cells to automatically split - note that this
# can produce rather unexpected results (see the plot of qt7)
```

```
qt7 = qt_create(rast, split_threshold=.15, split_method="range", split_if_any_NA=FALSE)
# 'split_if_any_NA=FALSE' can be used in conjunction with 'max_cell_size' to
# avoid tiny cells on the border of an irregularly shaped raster
qt8 = qt_create(rast, split_threshold=.15, split_method="range",
   split_if_any_NA=FALSE, max_cell_length=1000)

par(mfrow=c(1,3))
qt_plot(qt6, border_lwd=.4)
qt_plot(qt7)
qt_plot(qt8)
par(mfrow=c(1,1))

####################################
# using 'split_method' and 'combine_method'
####################################

# use the standard deviation instead of the range
qt9 = qt_create(rast, split_threshold=.1, split_method = "sd")
# use the min to aggregate values rather than the mean
qt10 = qt_create(rast, split_threshold=.1, split_method = "sd", combine_method="min")

# compare the two quadtrees - note that their structures are identical
par(mfrow=c(1,2))
qt_plot(qt9, crop=TRUE, na_col=NULL, main="split_method='sd', combine_method='mean'", zlim=c(0,1))
qt_plot(qt10, crop=TRUE, na_col=NULL, main="split_method='sd', combine_method='min'", zlim=c(0,1))
par(mfrow=c(1,1))

####################################
# using custom split and combine functions
####################################

# 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))
  }
}

qt11 = qt_create(rast, split_method="custom", split_fun=split_fun,
                 split_args=list(threshold=.8))
qt_plot(qt11, crop=TRUE, na_col=NULL, border_lwd=.5)

# custom combine function - if the mean of the values is less than
# 'threshold', set the cell value to 'low_val'. If it's greater
# than 'threshold', set the cell value to 'high_val'
cmb_fun = function(vals, args){
  if(any(is.na(vals))){
    return(NA)
  }
  if(mean(vals) < args$threshold){
    return(args$low_val)
  } else {
    return(args$high_val)
  }
}
```

```
qt12 = qt_create(rast, split_threshold = .1, split_method="range", combine_method="custom",
            combine_fun = cmb_fun, combine_args = list(threshold=.5, low_val=0, high_val=1))
qt_plot(qt12, crop=TRUE, na_col=NULL)

# note that the split and combine functions are required to have an 'args'
# parameter, but they don't have to use it
cmb_fun2 = function(vals, args){
  return(max(vals) - min(vals))
}

qt13 = qt_create(rast, split_threshold = .1, split_method = "range",
                 combine_method="custom", combine_fun = cmb_fun2)
qt_plot(qt13, crop=TRUE, na_col=NULL, border_lwd=.5)

####################################
# using template quadtrees
####################################

data(habitat_roads)
template = habitat_roads

# has the exact same extent and resolution as 'rast'
# this raster has 1 where a road occurs and 0 otherwise
plot(template)

# we can use a custom function so that a quadrant is split if it contains any 1's
split_if_one = function(vals, args){
  if(any(vals == 1, na.rm=TRUE)) return(TRUE)
  return(FALSE)
}
qt_template = qt_create(template, split_method="custom", split_fun=split_if_one,
   split_threshold=.01)
# now use the template to create a quadtree from 'rast'
qt14 = qt_create(rast, template_quadtree = qt_template)

par(mfrow=c(1,2))
qt_plot(qt_template, crop=TRUE, na_col=NULL, border_lwd=.5)
qt_plot(qt14, crop=TRUE, na_col=NULL, border_lwd=.5)
par(mfrow=c(1,1))
```

`qt_extent` *Get the extent of a quadtree*

Description

Gets the extent of the quadtree as an 'extent' object (from the raster package)

Usage

```
qt_extent(quadtree, original = FALSE)
```

Arguments

quadtree	quadtree object
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 NA rows/columns were added to pad the dimensions.

Value

Returns an 'extent' object (from the 'raster' package)

Examples

```
data(habitat)
rast = habitat

# create a quadtree
qt = qt_create(rast, split_threshold=.1, adj_type="expand")

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

ext
ext_orig

# plot them
qt_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)
```

---

qt_extract *Extract the values of a quadtree at the given locations*

Description

Extract the cell values and optionally the cell extents

Usage

```
qt_extract(quadtree, pts, extents = FALSE)
```

Arguments

quadtree	A quadtree object
pts	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

The return type depends on the value of `extents`.

If `extents = FALSE`, the function returns a numeric vector corresponding to the values at the points represented by `pts`. If a point falls within the quadtree extent and the corresponding cell is `NA`, or if the point falls outside of the extent, `NaN` is returned.

If `extents = TRUE`, the function returns a 5-column numeric matrix providing the extent of each cell along with the cell's value. The 5 columns are, in this order: `xmin`, `xmax`, `ymin`, `ymax`, `value`. If a point falls in a `NA` cell, the cell extent is still returned but `value` will be `NaN`. If a point falls outside of the quadtree, all values will be `NaN`.

Examples

```
data(habitat)
rast = habitat

# create quadtree
qt1 = qt_create(rast, split_threshold=.1, adj_type="expand")
qt_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 = qt_extract(qt1,pts)

# plot the quadtree and the points
qt_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
qt_extract(qt1,pts,extents=TRUE)
```

---

qt_find_lcp *Find the LCP between two points on a quadtree*

Description

Finds the least cost path (LCP) between two points, using a quadtree as a resistance surface

Usage

```
qt_find_lcp(lcp_finder, end_point, use_original_end_points = FALSE)
```

Arguments

`lcp_finder`	the LCP finder object returned from `qt_lcp_finder`
`end_point`	numeric vector with two elements - the x and y coordinates of the the destination point

use_original_end_points
: boolean; by default the start and end points of the returned path are not the points given by the user but instead the centroids of the cells that those points fall in. If this parameter is set to TRUE the start and end points (representing the cell centroids) are replaced with the actual points specified by the user. Note that this is done after the calculation and has no effect on the path found by the algorithm.

Details

See qt_lcp_finder for more information on how the LCP is found

Value

qt_find_lcp returns a five column matrix representing the least cost path. It has the following columns:

- x: x coordinate of this point
- y: y coordinate of this point
- 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

If no path is possible between the two points, a 0-row matrix with the previously described columns is returned. Also, note that when creating the LCP finder object using qt_lcp_finder, NULL will be returned if start_point falls outside of the quadtree. If NULL is passed to the lcp_finder parameter, a 0-row matrix is returned.

IMPORTANT NOTE: the use_original_end_points options ONLY changes the x and y coordinates of the first and last points - it doesn't change the cost_tot or dist_tot columns. This means that even though the start and end points have changed, the cost_tot and dist_tot columns still represent the cost and distance using the cell centroids of the start and end cells.

See Also

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

Examples

```
library(raster)

# create a quadtree
data(habitat)
rast = habitat
qt = qt_create(rast, split_threshold = .1, adj_type="expand")
qt_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 = qt_lcp_finder(qt, start_pt)
path = qt_find_lcp(lcpf, end_pt)

# plot the LCP
qt_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: see "Examples" in ?qt_lcp_finder for more examples
```

`qt_find_lcps` *Find LCPs to surrounding points*

Description

Calculates the LCPs to surrounding points. Constraints can be placed on the LCPs, so that only LCPs that are less than some specified cost-distance are returned. In addition to cost, LCPs can be constrained by distance or cost-distance + distance (see Details).

Usage

```
qt_find_lcps(lcp_finder, limit_type = "none", limit = NULL)
```

Arguments

`lcp_finder`	the LCP finder object returned from `qt_lcp_finder`
`limit_type`	character; one of "none", "costdistance", or "costdistance+distance". Abbreviations can also be used - "n", "cd", "cd+d". Specifies what variable (if any) to constrain the paths on
`limit`	numeric; the maximum value allowed for the variable specified by `limit_type`. If `limit_type` is "none", this parameter does not need to be provided

Details

When `limit_type` is "none", least-cost paths are calculated to all cells within the search area (as defined by `xlims` and `ylims` in `qt_lcp_finder()`).

When `limit_type` is "costdistance", all paths found will have a cost-distance less than `limit`. As described in the documentation for `qt_lcp_finder`, the cost-distance is the cost of the cell times the length of the segment that falls within the cell. Because all edges connect two cells, the segments that fall in each cell are first calculated and then added.

When `limit_type` is "costdistance+distance", the cost-distance and distance are added together. This is primarily for use when the quadtree contains "resistance" values between 0 and 1. When the resistance values are below 1, the cost-distance will always be lower than the distance - in fact, if there are resistance values of 0, the total cost of a path could be 0. Adding the cost-distance and the cost ensures that if there is no resistance, the cost of the path will be equal to the distance traveled. Thus, if the limit is set at 15, the longest possible path would be 15 (which would only occur if it travels over cells that all have a resistance of 0) and would decrease as the resistance of the underlying surface increases. Note that an equivalent method would be to simply add 1 to all the values so they fall between 1 and 2, and then use "costdistance" as the limiting variable.

A very important note to make is that once the LCP tree is calculated, it never gets smaller. The implication of this is that great care is needed if using a LCP finder more than once - in fact, this should

be avoided. For example, I could use `qt_find_lcps(lcp_finder,limit_type="cd",limit=10)` to find all LCPs that have a cost-distance less than 10. I could then use `qt_lcp_summary` to view all cells that are reachable within 10 cost units. However, if I then run `qt_find_lcps(lcp_finder,limit_type="cd",limi` to find all LCPs that have a cost-distance less than 5, the underlying LCP network **will remain unchanged**. That is, if I run `qt_lcp_summary` on `lcp_finder`, it will **return paths with a cost-distance greater than 5**, since we had previously used `lcp_finder` to find paths less than 10. As mentioned before, this happens because the underlying data structure only ever adds nodes, and never removes nodes.

Value

Returns a matrix summarizing each LCP found. `qt_lcp_summary` is used to generate this matrix - see the help for that function for details on the return matrix. Note that this function does **not** return the full paths to each point - however, each of the paths summarized in the output matrix has already been calculated, and can be retrieved using `qt_find_lcp()` (without having to recalculate the path, since it's already been calculated).

See Also

`qt_lcp_finder()` creates the LCP finder object used as input to this function. `qt_find_lcp()` returns the LCP between two points. `qt_lcp_summary()` outputs a summary matrix of all LCPs that have been calculated so far.

Examples

```
library(raster)

# create a quadtree
data(habitat)
rast = habitat
qt = qt_create(rast, split_threshold=.1, adj_type="expand")

start_pt = c(19000,25000)

####################################
# using the 'limit_type' parameter
####################################

# find all LCPs
lcpf1 = qt_lcp_finder(qt, start_pt)
paths1 = qt_find_lcps(lcpf1, limit_type="none")

# limit LCPs by cost-distance
lcpf2 = qt_lcp_finder(qt, start_pt)
paths2 = qt_find_lcps(lcpf2, limit_type="cd", limit=5000)

# limit LCPs by cost-distance + distance
lcpf3 = qt_lcp_finder(qt, start_pt)
paths3 = qt_find_lcps(lcpf3, limit_type="cd+d", limit=5000)

# Now plot the reachable cells, by method - we'll plot the centroids of the reachable cells
qt_plot(qt, crop=TRUE, na_col=NULL, border_col="gray60", col=c("white", "gray30"),
   main="reachable cells, by 'limit_type'")
points((paths1$xmin + paths1$xmax)/2, (paths1$ymin + paths1$ymax)/2, pch=16, col="black", cex=1.4)
points((paths2$xmin + paths2$xmax)/2, (paths2$ymin + paths2$ymax)/2, pch=16, col="red", cex=1.1)
points((paths3$xmin + paths3$xmax)/2, (paths3$ymin + paths3$ymax)/2, pch=16, col="blue", cex=.8)
```

```
points(start_pt[1], start_pt[2], bg="green", col="black", pch=24, cex=1.5)
legend("topright", title="limit_type", legend=c("none", "cd", "cd+d"), pch=c(16,16,16),
   col=c("black", "red", "blue"), pt.cex=c(1.4,1.1,.8))
legend("topleft", legend="start point", pch=24, pt.cex=1.5, pt.bg="green", col="black")

###################################
# An example of what NOT to do
###################################

lcpf4 = qt_lcp_finder(qt, start_pt)
paths4a = qt_find_lcps(lcpf4, limit_type="cd", limit=5000)
paths4b = qt_find_lcps(lcpf4, limit_type="cd", limit=500)
# ^^^ DON'T DO THIS! ^^^ (don't try to reuse the lcp finder to find *shorter* paths)

range(paths4b$lcp_cost) # returns paths with cost greater than 500 even though
                        # we set the limit at 500!!!

# if we want to find shorter paths, we need to create a new LCP finder
lcpf5 = qt_lcp_finder(qt, start_pt)
paths5 = qt_find_lcps(lcpf5, limit_type="cd", limit=500)
range(paths5$lcp_cost)
```

`qt_lcp_finder` *Create an object for finding LCPs on a quadtree*

Description

This function creates an object that can then be used by `qt_find_lcp` and `qt_find_lcps` to find least-cost paths (LCPs).

Usage

```
qt_lcp_finder(quadtree, start_point, xlims = NULL, ylims = NULL)
```

Arguments

`quadtree`	a quadtree object to be used as a resistance surface
`start_point`	numeric vector with 2 elements - the x and y coordinates of the starting point of the path(s)
`xlims`	numeric vector with 2 elements - paths will be constrained so that all points fall within the min and max x coordinates specified in `xlims`. If `NULL` the x limits of `quadtree` are used
`ylims`	same as `xlims`, but for y

Details

This function creates an object that can then be used by `qt_find_lcp` or `qt_find_lcps` to calculate least-cost paths.

Dijkstra's algorithm is used to find least-cost-paths (LCPs) on a network. The network used in this case consists of the cell centroids (nodes) and the neighbor connections (edges). The cost of each edge is taken as the length of the edge times the weight - because the edge travels between two cells, the cost of the edge is weighted by the distance that falls within each cell.

Dijkstra's algorithm essentially builds a tree data structure, where the starting node is the root of the tree. It iteratively builds the tree structure, and in each iteration it adds the node that is "closest" to the current tree - that is, it chooses the node which is easiest to get to. The result is that even if only one LCP is desired, LCPs to other nodes are also calculated in the process.

The LCP finder object internally stores the results as a tree-like structure. Finding the LCP to a given point can be seen as a two-step process. First, construct the tree structure as described above. Second, starting from the destination node, travel up the tree, keeping track of the sequence of nodes passed through, until the root (the starting node) is reached. This sequence of nodes (in reverse, since we started from the destination node) is the LCP to that point.

Once the tree has been constructed, LCPs can be found to any of the of the child nodes without further computation. This allows for efficient computation of multiple LCPs. The LCP finder saves state - whenever an LCP is asked to be calculated, it first checks whether or not a path has been found to that node already - if so, it simply returns the path using the process described above. If not, it builds out the existing tree until the desired node has been reached.

Two slightly different ways of calculating LCPs are provided that differ in their stop criteria - that is, the condition on which the tree stops being built. `qt_find_lcp()` finds a path to a specific point. As soon as that node has been added to the tree, the algorithm stops and the LCP is returned. `qt_find_lcps()` doesn't use a destination point - instead, the tree continues to be built until the paths exceed a given cost-distance, depending on which one the user selects. In addition, this constraint can be ignored in order to find all LCPs within the given set of nodes. See the documentation for those two functions for more details.

An important note is that because of the heterogeneous nature of a quadtree, the paths found likely won't reflect the 'true' least cost path. This is because treating the centroids of the cells as the nodes introduces some distortion, especially with large cells.

Also note that the `xlims` and `ylims` arguments in `qt_lcp_finder()` can be used to restrict the search space to the rectangle defined by xlims and ylims. This speeds up the computation of the LCP by limiting the number of cells considered.

Another note is that an LCP finder object is specific to a given starting point. If a new starting point is used, a new LCP finder is needed.

Value

returns an LCP finder object. If `start_point` falls outside of the quadtree extent, `NULL` is returned.

See Also

`qt_find_lcp()` returns the LCP between two points. `qt_find_lcps()` finds all LCPs whose cost-distance is less than some value. `qt_lcp_summary()` outputs a summary matrix of all LCPs that have been calculated so far.

Examples

```
library(raster)

####################################
# create a quadtree
####################################

data(habitat)
rast = habitat
qt = qt_create(rast, split_threshold = .1, adj_type="expand")
qt_plot(qt, crop=TRUE, na_col=NULL, border_lwd=.4)
```

```
####################################
# basic usage
####################################

# --------------------
# find the LCP to a single point
# --------------------
start_pt1 = c(6989,34007)
end_pt1 = c(33015,38162)

# create the LCP finder object and find the LCP
lcpf1 = qt_lcp_finder(qt, start_pt1)
path1 = qt_find_lcp(lcpf1, end_pt1)

# plot the LCP
qt_plot(qt, crop=TRUE, na_col=NULL, border_col="gray30", border_lwd=.4)
points(rbind(start_pt1, end_pt1), pch=16, col="red")
lines(path1[,1:2], col="black")

# --------------------
# find all LCPs
# --------------------
# calculate all LCPs
paths_summary1 = qt_find_lcps(lcpf1, limit_type="none")
# retrieve each individual LCP
all_paths1 = lapply(1:nrow(paths_summary1), function(i){
  row_i = paths_summary1[i,]
  pt_i = with(row_i, c((xmin+xmax)/2, (ymin+ymax)/2))
  return(qt_find_lcp(lcpf1,pt_i))
})

#plot all the LCPs
qt_plot(qt, crop=TRUE, na_col=NULL, border_col="gray30", border_lwd=.4)
lapply(all_paths1, lines)
points(start_pt1[1], start_pt1[2], bg="red", col="black", pch=21, cex=1.2)

# --------------------
# find all cells reachable under a given threshold
# --------------------
start_pt2 = c(19000,25000)
limit = 5000

# create the LCP finder object and find all the valid LCPs
lcpf2 = qt_lcp_finder(qt, start_pt2)
# we could use limit_type="none" if we wanted to find LCPs to ALL cells
paths_summary2 = qt_find_lcps(lcpf2, limit_type="cd", limit=limit)

# plot the centroids of the reachable cells
qt_plot(qt, main=paste0("reachable cells; cost+distance < ", limit), crop=TRUE,
        na_col=NULL, border_col="gray60")
with(paths_summary2, points((xmin+xmax)/2, (ymin+ymax)/2, pch=16, col="black", cex=.4))
points(start_pt2[1], start_pt2[2], col="red", pch=16)

# --------------------
# limiting the search area
# --------------------
```

```
# define the search area
box_length = 7000
xlims = c(start_pt2[1] - box_length/2, start_pt2[1] + box_length/2)
ylims = c(start_pt2[2] - box_length/2, start_pt2[2] + box_length/2)

# find the LCPs to all the cells inside the search area
lcpf3 = qt_lcp_finder(qt, start_pt2, xlims=xlims, ylims=ylims)
paths_summary3 = qt_find_lcps(lcpf3, limit_type="none")

# retrive each LCP
all_paths3 = lapply(1:nrow(paths_summary3), function(i){
  row_i = paths_summary3[i,]
  pt_i = with(row_i, c((xmin+xmax)/2, (ymin+ymax)/2))
  return(qt_find_lcp(lcpf3,pt_i))
})

# plot the results
qt_plot(qt, crop=TRUE, na_col=NULL, border_col="gray60")
with(paths_summary3, points((xmin+xmax)/2, (ymin+ymax)/2, pch=16, col="black", cex=.4))
lapply(all_paths3, lines)
rect(xlims[1], ylims[1], xlims[2], ylims[2], border="red", lwd=2)
points(start_pt2[1], start_pt2[2], col="red", pch=16)


####################################
# a larger example to demonstrate run time
####################################

#generate a large matrix of random values between 0 and 1
nrow = 570
ncol = 750
rast = raster(matrix(runif(nrow*ncol), nrow=nrow, ncol=ncol), xmn=0, xmx=ncol, ymn=0, ymx=nrow)

#make the quadtree
qt1 = qt_create(rast, split_threshold = .9, adj_type="expand")

#get the LCP finder
lcpf = qt_lcp_finder(qt1, c(1,1))

# the LCP finder saves state. So finding the path the first time requires
# computation, and takes longer, but running it again is nearly instantaneous
system.time(qt_find_lcp(lcpf, c(740,560))) #takes longer
system.time(qt_find_lcp(lcpf, c(740,560))) #runs MUCH faster

# in addition, because of how Dijkstra's algorithm works, the LCP finder also
# found many other LCPs in the course of finding the first LCP, meaning that
# subsequent LCP queries for different destination points will be much faster
# (since the LCP finder saves state)
system.time(qt_find_lcp(lcpf, c(740,1)))
system.time(qt_find_lcp(lcpf, c(1,560)))

# now save the paths so we can plot them
path1 = qt_find_lcp(lcpf, c(740,560))
path2 = qt_find_lcp(lcpf, c(740,1))
path3 = qt_find_lcp(lcpf, c(1,560))

# plot the paths
```

```
qt_plot(qt1, crop=TRUE, border_col="transparent", na_col=NULL)
lines(path1[,1:2])
lines(path2[,1:2], col="red")
lines(path3[,1:2], col="blue")
```

`qt_lcp_summary` *Show a summary matrix of all LCPs currently calculated*

Description

Given an LCP finder object, returns a matrix that summarizes all of the LCPs that have already been calculated by the LCP finder.

Usage

```
qt_lcp_summary(lcp_finder)
```

Arguments

`lcp_finder`	an LCP finder object created using `qt_lcp_finder()`

Details

Note that this function returns **all** of the paths that have been calculated. As explained in the documentation for `qt_lcp_finder()`, finding one LCP likely involves finding other LCPs as well. Thus, even if the LCP finder 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 9-column matrix with one row for each LCP (and therefore one row per 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

`qt_lcp_finder()` creates the LCP finder object used as input to this function. `qt_find_lcp()` returns the LCP between two points. `qt_find_lcps()` calculates all LCPs whose cost-distance is less than some value.

Examples

```
library(raster)

# create a quadtree
data(habitat)
rast = habitat
qt = qt_create(rast, 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 = qt_lcp_finder(qt, start_pt)
lcp = qt_find_lcp(lcpf, end_pt)

# retrieve ALL the paths that have been calculated
paths = qt_lcp_summary(lcpf)

# put points in each of the cells to which an LCP has been calculated
qt_plot(qt, crop=TRUE, na_col=NULL, border_col="gray60")
points((paths$xmin + paths$xmax)/2, (paths$ymin + paths$ymax)/2, pch=16, col="black", cex=.4)
points(rbind(start_pt, end_pt), col=c("red", "blue"), pch=16)
```

`qt_plot` *Plot a quadtree object*

Description

Plot a quadtree object

Usage

```
qt_plot(
  quadtree,
  add = FALSE,
  col = NULL,
  alpha = 1,
  nb_line_col = NULL,
  border_col = "black",
  border_lwd = 1,
  xlim = NULL,
  ylim = NULL,
  zlim = NULL,
  crop = FALSE,
  na_col = "white",
  adj_mar_auto = 6,
  legend = TRUE,
  legend_args = list(),
  ...
)
```

Arguments

quadtree	a quadtree object
add	boolean; if TRUE, the quadtree 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.
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
border_lwd	numeric; the line width of the cell borders - passed to the 'lwd' parameter of the 'rect' function
xlim	two element numeric vector; optional; defines the minimum and maximum values of the x axis.
ylim	two element numeric vector; optional; defines the minimum and maximum values of the y axis.
zlim	two element numeric vector; optional; defines how the colors are assigned to the cell values. If this value is NULL (the default), it uses the min and max cell values. In this case, the first color in col corresponds to the lowest cell value and the last color is col corresponds to the highest cell value. 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.
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
adj_mar_auto	numeric; if not NULL, it 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). 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.

Examples

```
library(raster)
data(habitat)
rast = habitat
# create quadtree
```

```
qt1 = qt_create(rast, split_threshold=.1, adj_type="expand")

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

# default - no additional parameters provided
qt_plot(qt1)

####################################
# 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
qt_plot(qt1, crop=TRUE, main="cropped")

# crop and don't plot NA cells
qt_plot(qt1, crop=TRUE, na_col=NULL, main="cropped")

# use 'xlim' and 'ylim' to zoom in on an area
qt_plot(qt1, xlim = c(10000,20000), ylim = c(20000,30000), main="zoomed in")

####################################
# COLORS
####################################

# change border color and width
qt_plot(qt1, border_col="transparent") #no borders
qt_plot(qt1, border_col="gray60") #gray borders
qt_plot(qt1, border_lwd=.3) #change line thickness of borders

# change color palette
qt_plot(qt1, col=c("blue", "yellow", "red"))
qt_plot(qt1, col=hcl.colors(100))
qt_plot(qt1, col=c("black", "white"))

# change color transparency
qt_plot(qt1, alpha=.5)
qt_plot(qt1, col=c("blue", "yellow", "red"), alpha=.5)

# change color of NA cells
qt_plot(qt1, na_col="lavender")

# don't plot NA cells at all
qt_plot(qt1, na_col=NULL)

# change 'zlim'
qt_plot(qt1, zlim=c(0,5))
qt_plot(qt1, zlim=c(.2,.7))

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

```
# plot all neighbor connections
qt_plot(qt1, nb_line_col="black", border_col="gray60")

# don't plot connections to NA cells
qt_plot(qt1, nb_line_col="black", border_col="gray60", na_col=NULL)

####################################
# LEGEND
####################################

# no legend
qt_plot(qt1, legend=FALSE)

# increase right margin size
qt_plot(qt1, adj_mar_auto=10)

# use 'legend_args' to customize the legend
qt_plot(qt1, adj_mar_auto=10, legend_args=list(lgd_ht_pct=.8, bar_wd_pct=.4))
```

---

qt_proj4string *Retrieve the proj4string of a quadtree*

Description

Retrieve the proj4string of a quadtree

Usage

```
qt_proj4string(quadtree)
```

Arguments

quadtree	a quadtree object

Value

A character containing the proj4string

---

qt_read *Read/write a quadtree*

Description

Read/write a quadtree

Usage

```
qt_read(filepath)

qt_write(quadtree, filepath)
```

Arguments

filepath	character; the filepath to read from or write to
quadtree	quadtree object; the quadtree to write

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').

Note that typically the quadtree isn't particularly space-efficient - it's not uncommon for a quadtree file to be larger than the original raster file (although, of course, this depends on how 'coarse' the quadtree is in relation to the original raster). This is likely because the quadtree has to store much more information about each cell (the x and y limits, its value, pointers to its neighbors, among other things) while a raster can store only the value since the coordinates of the cell can be determined from the knowledge of the extent and the dimensions of the raster.

It's entirely possible that a quadtree implemention could be written that is MUCH more space efficient. However, this was not the primary goal when creating this package.

Examples

```
## Not run:
qt = qt_read("path/to/quadtree.qtree")
qt_write(qt, "path/to/newQuadtree.qtree")

## End(Not run)
```

---

qt_set_values *Change values of quadtree cells*

Description

Given 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

```
qt_set_values(quadtree, pts, vals)
```

Arguments

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

Details

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

Also note that the structure of the quadtree will not be changed - only the cell values will change.

Value

No return value

Examples

```
data(habitat)
rast = habitat

# create a quadtree
qt = qt_create(rast, split_threshold = .1)

par(mfrow=c(1,2))
qt_plot(qt, main="original")

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

# plot it out to see what happened
qt_plot(qt, main="after modification")
```

---

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

Description

The quadtree class is the underlying C++ data structure used in the quadtree package. Note that the average user should not need to use these functions - the functions prefixed with 'qt_' are R wrapper functions that provide access to the many of the member functions.

Details

Note that the name of the class as it is defined is actually 'QuadtreeWrapper', but it is exposed to R simply as 'quadtree'. Thus, this class is defined in the 'QuadtreeWrapper.h' and QuadtreeWrapper.cpp' files. As the name suggests 'QuadtreeWrapper' is a wrapper for the 'Quadtree' class. The 'Quadtree' class was written to be completely independent of R and Rcpp, and thus operates as a stand-alone C++ class. 'QuadtreeWrapper' contains an instance of a 'Quadtree' object and contains the code necessary to interface between R and C++.

Each member function exposed to R is described below. Note that when a function directly corresponds with one of the 'qt_*' functions, the user is referred to the documentation for that function for more details (in order to avoid replication of documentation)

Fields

`constructor`

- **Description**: default constructor. Can be used as follows: `qt = new(quadtree)`
- **Parameters**: none
- **Returns**: an empty quadtree object

`constructor` • **Description**: constructor. Can be used as follows: `qt = new(quadtree,xlims,ylims,maxCellLeng` Used in `qt_create()`. The parameters for this constructor correspond with the similarly named parameters in `qt_create()` - 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`: 2-element numeric vector
  - `ylims`: 2-element numeric vector
  - `maxCellLength`: 2-element numeric vector - first element is for the x dimension, second is for the y dimension
  - `minCellLength`: 2-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 does not require an instance of `Rcpp_quadtree` to be called. `qt_read()` is a wrapper for this function - see its documentation for more details.

- **Parameters**:
  - `filePath`: string; the file to read from
- **Returns**: a `Rcpp_quadtree` object

`asList` • **Description**: outputs a list containing details about each cell. `qt_as_data_frame()` is a wrapper for this function that rbinds 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 `qt_as_data_frame()` - see that help page for details.

`copy` • **Description**: returns a deep copy of a quadtree. `qt_copy()` is a wrapper for this function - see the documentation for that function for more details.

- **Parameters**: none
- **Returns**: a quadtree object

`createTree` • **Description**: constructs the quadtree from a matrix. `qt_create()` is a wrapper for this function and should be used to create quadtrees. The parameters correspond with the similarly named parameters in `qt_create` - 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`: quadtree
- **Returns**: void - no return value

`extent` • **Description**: returns the extent of the quadtree. This is equivalent to `qt_extent(qt,original=FALSE)`

- **Parameters**: none
- **Returns**: 4-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 (as a 'Rcpp_node' object)
- **Parameters**:
  - `x`: double; x coordinate
  - `y`: double; y coordinate
- **Returns**: a 'Rcpp_node' object representing the cell that contains the point

`getCellDetails`
- **Description**: Given points defined by their x and y coordinates, returns a matrix giving details on the cells at each of the points. `qt_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 `qt_extract()` for details about the matrix columns

`getCells`
- **Description**: Given x and y coordinates of points, returns a list of the cells at those points (as 'Rcpp_node' 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 `Rcpp_node` objects corresponding to the x and y coordinates passed to the function

`getNbList`
- **Description**: Returns the neighbor relationships between all cells
- **Parameters**: none
- **Returns**: `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`: the ID, x and y coordinates of the centroid, and cell value for the cell of interest. Note that all of these values of these columns will be same across all rows because they refer to the same cell.
  - `id1`, `x1`, `y1`, `val1`: the ID, x and y coordinates of the centroid, and cell value for each cell that neighbors the cell of interest (i.e. the cell represented by the columns suffixed with '0').
  - `isLowest`: 1 or 0 - whether or not the cell of interest (i.e. the cell represented by the columns suffixed with '0') is a terminal node, where 1 means it is a terminal node (no children) and 0 means it is not a terminal node (has children).

`getShortestPathFinder`
- **Description**: Returns a shortestPathFinder object (i.e. object with class `Rcpp_shortestPathFinder`) that can be used to find least-cost paths on the quadtree. `qt_lcp_finder()` is a wrapper for this function. For details on the parameters see the **Description** of the similarly named parameters in `qt_lcp_finder()`
- **Parameters**:
  - `startPoint`: two element numeric vector
  - `xlims`: two element numeric vector
  - `ylims`: two element numeric vector
- **Returns**: an object with class `Rcpp_shortestPathFinder`

`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. `qt_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 `qt_create()`). Note that this does **not** return the maximum cell size in the quadtree - it returns the maximum *allowable* cell size. 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 in 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 `qt_create()`). Note that this does **not** return the minimum cell size in the quadtree - it returns the minimum *allowable* cell size. 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 in 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**: 2-element numeric vector that given 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 `qt_extent(qt,original=TRUE)`
- **Parameters**: none
- **Returns**: 4-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**: 2-element numeric vector

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

`projection`
- **Description**: Returns the proj4string of the quadtree
- **Parameters**: none
- **Returns**: a string

`root`
- **Description**: Returns the root node of the quadree
- **Parameters**: none

- **Returns**: a Rcpp_node 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 property that records the proj4string of the quadtree
- **Parameters**:
  - `proj4string`: 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. `qt_set_values()` is a wrapper for this function - see its documentation page for more details. `qt_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
  - `newVals`: numeric vector; must be the same length as x and y
- **Returns**: void - no return value

`writeQuadtree`
- **Description**: Writes a quadtree to a file. `qt_write()` 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

`shortestPathFinder-class`

C++ Shortest Path Finder

Description

The 'shortestPathFinder' class is a C++ data structure used to find least-cost paths using a quadtree as a resistance surface. A 'shortestPathFinder' object is essentially defined by a quadtree and a starting point on that quadtree, and it can be used to find least-cost paths from the starting point to any other point on the quadtree. The average user should not need to use any of the methods defined here, as wrapper functions have been defined to perform these methods.

Details

Note that there is no constructor made accessible to R. This is because a 'shortestPathFinder' object is always used "on top of" a quadtree object - to create a 'shortestPathFinder', therefore, use the 'getShortestPathFinder' member function from the 'quadtree' class.

Fields

`getAllPathsSummary` • **Description**: Returns a matrix summarizing all the LCPs calculated so far. `qt_lcp_summary()` 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 `qt_lcp_summary()` for details.

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

- **Parameters**: none
- **Returns**: 4-element numeric vector, in this order: xMin, xMax, yMin, yMax

`getShortestPath` • **Description**: Finds the least-cost path from the starting point to another point. `qt_find_lcp` is a wrapper for this function - see its documentation for more details.

- **Parameters**:
  - `endPoint`: 2-element numeric vector - the point to find a shortest path to
- **Returns**: A matrix representing the least-cost path. See `qt_find_lcp()` for details on the return matrix.

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

- **Parameters**: none
- **Returns**: 2-element numeric vector (x,y)

`makeNetworkAll` • **Description**: Calculates least-cost paths to all cells in the search area. This corresponds to `qt_find_lcps(lcp_finder,limit_type="none")`. See documentation of that function for more details.

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

`makeNetworkCost` • **Description**: Calculates all least-cost paths whose cost-distance is less than a given threshold. This correspnds to `qt_find_lcps(lcp_finder,limit_type="costdistance")`. See documentation of that function for more details

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

`makeNetworkCostDist` • **Description**: Calculates all least-cost paths whose "costdistance + distance" is less than a given threshold. This correspnds to `qt_find_lcps(lcp_finder,limit_type="costdi`. See documentation of that function for more details.

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

Index