Package ‘quadtree’

July 1, 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

LinkingTo Rcpp

RoxygenNote 7.1.1

NeedsCompilation yes

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

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`, `qt_extent_orig`: 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,
  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 to create the color ramp used in the plot.
`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 <div>-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` 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
```

`qt_create` *Create a quadtree from gridded data*

Description

Create a quadtree from gridded data

Usage

```
qt_create(
  x,
  range_limit,
  max_cell_length = NULL,
  adj_type = "expand",
  resample_n_side = NULL,
  extent = NULL,
  proj4string = 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.
`range_limit`	numeric; if the cell values within a quadrant have a range larger than this value, the quadrant is split. See 'Details' for more
`max_cell_length`	double; the maximum size allowed for a quadtree cell. If `NULL` no restrictions are placed on the quadtree cell size. See 'Details' for more

adj_type — character; either 'expand' or 'resample'. See 'Details' for more.

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

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. 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. If no value is provided and x is a matrix, the 'proj4string' of the quadtree is set to NA.

Details

A quadtree is created from a raster by successively dividing the raster/matrix into smaller and smaller cells, with the decision on whether to divide a cell determined by range_limit. Initially, all of the cells in the raster are considered. If the difference between the maximum and minimum cell values exceeds range_limit, the raster is divided into four quadrants - otherwise, the raster is not divided further and the mean of all values in the raster is taken as the value for the resulting cell. Then, the process is repeated for each of those 'child' cells, and then for their children, and so on and so forth, until either range_limit is not exceeded or the smallest possible cell size has been reached.

If a quadrant contains both NA cells and non-NA cells, that quadrant is automatically divided. However, if a quadrant consists entirely of NA cells, that cell is not divided further (even if the cell is larger than max_cell_length).

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.

If max_cell_length is not NA, 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.

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. There are two steps. First, the raster must be made square. 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. In the second step, this raster is then 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. The dimensions should be a power of 2 (see above for an explanation), although other numbers will be accepted (but will trigger a warning).

Examples

```
#create raster of random values
nrow = 57
ncol = 75
rast = raster(matrix(runif(nrow*ncol), nrow=nrow, ncol=ncol), xmn=0, xmx=ncol, ymn=0, ymx=nrow)

#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, range_limit = .9, adj_type="expand")
qt_plot(qt1) #plot the quadtree
qt_plot(qt1, crop=TRUE) #we can use 'crop=TRUE' 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
qt2 = qt_create(rast, range_limit = .9, adj_type="resample", resample_n_side = 128)
qt_plot(qt2)
qt_plot(qt2, crop=TRUE)

#now use the 'max_cell_length' argument to force any cells with sides longer
#than 2 to split
qt3 = qt_create(rast, range_limit = .9, max_cell_length = 2, adj_type="expand")

#compare qt1 (no max cell length) and qt3 (max cell length = 2)
par(mfrow=c(1,2))
qt_plot(qt1,crop=TRUE, main="no max cell length")
qt_plot(qt3,crop=TRUE, main="max cell length = 2")
```

---

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)

qt_extent_orig(quadtree)
```

Arguments

quadtree	quadtree object

Details

`qt_extent` returns the total extent covered by the quadtree.

`qt_extent_orig` returns the extent of the original raster used to create the quadtree, before NA rows/columns were added to pad the dimensions. This essentially represents the extent in which the non-NA data occurs.

Value

Returns an 'extent' object

Examples

```
#create raster of random values
nrow = 57
ncol = 75
rast = raster(matrix(runif(nrow*ncol), nrow=nrow, ncol=ncol), xmn=0, xmx=ncol, ymn=0, ymx=nrow)
qt = qt_create(rast, .9, adj_type="expand")

qt_extent(qt)
qt_extent_orig(qt)
```

`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`, 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`, `NA` is returned. If the point falls outside of the quadtree 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 `NA`. If a point falls outside of the quadtree, all values will be `NaN`.

Examples

```
# create raster of random values
nrow = 57
ncol = 75
rast = raster(matrix(runif(nrow*ncol), nrow=nrow, ncol=ncol), xmn=0, xmx=ncol, ymn=0, ymx=nrow)

# create quadtree
qt1 = qt_create(rast, range_limit = .9, adj_type="expand")

# create points at which we'll extract values
```

```
pts = cbind(-5:15, 45:65)

# plot the quadtree and the points
qt_plot(qt1, border_col="gray60")
points(pts, pch=16,cex=.6)

# extract values only
qt_extract(qt1,pts)

# 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`; `qt_find_lcps`; `qt_lcp_summary`

Examples

```
library(raster)

# ----- create a quadtree
# create raster of random values
nrow = 57
ncol = 75
set.seed(4)
rast = raster(matrix(runif(nrow*ncol), nrow=nrow, ncol=ncol), xmn=0, xmx=ncol, ymn=0, ymx=nrow)

# create quadtree
qt = qt_create(rast, range_limit = .9, adj_type="expand")

# defined our start and end points
start_pt = c(.231,.14)
end_pt = c(74.89,56.11)

# make an LCP finder object
lcpf = qt_lcp_finder(qt, start_pt)

# use the LCP finder object to find the LCP to a certain point
# this path will have the cell centroids as the start and end points
path1 = qt_find_lcp(lcpf, end_pt)
# this path will be identical to path1 except that the start and end points
# will be the user-provided start and end points rather than the cell centroids
path2 = qt_find_lcp(lcpf, end_pt, use_original_end_points = TRUE)

head(path1)
head(path2)

# plot the result
qt_plot(qt, crop=TRUE, border_col="gray60", na_col=NULL)
points(rbind(start_pt, end_pt), pch=16, col="red")
lines(path1[,1:2], col="black", lwd=2.5)
lines(path2[,1:2], col="red", lwd=1)
points(path1, cex=.7, pch=16)
```

qt_find_lcps *Find LCPs to surrounding points*

Description

Finds 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", "distance", or "costdistance+distance". Abbreviations can also be used - i.e "n", "cd", "d", "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 "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 "distance", only distance is considered. Note that this is **not** straight line distance. It is the Euclidean distance of the least-cost path - it ignores the cost value. Thus all the paths found will have a total distance less than `limit`.

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 increase 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.

See Also

qt_lcp_finder, qt_find_lcp, qt_lcp_summary

Examples

```
library(raster)

# ----- create a quadtree
# create raster of random values
nrow = 57
ncol = 75
set.seed(4)
rast = raster(matrix(runif(nrow*ncol), nrow=nrow, ncol=ncol), xmn=0, xmx=ncol, ymn=0, ymx=nrow)

# create quadtree
qt = qt_create(rast, range_limit = .9, adj_type="expand")

start_pt = c(ncol/2,nrow/2)

#--------- 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=18)

#--------- limit LCPs by cost-distance + distance
lcpf3 = qt_lcp_finder(qt, start_pt)
paths3 = qt_find_lcps(lcpf3, limit_type="cd+d", limit=18)

#--------- Now plot the reachable cells, by method
# 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
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",

#----------------------------------------------------
# An example of what NOT to do
#----------------------------------------------------
lcpf5 = qt_lcp_finder(qt, start_pt)
paths5a = qt_find_lcps(lcpf5, limit_type="cd", limit=18)
paths5b = qt_find_lcps(lcpf5, limit_type="cd", limit=5)
#^^^ DON'T DO THIS! ^^^ (don't try to reuse the lcp finder to find *shorter* paths)

nrow(paths5a)
nrow(paths5b) #they're the same length!!!
```

```
range(paths5b$lcp_cost) #returns paths with cost greater than 5!!!

#if we want to find shorter paths, we need to create a new LCP finder
lcpf6 = qt_lcp_finder(qt, start_pt)
paths6 = qt_find_lcps(lcpf6, limit_type="cd", limit=5)
range(paths6$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 LCPs. Dijkstra's algorithm is a network algorithm. 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

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

See Also

`qt_find_lcp()` finds a LCP between two points. `qt_lcp_find_paths()` 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

```
#----------------------------------------------------
# basic usage
#----------------------------------------------------
library(raster)

# ----- create a quadtree
# create raster of random values
nrow = 57
ncol = 75
set.seed(4)
rast = raster(matrix(runif(nrow*ncol), nrow=nrow, ncol=ncol), xmn=0, xmx=ncol, ymn=0, ymx=nrow)

# create quadtree
qt = qt_create(rast, range_limit = .9, adj_type="expand")

# ----- find the LCP to a single point
start_pt1 = c(.231,.14)
end_pt1 = c(74.89,56.11)

# create the LCP finder object
spf1 = qt_lcp_finder(qt, start_pt1)
path1 = qt_find_lcp(spf1, end_pt1)
```

```
qt_plot(qt, crop=TRUE)
points(rbind(start_pt1, end_pt1), pch=16, col="red")
lines(path1[,1:2], col="black", lwd=2.5)
points(path1, cex=.7, pch=16)

# ----- find the LCPs under a given threshold
start_pt2 = c(ncol/2,nrow/2)
# create the LCP finder object
spf2 = qt_lcp_finder(qt, start_pt2)
limit = 10
paths2 = qt_find_lcps(spf2, 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")
points((paths2$xmin + paths2$xmax)/2, (paths2$ymin + paths2$ymax)/2,
       pch=16, col="black")
points(start_pt2[1], start_pt2[2], col="red", pch=16)

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

xlims = c(30.5,50.5)
ylims = c(10.5,50.5)
spf3 = qt_lcp_finder(qt, start_pt2, xlims=xlims, ylims=ylims)
paths3 = qt_find_lcps(spf3, limit_type="none")

qt_plot(qt, crop=TRUE, na_col=NULL, border_col="gray60")
points((paths3$xmin + paths3$xmax)/2, (paths3$ymin + paths3$ymax)/2,
       pch=16, col="black")
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
#----------------------------------------------------

nrow = 570
ncol = 750
rast = raster(matrix(runif(nrow*ncol), nrow=nrow, ncol=ncol), xmn=0, xmx=ncol, ymn=0, ymx=nrow)

qt1 = qt_create(rast, range_limit = .9, adj_type="expand")
spf = 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(spf, c(740,560))) #takes longer
system.time(qt_find_lcp(spf, 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(spf, c(740,1)))
system.time(qt_find_lcp(spf, c(1,560)))
```

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

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`	the LCP finder object returned from `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, qt_find_lcp, qt_find_lcps #' @examples library(raster)

—– create a quadtree # create raster of random values nrow = 57 ncol = 75 set.seed(4) rast = raster(matrix(runif(nrow*ncol), nrow=nrow, ncol=ncol), xmn=0, xmx=ncol, ymn=0, ymx=nrow)

create quadtree qt = qt_create(rast, range_limit = .9, adj_type="expand")

start_pt = c(ncol/2,nrow/2)

#——— find all LCPs lcpf = qt_lcp_finder(qt, start_pt) paths = qt_find_lcps(lcpf, limit_type="cd+d", limit=10) paths

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") points(start_pt[1], start_pt[2], col="red", pch=16)

---

qt_plot — *Plot a quadtree object*

Description

Plot a quadtree object

Usage

```
qt_plot(
  qt,
  colors = NULL,
  nb_line_col = NULL,
  border_col = "black",
  xlim = NULL,
  ylim = NULL,
  crop = FALSE,
  na_col = "white",
  adj_mar_auto = 6,
  legend = TRUE,
  legend_args = list(),
  ...
)
```

Arguments

qt	a quadtree object
colors	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.
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
xlim	two element numeric vector; defines the minimum and maximum values of the x axis.

ylim	two element numeric vector; defines the minimum and maximum values of the y axis.
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 the two required parameters to add_legend, zlim and cols, are supplied automatically, so if the list contains elements named zlim or cols, they will be ignored.
...	arguments passed to the default plot function

Details

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

Examples

```
# create raster of random values
nrow = 57
ncol = 75
set.seed(2)
rast = raster(matrix(runif(nrow*ncol), nrow=nrow, ncol=ncol), xmn=0, xmx=ncol, ymn=0, ymx=nrow)

# create quadtree
qt1 = qt_create(rast, range_limit = .9, 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")

# use 'xlim' and 'ylim' to zoom in on an area
qt_plot(qt1, xlim = c(30,50), ylim = c(10,20), main="zoomed in")

# -----------
# COLORS
```

```
# -----------

# change border color
qt_plot(qt1, border_col="transparent") #no borders
qt_plot(qt1, border_col="gray60")

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

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

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

# -----------
# 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, crop=TRUE, 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 the quadtree.

Examples

```
qt = qt_read("path/to/quadtree.qtree")
qt_write(qt, "path/to/newQuadtree.qtree")
```

Index