Package 'quadtree'

April 7, 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

R topics documented:

---

quadtree-package   *A short title line describing what the package does*

Description

A more detailed description of what the package does. A length of about one to five lines is recommended.

Details

This section should provide a more detailed overview of how to use the package, including the most important functions.

Author(s)

Your Name, email optional.

Maintainer: Your Name <your@email.com>

References

This optional section can contain literature or other references for background information.

See Also

Optional links to other man pages

Examples

```
## Not run:
   ## Optional simple examples of the most important functions
   ## These can be in \dontrun{} and \donttest{} blocks.

## End(Not run)
```

---

qt_create   *Create a quadtree from a raster*

Description

Create a quadtree from a raster

Usage

```
qt_create(rast, range_limit, adj_type = "expand", resample_n_side = NA)
```

Arguments

`rast`	a `raster` object
`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
`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

Details

A quadtree is created from a raster by successively dividing the raster 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.

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. 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.

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
qt1 = qt_create(rast, range_limit = .9, adj_type="expand") #automatically adds NA cells to bring the dimensions
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
qt2 = qt_create(rast, range_limit = .9, adj_type="resample", resample_n_side = 128) #resample to 128 since it's
qt_plot(qt2)
qt_plot(qt2, crop=TRUE)
```

---

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

Description

Extract the values of a quadtree at the given locations

Usage

```
qt_extract(quadtree, pts)
```

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

Value

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.

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
qt_extract(qt1,pts)
```

---

qt_lcp_finder *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_lcp_finder(quadtree, start_point, xlims = NULL, ylims = NULL)

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

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
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

These two functions are intended to be used in conjunction with one another. qt_lcp_finder creates the object used to find the LCP(s), and qt_find_lcp uses this object to find the LCP to a given point. See 'Examples' for examples of its usage.

The code is structured this way to minimize the computation needed to compute multiple LCPs from a single point. Dijkstra's algorithm iteratively searches for least cost paths, and in the process of finding one LCP it inevitably finds LCPs to other points. Because of this, we can save computation if we use an object that can save its current state rather than having to replicate our work.

The LCP finder stops its search once it reaches the desired node. However, because state is saved, if another LCP is requested and the LCP to that point has not yet been calculated, the LCP finder starts where it left off rather than starting over again. If the LCP to that point has been calculated, it returns the path without having to perform the algorithm again.

Note, however, that this only applies to the LCPs found using the same starting point. If a different starting point is used, a different LCP finder object is needed.

As mentioned before, Dijkstra's algorithm is used to compute the shortest path. 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.

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.

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.

Value

qt_lcp_finder returns an LCP finder object

qt_find_lcp returns a two column matrix representing the coordinates of the LCP to the destina-
tion point

Examples

```
#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
qt1 = qt_create(rast, range_limit = .9, adj_type="expand") #automatically adds NA cells to bring the dimensions
qt_plot(qt1,crop=TRUE)
start_pt = c(.231,.14)
end_pt = c(74.89,56.11)
#create the LCP finder object
spf = qt_lcp_finder(qt1, start_pt)

#use the LCP finder object to find the LCP to a certain point
path1 = qt_find_lcp(spf, end_pt) #this path will have the cell centroids as the start and end points
path2 = qt_find_lcp(spf, end_pt, use_original_end_points = TRUE) #this path will be identical to path1 except th

#plot the result
qt_plot(qt1, crop=TRUE, border_col="gray60")
points(rbind(start_pt, end_pt), pch=16, col="red")
lines(path1, col="black", lwd=2.5)
lines(path2, col="red", lwd=1)
points(path1, cex=.7, 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") #automatically adds NA cells to bring the dimensions
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 runn:
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
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")
lines(path1)
lines(path2, col="red")
lines(path3, col="blue")
```

---

qt_plot   *Plot a quadtree object*

Description

Plot a quadtree object

Usage

```
qt_plot(
  qt,
  colors = NULL,
  nb = FALSE,
  border_col = "black",
  xlim = NULL,
  ylim = NULL,
  crop = FALSE,
  na_col = "white",
  ...
)
```

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	boolean; whether or not to include lines connecting neighboring cells
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. If TRUE, xlim and ylim are ignored
na_col	character; the color to use for NA cells
...	arguments passed to the default plot function

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") #automatically adds NA cells to bring the dimensions

#-----------
#DEFAULT
#-----------
qt_plot(qt1) #default - no additional parameters provided

#-----------
#CHANGE PLOT EXTENT
#-----------
#note that additional parameters like 'main', 'xlab', 'ylab', etc. will be passed to the default 'plot()' functi
qt_plot(qt1, crop=TRUE, main="cropped") #crop extent to the original extent of the raster
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")

#-----------
#SHOW NEIGHBOR CONNECTIONS
#-----------
qt_plot(qt1, nb=TRUE, border_col="gray60")
```

---

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")
```

---

rcpp_hello_world   *Simple function using Rcpp*

Description

Simple function using Rcpp

Usage

```
rcpp_hello_world()
```

Examples

```
## Not run:
rcpp_hello_world()

## End(Not run)
```

Index