Package: fastshp (via r-universe)

December 31, 2024

Version 0.1-2

Title Fast routines for hanlding large ESRI shapefiles (.shp)

Author Simon Urbanek <simon.urbanek@r-project.org>

Maintainer Simon Urbanek <simon.urbanek@r-project.org>

Description Routines for handling of large ESRI shapefiles (.shp). This includes reading, thinning of points and matching of points to containing shapes. The main aim for this package is to provide the speed to support large shapefiles (millions of points). It is several orders of maginute faster than some other shapefile packages.

License GPL-2

URL http://www.rforge.net/fastshp

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

RemoteUrl https://github.com/s-u/fastshp

RemoteRef HEAD

RemoteSha 3e0eb833f6292013423d93c71f7c26f6dc2d70f1

Contents

as.shp 2
centr 2
inside 3
merge.tiles 4
plot.shp 5
read.shp 6
thin 8

Index **11**

as.shp — *Coerce objects to shapefiles*

Description

`as.shp` coerces lists of polygons to adhere to the `shp` representation.

Usage

```
as.shp(x)
```

Arguments

`x`	object to coerce

Details

Objects of the class `"shp"` are lists of shapes. Each shape comprises of polygons and has an identifier and a bounding box.

The `as.shp` method coerces various representations of shape sets into `"shp"` objects.

Currently the only supported representation to convert from is a list of `"x"`, `"y"` pairs, e.g.: `list(list(x=..., y=...), list(x=..., y=...), ...)` where each element of the list defines a shape. The shapes will be assigned sequential identifiers starting at 1.

Value

Object of the class `"shp"`.

Author(s)

Simon Urbanek

centr — *Compute shape centroids and areas.*

Description

`centr` computes centroids and areas of shapes

Usage

```
centr(shp)
```

Arguments

shp	shapefile object as returned by `read.shp(..., type='list')` or a list of coordinate vector pairs, i.e., `list(shape1=list(x, y), shape2=list(x, y), ...)` where each coordinate vector is a real vector

Value

Data frame with columns:

cx	x coordinate of the centroid
cy	y coordinate of the centroid
area	area of the shape

Note

If a shape has more than one part, only the centroid and area of the first part is computed, subsequent parts are discarded with a warning.

See Also

`read.shp`

---

inside *Finds shapes that contain given points*

Description

`inside` returns indices of shapes that contain given points

Usage

```
inside(shp, x, y, clockwise = TRUE, all = FALSE)
```

Arguments

shp	shape object (as returned by `read.shp(..., format="polygon")`) to match against
x	x coordinates of the points to match
y	y coordinates of the points to match
clockwise	logical: if `TRUE` then polygons are oriented clockwise, otherwise counter-clockwise
all	logical: if `TRUE` then the result is a list of vectors listing all matches, otherwise only the first match per point is returned

Details

The matching uses bounding box as a first approximation and then winding rule (due east) to determine whether the point is inside. If more than one shape matches, the index of the first matching shape is returned unless `all=TRUE` is set in which case each entry is a list of all matches. Points exactly on the boundary (as far as possible by floating point arithmetics) are not considered inside. Note that the shape cooridnates must be in R polygon format (`format="polygon"` in `read.shp`) or have just one part, otherwise parts will not be treated properly.

Value

If `all=FALSE`: Integer vector of the same length as the number of points, each value is either an index of the first matching shape or `NA` if the point is not inside of any shapes.

If `all=TRUE`: List of integer vectors with the indices of matching shapes (which whill be empty if there is no match). There will be as many elements in the list as there are points.

Note

Holes are currently not considered by this method. There is a slower implementation using CGAL that treats holes properly.

Author(s)

Simon Urbanek

`merge.tiles` *Merges adjacent tiles (polygons) where possible*

Description

`merges.tiles` merges adjacent polygons

Usage

```
merge.tiles(x, y, id = rep(1L, length(x)))
```

Arguments

`x`	x coordinates
`y`	y coordinates
`id`	identifiers defining tiles - all contiguous coordinates with the same `id` define one polygon

Details

Each tile is a polygon defined by points with the `id` of the tile. The points for the same `id` must be contiguous. No tiles are allowed to overlap. Shared edges must be defined by points common to both tiles. The orientation must be consistent since only edges with opposite orientation will be merged.

The algorithm used to merge finds all common points in the data, for each common point checks whether edges from that point are shared and any such edges are marked for deletion.

The resulting set of `ids` will be a subset of the initial `ids`. Merged tiles recevied the `id` that comes earlier in the input `id` vector.

Value

List:

`x`	x coordinates
`y`	y coordinates
`id`	identifiers defining the resulting tiles

Author(s)

Simon Urbanek

`plot.shp` *Plot shape files*

Description

`plot.shp` plots a list of shapes

Usage

```
plot.shp(x, xlim, ylim, asp = 1/cos(sum(ylim)/360 * pi), add = FALSE,
         axes = FALSE, full = TRUE, hold = FALSE,
         col = "#e0e0e0", border = "#808080", ...)
```

Arguments

`x`	`"shp"` object as returned from `read.shp`
`xlim`	range to plot in `x` direction; defaults to the range of all bounding boxes
`ylim`	range to plot in `x` direction; defaults to the range of all bounding boxes
`asp`	aspect ratio; the default ensures least distortion in the center of the image
`add`	logical; if `TRUE` polygons are added to the existing plot and `xlim`, `ylim`, `asp`, `axes`, `full` are ignored. Otherwise a new plot is created.
`axes`	logical; passed to `plot`
`.`	

full	logical; if TRUE then margins are removed for full-size plot, otherwise margins are not touched.
hold	logical; if TRUE then the drawing code is wrapped in dev.hold()/dev.flush()
col	colors for the shapes; each shape (possibly consisting of multiple polygons) consumes one element
border	borders for the shapes; consumer and recycled jsut like col
...	additional arguments

Value

NULL invisibly

Note

It is most efficient to plot the result of read.shp(..., format="list"). All other types are converted into that type before plotting. It is most inefficient to use format="table".

Author(s)

Simon Urbanek

See Also

read.shp

Examples

```
# Census 2010 TIGER/Line(TM) state shapefile
fn <- system.file("shp", "tl_2010_us_state10.shp.xz", package="fastshp")
# contrary to the advice above we use the table format, because
# it is a huge file with many points, so we use constrained thinning
# which works on tables
s <- read.shp(xzfile(fn, "rb"), "table")
# substantially reduce the number of points
s <- s[thin.shp(s, 0.01),]
# focus on continental US only
q <- list(x=c(-127.35, -65), y = c(51.5, 22.23))
plot.shp(s, q$x, q$y)
```

---

read.shp *Read ESRI shapefile*

Description

read.shp reads ESRI shapefile format (.shp). Currently only polygons and polylines are supported.

Usage

```
read.shp(where, format = c("list", "pairlist", "polygon", "table"),
         close = TRUE)
```

Arguments

`where`	filename to read the data from or a binary connection or a raw vector with the data content
`format`	output format (see below for details), defaults to "list".
`close`	if `where` is a connection then this flag determines whether the connection will be closed automatically after reading the shapefile (`TRUE`) or not (`FALSE`).

Value

The result depends on the value of the `format` argument:

`"list"` list (generic vector) of shapes exactly as represented in the `.shp` file format: each shape is represented by a list with the following elements:

- idshape identifier
- typeshape type
- boxbounding box - a vector of `xmin, ymin, xmax, ymax`
- parts0-based index of the beginning of each part
- xx coordinates (typically longitude)
- yy coordinates (typically latitude)

`"pairlist"` same as "list" except that the list of shapes is stored in a pairlist and not a list. This is the most memory-efficient way of reading a shapefile, because and all other formats are derived from first reading this format. Pairlists are good for linear scans but inefficient for indexing.

`"polygon"` same as "list" except that coordinates are represented in R polygon format (parts are separated by `NA`s in the coordinates) instead of part indexing. This is typically the preferred format for plotting.

`"table"` The result is a data frame with the following columns: `id`, `type`, `part`, `x`, `y`. Coordinates for each part are therefore identified by common `id`, `part` tuple. This is typically the preferred format for performing computations on the coordinates.

Note

Although other packages exist for reading shapefiles, the focus of this implementation is speed, so it works on very large compilations of shapes, such as the Tiger database which is impossible to load using naive R implementations.

Author(s)

Simon Urbanek

Examples

```
# Census 2010 TIGER/Line(TM) state shapefile
fn <- system.file("shp", "tl_2010_us_state10.shp.xz", package="fastshp")
s <- read.shp(xzfile(fn, "rb"))
```

`thin` *Thin out polyline/polygon by removing unneeded points.*

Description

`thin` thins out a polyline/polygon by removing points that are deemed to have no visual effect under the given tolerance.

Usage

```
thin(x, y, tolerance = 1e-4, lock = NULL, method = 2L, id = NULL)
thin.shp(shp, tolerance = 1e-3, max.width = 5L, all = !is.data.frame(shp))
```

Arguments

`x`	x coordinates of the points
`y`	y coordinates of the points
`tolerance`	maximum allowable distance for a point to be removed
`lock`	defines points that cannot be removed. Can be `NULL` (any point can be removed), a logical vector of the same length as the number of points or a numeric vector specifying the indices of points that will cannot be removed.
`method`	Must be one of `1L` - fast, linear method, but guarantees only `n * tolerance` accuracy where `n` is the number of subsequently removed points. `2L` - slower (`O(n^2)`),more conservative method that guarantees `tolerance` accuracy even with increasing `n`.
`id`	optional index
`shp`	shape object as returned by `read.shp(..., format="table")` or a connection/raw vector/filename which is passed to `read.shp` to obtain such an object
`max.width`	the maximum number of shapes that a single point can belong to. It determines the size of the adjacency table created in the process.
`all`	determines whether the thinning information is included in the shape object and returned as the whole object (`all=TRUE`) or just the thinning logical vector (otherwise)

Details

`thin` performs thinning of one or more polygons defined by coordinates `x` and `y`, where polygons are separated by `NA`.

The default algorithm used here is very simple and fast: it performs a linear scan through all points and for each convex point it measures the distance of the point from a line connecting last unthinned point and the subsequent point. If this distance is below tolerance it is removed. Note that the `x`, `y` space must be Euclidean so coordinates may need to be transformed accordingly (i.e. typically you don't want to use uncorrected lat/lon!). This fast algorithm guarantees only `n * tolerance` accuracy with `n` being the number of subsequently removed points. The extra error will be more noticeable for subsequent slowly drifting points.

The alternative algorithm (`method = 2L`) additionally checks whether any of the previously removed points would be out of tolerance as well - this adds complexity (it is quardatic in the number of removed points), but guarantees that the result is never further than `tolerance` away from the original shape.

The input `x`, `y` can contain multiple segments separated by `NA` (R polygon format). Segments are always assumed to be a loop (you can still use `keep` to force both ends to be non-removable).

`thin.shp` performs a constrained thinning (eventually using `thin`) whereby segments that are shared by two or more polygons are guaranteed to be shared even after thinning. This is done by computing the index from each shared point to all the same points, then comparing running segments that have the same shape `id` list in that index and referenciong only the the first set of points so that the thinnig of those will be used for all subsequent segments of the same point sequence. In addition, all points as which the shape `id` list changes are declared as fixed points that cannot be removed.

All points are compared by their actual coordinate value, no fudge factor is applied, so the source is assumed to be consistent.

Value

`thin`: logical vector of the same length as the number of points with `TRUE` for points that are kept and `FALSE` for removed points.

`this.shp`: same as `thin` if `all = FALSE`, otherwise the `shp` shape obejct is augmented with `thin` element which contains the result of `thin` and the object itself is returned.

Author(s)

Simon Urbanek

Examples

```
# load 2010 Census TIGER/Line(TM) state data (if included)
shp <- system.file("shp","tl_2010_us_state10.shp.xz",package="fastshp")
if (nzchar(shp)) {
  s <- read.shp(xzfile(shp, "rb"), "pol")
  # thin on a cylindrical projection (around ca. 37 deg lat)
  t <- lapply(s, function(o) thin(o$x / 1.25, o$y, 1e-3, method = 1L))
  par(mar = rep(0, 4))
  plot(c(-125,-67), c(25, 49.4), asp=1.25, ty='n', axes=FALSE)
```

```
  for (i in seq.int(s))
     polygon(s[[i]]$x[t[[i]]], s[[i]]$y[t[[i]]], col="#eeeeee")
 cat(" reduction: ", 100 - sum(sapply(t, sum)) / sum(sapply(t, length)) * 100, "%\n", sep='')
  # use the more conservative algorithm
  t <- lapply(s, function(o) thin(o$x / 1.25, o$y, 1e-3, method = 2L))
 cat(" reduction: ", 100 - sum(sapply(t, sum)) / sum(sapply(t, length)) * 100, "%\n", sep='')
  # use constrained thinning:
  st <- read.shp(xzfile(shp, "rb"), "table")
  st$x <- st$x / 1.25
  a <- thin.shp(st, 1e-3)
  cat(" reduction: ", 100 - sum(a) / length(a) * 100, "%\n", sep='')
  par(mfrow=c(1, 2))
  # compare unconstrained and constrained thinning up close (NY/NJ area)
  plot(0, 0, xlim=c(-74.22, -74.15), ylim=c(40.55,40.67), asp=1.25, axes=FALSE)
  for (i in seq.int(s))
  polygon(s[[i]]$x[t[[i]]], s[[i]]$y[t[[i]]], col=c("#0000ff80","#80800080")[i %% 2 + 1L], border=1)
  plot(0, 0, xlim=c(-74.22, -74.15), ylim=c(40.55,40.67), asp=1.25, axes=FALSE)
  for (i in unique(st$id))
  polygon(st$x[st$id==i]*1.25, st$y[st$id==i], col=c("#0000ff80","#80800080")[i %% 2 + 1L], border=1)
}
```

Index

∗ **IO**
    read.shp, 6
∗ **manip**
    as.shp, 2
    centr, 2
    inside, 3
    merge.tiles, 4
    thin, 8
∗ **plot**
    plot.shp, 5

as.shp, 2

centr, 2

inside, 3

merge.tiles, 4

plot.shp, 5

read.shp, *3*, *5*, *6*, 6, *8*

thin, 8