

# Package ‘spaths’

April 9, 2024

**Type** Package

**Title** Shortest Paths Between Points in Grids

**Version** 1.1.0

**Description** Shortest paths between points in grids. Optional barriers and custom transition functions. Applications regarding planet Earth, as well as generally spheres and planes. Optimized for computational performance, customizability, and user friendliness. Graph-theoretical implementation tailored to gridded data. Currently focused on Dijkstra's (1959) <doi:10.1007/BF01386390> algorithm. Future updates broaden the scope to other least cost path algorithms and to centrality measures.

**License** MIT + file LICENSE

**Encoding** UTF-8

**URL** <https://github.com/ChristianDueben/spaths>

**BugReports** <https://github.com/ChristianDueben/spaths/issues>

**Imports** base (>= 4.0.0), Rcpp (>= 1.0.9), data.table, parallel, stats, utils

**LinkingTo** Rcpp

**Suggests** terra, knitr, rmarkdown, testthat (>= 3.0.0)

**VignetteBuilder** knitr

**RoxygenNote** 7.3.1

**Config/testthat/edition** 3

**NeedsCompilation** yes

**Author** Christian Dübén [aut, cre]

**Maintainer** Christian Dübén <christian.dueben@uni-hamburg.de>

**Repository** CRAN

**Date/Publication** 2024-04-09 07:50:02 UTC

## R topics documented:

<code>rnd_locations</code> . . . . .	2
<code>shortest_paths</code> . . . . .	3

---

rnd_locations	<i>Random location drawing</i>
---------------	--------------------------------

---

### Description

This function draws random unprojected (lonlat) locations.

### Usage

```
rnd_locations(  
  nobs,  
  xmin = -180,  
  xmax = 180,  
  ymin = -90,  
  ymax = 90,  
  output_type = c("data.table", "data.frame", "SpatVector")  
)
```

### Arguments

nobs	Number of observations
xmin	Minimum longitude
xmax	Maximum longitude
ymin	Minimum latitude
ymax	Maximum latitude
output_type	type of output object. Either "data.table" (default), "data.frame", or "SpatVector".

### Details

By default, the function draws a global sample of random locations. You can restrict it to a certain region via specifying `xmin`, `xmax`, `ymin`, and `ymax`. The function draws from a uniform spatial distribution that assumes the planet to be a perfect sphere. The spherical assumption is common in GIS functions, but deviates from Earth's exact shape.

### Value

Returns a `data.table`, `data.frame`, or `SpatVector` object of unprojected (lonlat) points.

### Examples

```
rnd_locations(1000)
```

---

`shortest_paths`*Shortest Paths and/ or Distances between Locations*

---

**Description**

The shortest paths and/ or distance between locations in a grid according to Dijkstra's (1959) algorithm.

**Usage**

```
shortest_paths(  
  rst,  
  origins,  
  destinations = NULL,  
  output = c("distances", "lines", "both"),  
  output_class = NULL,  
  origin_names = NULL,  
  destination_names = NULL,  
  pairwise = FALSE,  
  contiguity = c("queen", "rook"),  
  spherical = TRUE,  
  radius = 6371010,  
  extent = NULL,  
  dist_comp = c("spathes", "terra"),  
  tr_fun = NULL,  
  v_matrix = FALSE,  
  tr_directed = TRUE,  
  pre = TRUE,  
  early_stopping = TRUE,  
  bidirectional = FALSE,  
  update_rst = NULL,  
  touches = TRUE,  
  ncores = NULL,  
  par_lvl = c("update_rst", "points"),  
  show_progress = FALSE,  
  bar_limit = 150L,  
  path_type = c("int", "unsigned short int"),  
  distance_type = c("double", "float", "int", "unsigned short int")  
)
```

**Arguments**

`rst` SpatRaster (terra), RasterLayer (raster), matrix, or list of matrices object. RasterLayers are converted to SpatRasters. Pixels with non-NA values in all layers mark the cells through which the algorithm may pass. The values of `rst` can be accessed in `tr_fun`.

origins	Origin points at which the shortest paths start. If <code>rst</code> is a <code>SpatRaster</code> or <code>RasterLayer</code> object, these points can be passed as a single <code>SpatVector</code> ( <code>terra</code> ), <code>sf</code> ( <code>sf</code> ), or <code>Spatial*</code> ( <code>sp</code> ) object. <code>sf</code> and <code>sp</code> objects are converted to <code>SpatVectors</code> . Polygons and lines are converted to points using their centroid. If <code>rst</code> is a matrix or list of matrices, <code>origins</code> must be a single matrix, <code>data.frame</code> , or <code>data.table</code> of coordinates with columns named "x" and "y". The coordinates must refer to points in the reference system that <code>rst</code> utilizes. Lines and polygons are thus not accepted in this case. Details on which points the function connects are outlined below.
destinations	Destination points to which the shortest paths are derived. It defaults to <code>NULL</code> , resulting in the function to compute shortest paths between the <code>origins</code> points. Otherwise, the same input rules as for <code>origins</code> apply. Details on which points the function connects are outlined below.
output	"distances" (default), "lines", or "both". "distances" lists the total transition costs along the shortest paths. By default, it is the distance between origin and destination in meters, if <code>rst</code> is an unprojected <code>SpatRaster</code> or <code>RasterLayer</code> or if <code>dist_comp = "terra"</code> . Otherwise, it is denoted in the projection's units. If you pass another function to <code>tr_fun</code> , the total transition cost is measured in the units of <code>tr_fun</code> 's results. "lines" returns the shortest paths as spatial lines. "both" returns both distances and lines. "distances" is faster and requires less RAM than "lines" or "both".
output_class	Class of the returned object. With <code>output = "distances"</code> , the options are "data.table" (default) and "data.frame". With <code>output = "lines"</code> or <code>output = "both"</code> , the options are "SpatVector" (default when <code>rst</code> is a <code>SpatRaster</code> or <code>RasterLayer</code> ) and "list" (default when <code>rst</code> is a matrix or a list of matrices). In the case of "list", the attributes, the line coordinates, and the CRS are returned as individual list elements. The first element in the list of line coordinates refers to the first row in the attributes table etc. "SpatVector" is only available with a <code>SpatRaster</code> or <code>RasterLayer</code> <code>rst</code> . <code>output_class</code> changes the format of the returned object, not the information it contains.
origin_names	Character specifying the name of the column in the <code>origins</code> object used to label the origins in the output object. It defaults to row numbers.
destination_names	Character specifying the name of the column in the <code>destinations</code> object used to label the destinations in the output object. It defaults to row numbers.
pairwise	Logical specifying whether to compute pairwise paths, if <code>origins</code> and <code>destinations</code> have equally many rows. If <code>TRUE</code> , the function computes the shortest path between the first origin and the first destination, the second origin and the second destination, etc. Otherwise, it derives the shortest paths from all origins to all destinations. <code>pairwise = TRUE</code> can alter the order in which results are returned. Check the output's <code>origins</code> and <code>destinations</code> columns for the respective order.
contiguity	"queen" (default) for queen's case contiguity or "rook" for rook's case contiguity. In the latter case, the algorithm only moves between horizontally or vertically adjacent cells. In the former case, it is also travels between diagonally adjacent cells. "rook" is more efficient than "queen" as it implies fewer edges.
spherical	Logical specifying whether coordinates are unprojected, i.e. lonlat, and refer to a sphere, e.g. a planet, if <code>rst</code> is a matrix or a list of matrices. It defaults to <code>TRUE</code> .

	<p>If FALSE, the function assumes the coordinates to originate from a planar projection. This argument has no effect when <code>rst</code> is a <code>SpatRaster</code> or <code>RasterLayer</code>.</p>
<code>radius</code>	<p>Radius of the object, e.g. planet, if <code>spherical = TRUE</code>. This argument has no effect when <code>rst</code> is a <code>SpatRaster</code> or <code>RasterLayer</code>.</p>
<code>extent</code>	<p>Vector of length 4, specifying the extent of <code>rst</code>, if <code>rst</code> is a matrix or a list of matrices. It must contain <code>xmin</code>, <code>xmax</code>, <code>ymin</code>, and <code>ymax</code>, in that order. The argument has no effect when <code>rst</code> is a <code>SpatRaster</code> or <code>RasterLayer</code>.</p>
<code>dist_comp</code>	<p>Method to compute distances between adjacent cells. "spaths" (default) or "terra". The default "spaths" uses spherical (Haversine) distances in case of lonlat data and planar (Euclidean) distances in case of projected (non-lonlat) data. The functions are optimized based on the fact that many inter-pixel distances are identical. Modelling the planet as a perfect sphere is in line with e.g. the <code>s2</code> package, but is of course an oversimplification. With "terra", the function derives distances via <code>terra::distance</code>. Because this computes all inter-pixel distances separately, it is slower than the "spaths" approach. It does take the non-spherical nature of the planet into account though. With <code>tr_fun</code>, you can specify a custom distance function that uses neither "spaths" nor "terra" distances.</p>
<code>tr_fun</code>	<p>The transition function based on which to compute edge weights, i.e. the travel cost between adjacent grid cells. Defaults to the geographic distance between pixel centroids. Permitted function parameter names are <code>d</code> (distance between the pixel centroids), <code>x1</code> (x coordinate or longitude of the first cell), <code>x2</code> (x coordinate or longitude of the second cell), <code>y1</code> (y coordinate or latitude of the first cell), <code>y2</code> (y coordinate or latitude of the second cell), <code>v1</code> (<code>rst</code> layers' values from the first cell), <code>v2</code> (<code>rst</code> layers' values from the second cell), and <code>nc</code> (number of CPU cores according to the <code>ncores</code> argument). If the data is unprojected, i.e. lonlat, or if <code>dist_comp = "terra"</code>, <code>d</code> is measured in meters. Otherwise, it uses the units of the CRS. If <code>rst</code> has one layer, the values are passed to <code>v1</code> and <code>v2</code> as vectors, otherwise they are passed as a data table where the first column refers to the first layer, the second column to the second layer etc. Note that data tables are data frames.</p>
<code>v_matrix</code>	<p>Logical specifying whether to pass values to <code>v1</code> and <code>v2</code> in <code>tr_fun</code> as matrices (TRUE) instead of data tables in the multi-layer case and vectors in the single-layer case (FALSE). It defaults to FALSE. Setting it to TRUE might e.g. be useful when defining <code>tr_fun</code> as a C++ Armadillo function.</p>
<code>tr_directed</code>	<p>Logical specifying whether <code>tr_fun</code> creates a directed graph. In a directed graph, transition costs can be asymmetric. Traveling from cells A to B may imply a different cost than traveling from B to A. It defaults to TRUE and only has an effect when <code>tr_fun</code> is not NULL. The default without <code>tr_fun</code> constructs an undirected graph.</p>
<code>pre</code>	<p>Logical specifying whether to compute the distances between neighboring cells before executing the shortest paths algorithm in C++. <code>pre</code> only has an effect, when no <code>tr_fun</code> is specified and <code>dist_comp = "spaths"</code>, as the distances are otherwise imported from R. TRUE (default) is in the vast majority of cases faster than FALSE. FALSE computes distances between neighboring cells while the shortest paths algorithm traverses the graph. This requires less RAM, but is slower than TRUE, unless <code>early_stopping = TRUE</code> and all points are close to</p>

each other. TRUE's speed advantage is even larger, when `update_rst` is not NULL.

<code>early_stopping</code>	Logical specifying whether to stop the shortest path algorithm once the target cells are reached. It defaults to TRUE, which can be faster than FALSE, if the points are close to each other compared to the full set of <code>rst</code> cells. If at least one points pair is far from each other, FALSE is the faster setting. FALSE computes the distance to all cells and then extracts the distance to the target cells. It, therefore, does not check for each visited cell, whether it is in the set of targets. TRUE and FALSE produce the same result and only differ in terms of computational performance.
<code>bidirectional</code>	Logical specifying whether to produce paths or distances in both directions, if destinations are not specified and no directed transition function is given. In that case, the distance and the path from point A to point B is the same as the distance and path from point B to point A. FALSE (default) only returns distances or paths in one direction. Declaring TRUE returns distances or paths in both directions. This parameter's objective is to control the return object's RAM requirement. It only has an effect, if destinations are not specified and no directed transition function is given.
<code>update_rst</code>	Object updating <code>rst</code> with moving barriers. It defaults to NULL, corresponding to <code>rst</code> not being updated. If <code>rst</code> is a <code>SpatRaster</code> or <code>RasterLayer</code> , <code>update_rst</code> can be a <code>SpatVector</code> ( <code>terra</code> ), <code>sf</code> ( <code>sf</code> ), or <code>Spatial*</code> ( <code>sp</code> ) object, or a list of them. <code>sf</code> and <code>sp</code> objects are converted to <code>SpatVectors</code> . The function updates <code>rst</code> by setting any cell intersecting with <code>update_rst</code> to NA, thereby not allowing the shortest paths algorithm to pass through that cell. <code>update_rst</code> only sets non-NA cells to NA, not vice versa. The elements of <code>update_rst</code> always update the unmodified <code>rst</code> . I.e. if <code>update_rst</code> is a list of two polygons, the shortest paths are derived three times: once based on the not updated <code>rst</code> , denoted layer 0 in the output, once based on <code>rst</code> updated with the first polygon, referred to as layer 1, and once based on <code>rst</code> updated with the second polygon, termed layer 2. The second polygon updates the unmodified <code>rst</code> , not the <code>rst</code> updated by the first polygon. If <code>rst</code> is a matrix or a list of matrices, <code>update_rst</code> can be a vector of cell numbers, a matrix, or a list of either. Analogously to the <code>SpatRaster</code> case, these objects mark which cells to set to NA. As in <code>terra</code> , cell numbers start with 1 in the top left corner and then increase first from left to right and then from top to bottom. The cell numbers in the vector and the NA cells in the matrix identify the pixels to set to NA. Accordingly, the matrix is of equal dimensions as <code>rst</code> .
<code>touches</code>	Logical specifying the <code>touches</code> argument of <code>terra::extract</code> used when <code>update_rst</code> is a <code>SpatVector</code> , <code>sf</code> , or <code>Spatial*</code> object. It defaults to TRUE. If FALSE, the function only removes cells on the line render path or with the center point inside a polygon.
<code>ncores</code>	An integer specifying the number of CPU cores to use. It defaults to the number of cores installed on the machine. A value of 1 induces a single-threaded process.
<code>par_lvl</code>	"points" or "update_rst", indicating the level at which to parallelize when using multiple cores and <code>update_rst</code> is a list. "points" parallelizes over the origin (and destination) point combinations in both the base grid not updated by <code>update_rst</code> and the grids updated with <code>update_rst</code> . The default "update_rst"

	is equivalent to "points" in the base grid, but parallelizes at the <code>update_rst</code> list level in the updated grid stage.
<code>show_progress</code>	Logical specifying whether the function prints messages on progress. It defaults to <code>FALSE</code> .
<code>bar_limit</code>	Integer specifying until up to how many paths or list elements of <code>update_rst</code> to display a progress bar, if <code>show_progress = TRUE</code> . It defaults to 150, in which case the function prints one = per computed path, if there are no more than 150 paths requested. In the grids updated with <code>update_rst</code> , the function displays one = per processed <code>update_rst</code> list element, not per path. In parallel applications, the progress bar can notably slow the execution as the functions only permit one thread to write to output at a time. Do not set the argument too high to avoid R crashes from text buffer overflows.
<code>path_type</code>	Data type with which C++ stores cell numbers. "int" (default) is the 4 byte signed integers that R also uses and is the fastest option. "unsigned short int" is a 2 byte unsigned integer which requires less RAM than "int", but only works if there are less than 65,535 non-NA cells and is comparatively slower because it requires type conversions.
<code>distance_type</code>	Data type with which C++ stores distances. "double" (default) is a double precision 8 byte floating point number. It is the fastest and most precise option and also used by R as its numeric data type. "float" is a single precision 4 byte floating point number, which stores decimal values less precisely than "double" does and is comparatively slower because it requires type conversions. "int" and "unsigned short int" are the integer types described in the <code>path_type</code> documentation above. With "int" and "unsigned short int", distances are rounded to integers. When employing these integers types, the distance between any cells in <code>rst</code> , not just the cells of interest, must not exceed 2,147,483,647 and 65,535 respectively. The distance difference caused by rounding double values to another type can accumulate along the shortest paths and can result in notable distance deviation in the output. The recommendation is to stick with the default "double" unless the machine does not have enough RAM to run the function otherwise.

## Details

This function computes shortest paths and/ or distance between locations in a grid using Dijkstra's algorithm. Examples are a ship navigating around land masses or a hiker traversing mountains.

Let us explore the ship example to illustrate how `shortest_paths` works. To compute shortest paths between ports around the world, you start with a global `SpatRaster`, in which all land pixels are set to NA and all ocean pixels are set a non-NA value, e.g. 1. A `SpatVector` marks port locations as points on water pixels. Passing these two objects to the parameters `rst` and `origins` respectively derives the shortest paths from each port to all other ports conditional on ships solely traversing water pixels and returns the distances, i.e. lengths of these paths. If you are not interested in the distances, but in the spatial lines themselves, set `output` to "lines". If you want to obtain both, set it to "both".

In a different application, you do not want to compute the paths between all ports, but only the paths between the ports on the northern hemisphere and the ports on the southern hemisphere, but not the paths between ports within the same hemisphere. To assess this, you split the ports into two

SpatVectors and pass them to origins and destinations respectively. What if you do not want to connect all origins to all destinations? Set `pairwise` to `TRUE` to connect the first origin just to the first destination, the second origin to the second destination, etc.

By default, the distance or transition cost between adjacent cells of the input grid is the geographic distance between the cells' centroids. What if the boat is a sailing vessel that minimizes travel time conditional on wind speed, wind direction, and ocean currents? Construct a `SpatRaster` with three layers containing information on the three variables respectively. Define a transition function that combines the three layers into a travel time measure and pass the `SpatRaster` to `rst` and this function to `tr_fun`. `tr_fun` makes this package very versatile. With custom transition functions, you can take this software out of the geo-spatial context and e.g. apply it to biomedical research.

Going back to the ship routing example, consider that there are hurricanes in the Caribbean. Ships traveling from India to Australia do not care, but ships traveling from Mexico to the Netherlands have to go around the storm and must not take the shortest path through the hurricane. You have ten `SpatVector` polygons delineating the extent of the hurricane on ten different days. You want to know what the shortest paths are given that ships must go around the polygon on that day. Calling `shortest_paths` ten times with ten different `SpatRasters` would be very inefficient. This would assemble the graph ten times and recompute also paths unaffected by the hurricanes, such as the path between India and Australia, in each iteration. Instead, pass the `SpatVector` polygons to `update_rst`. `shortest_paths` then produces the shortest paths for a hurricane-free route and all ten hurricane days, only reestimating the paths that are affected by the hurricane polygon on a specific day.

Applications to Earth should always pass a `SpatRaster` or `RasterLayer` to `rst`. The option to use a matrix or a list of matrices is meant for applications to other planets, non-geo-spatial settings, and users who cannot install the terra package on their system.

The largest source of runtime inefficiency is the quantity of non-NA pixels in the `rst` grid. Limit the `rst` argument to the relevant area. E.g. crop the grid to the North Atlantic when computing shipping routes between Canada and France. And set regions through which the shortest path does certainly not pass to NA.

`shortest_paths` is optimized for computational performance. Most of its functions are written in C++ and it does not use a general purpose graph library, but comes with its custom graph-theoretical implementation tailored to gridded inputs.

The [vignette](#) provides further details on the package.

## Value

If `'output = "distances"'`, the output is by default returned as a data table. If you want the result to be a data frame only, not a data table, set `'output_class'` to `"data.frame"`. If `'output'` is `"lines"` or `"both"`, the function returns a `SpatVector`, if `'rst'` is a `SpatRaster` or a `RasterLayer`, and a list, if `'rst'` is a matrix or a list of matrices. Explicitly setting `'output_class'` to `"list"` returns a list in any case. `'output_class = "SpatVector"'`, however, returns a `SpatVector` only if `'rst'` is a `SpatRaster` or a `RasterLayer`.

If `output = "distances"` or `output = "both"`, the `distances` variable marks which points are connected. Unconnected point pairs have an `Inf` distance, if `distance_type = "double"` or `distance_type = "float"`, and an `NA` distance, if `distance_type = "int"` or `distance_type = "unsigned short int"`. If `output = "lines"`, the `connected` variable marks which points are connected. Points are connected, when it is possible to travel between them via non-NA cells in `rst`.



## References

Dijkstra, E. W. 1959. "A note on two problems in connexion with graphs." *Numerische Mathematik* 1 (1): 269–71.

## Examples

```
# Generate example data
set.seed(2L)
input_grid <- terra::rast(crs = "epsg:4326", resolution = 2, vals = sample(c(1L, NA_integer_),
  16200L, TRUE, c(0.8, 0.2)))
origin_pts <- rnd_locations(5L, output_type = "SpatVector")
origin_pts$name <- sample(letters, 5)
destination_pts <- rnd_locations(5L, output_type = "SpatVector")

# Compute distances
shortest_paths(input_grid, origin_pts)
shortest_paths(input_grid, origin_pts, bidirectional = TRUE)
shortest_paths(input_grid, origin_pts, destination_pts)
shortest_paths(input_grid, origin_pts, origin_names = "name")
shortest_paths(input_grid, origin_pts, destination_pts, pairwise = TRUE)

# Compute lines
shortest_paths(input_grid, origin_pts, output = "lines")

# Compute distances and lines
shortest_paths(input_grid, origin_pts, output = "both")

# Use custom transition function
input_grid[input_grid == 1L] <- stats::runif(terra::freq(input_grid, value = 1L)$count,
  max = 1000)
shortest_paths(input_grid, origin_pts, tr_fun = function(d, v1, v2) sqrt(d^2 + abs(v2 - v1)^2),
  tr_directed = FALSE)

# Compute distances with grid updating
barrier <- terra::vect("POLYGON ((-179 1, 30 1, 30 0, -179 0, -179 1))", crs = "epsg:4326")
shortest_paths(input_grid, origin_pts, update_rst = barrier)
barriers <- list(barrier, terra::vect("POLYGON ((0 0, 0 89, 1 89, 1 0, 0 0))",
  crs = "epsg:4326"))
shortest_paths(input_grid, origin_pts, update_rst = barriers)
```

# Index

`rnd_locations`, 2

`shortest_paths`, 3