Package ‘dodgr’

May 6, 2020

Title Distances on Directed Graphs
Version 0.2.7
Description Distances on dual-weighted directed graphs using priority-queue shortest paths (Padgham (2019) <doi:10.32866/6945>). Weighted directed graphs have weights from A to B which may differ from those from B to A. Dual-weighted directed graphs have two sets of such weights. A canonical example is a street network to be used for routing in which routes are calculated by weighting distances according to the type of way and mode of transport, yet lengths of routes must be calculated from direct distances.

Depends R (>= 3.5.0)
License GPL-3
Imports callr, digest, magrittr, methods, osmdata, Rcpp (>= 0.12.6), RcppParallel, RcppThread
Suggests dplyr, geodist, ggplot2, igraph, igraphdata, jsonlite, knitr, purrr, rbenchmark, RColorBrewer, rmarkdown, roxygen2, scales, sf, testthat, tidygraph
LinkingTo Rcpp, RcppParallel, RcppThread
SystemRequirements C++11, GNU make
VignetteBuilder knitr
NeedsCompilation yes
Encoding UTF-8
LazyData true
URL https://github.com/ATFutures/dodgr
BugReports https://github.com/ATFutures/dodgr/issues
RoxygenNote 7.1.0
Author Mark Padgham [aut, cre], Andreas Petutschnig [aut], Robin Lovelace [ctb], Andrew Smith [ctb], Malcolm Morgan [ctb], Shane Saunders [cph] (Original author of included code for priority heaps)
R topics documented:

<table>
<thead>
<tr>
<th>R topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear_dodgr_cache</td>
<td>3</td>
</tr>
<tr>
<td>compare_heaps</td>
<td>3</td>
</tr>
<tr>
<td>dodgr</td>
<td>4</td>
</tr>
<tr>
<td>dodgr_cache_off</td>
<td>5</td>
</tr>
<tr>
<td>dodgr_cache_on</td>
<td>5</td>
</tr>
<tr>
<td>dodgr_centrality</td>
<td>6</td>
</tr>
<tr>
<td>dodgr_components</td>
<td>8</td>
</tr>
<tr>
<td>dodgr_contract_graph</td>
<td>8</td>
</tr>
<tr>
<td>dodgr_distances</td>
<td>9</td>
</tr>
<tr>
<td>dodgr_diffs</td>
<td>10</td>
</tr>
<tr>
<td>dodgr_flowmap</td>
<td>13</td>
</tr>
<tr>
<td>dodgr_flows_aggregate</td>
<td>14</td>
</tr>
<tr>
<td>dodgr_flows_disperse</td>
<td>16</td>
</tr>
<tr>
<td>dodgr_flows_si</td>
<td>18</td>
</tr>
<tr>
<td>dodgr_full_cycles</td>
<td>20</td>
</tr>
<tr>
<td>dodgr_fundamental_cycles</td>
<td>21</td>
</tr>
<tr>
<td>dodgr_insert_vertex</td>
<td>22</td>
</tr>
<tr>
<td>dodgr_isochrones</td>
<td>23</td>
</tr>
<tr>
<td>dodgr_isodists</td>
<td>24</td>
</tr>
<tr>
<td>dodgr_isoverts</td>
<td>25</td>
</tr>
<tr>
<td>dodgr_paths</td>
<td>26</td>
</tr>
<tr>
<td>dodgr_sample</td>
<td>27</td>
</tr>
<tr>
<td>dodgr_sflines_to_poly</td>
<td>28</td>
</tr>
<tr>
<td>dodgr_streetnet</td>
<td>29</td>
</tr>
<tr>
<td>dodgr_streetnet_sc</td>
<td>30</td>
</tr>
<tr>
<td>dodgr_times</td>
<td>31</td>
</tr>
<tr>
<td>dodgr_to_igraph</td>
<td>32</td>
</tr>
<tr>
<td>dodgr_to_sf</td>
<td>33</td>
</tr>
<tr>
<td>dodgr_to_sfc</td>
<td>34</td>
</tr>
<tr>
<td>dodgr_to_tidygraph</td>
<td>35</td>
</tr>
<tr>
<td>dodgr_uncontract_graph</td>
<td>35</td>
</tr>
<tr>
<td>dodgr_vertices</td>
<td>36</td>
</tr>
<tr>
<td>estimate_centrality_threshold</td>
<td>37</td>
</tr>
<tr>
<td>estimate_centrality_time</td>
<td>37</td>
</tr>
<tr>
<td>hampi</td>
<td>38</td>
</tr>
<tr>
<td>igraph_to_dodgr</td>
<td>39</td>
</tr>
<tr>
<td>match_points_to_graph</td>
<td>40</td>
</tr>
<tr>
<td>match_pts_to_graph</td>
<td>41</td>
</tr>
<tr>
<td>merge_directed_graph</td>
<td>42</td>
</tr>
<tr>
<td>os_roads_bristol</td>
<td>43</td>
</tr>
<tr>
<td>weighting_profiles</td>
<td>44</td>
</tr>
</tbody>
</table>
clear_dodgr_cache

Description

Remove cached versions of dodgr graphs. This function should generally not be needed, except if graph structure has been directly modified other than through dodgr functions; for example by modifying edge weights or distances. Graphs are cached based on the vector of edge IDs, so manual changes to any other attributes will not necessarily be translated into changes in dodgr output unless the cached versions are cleared using this function. See https://github.com/ATFutures/dodgr/wiki/Caching-of-streetnets-and-contracted-graphs for details of caching process.

Usage

clear_dodgr_cache()

Value

Nothing; the function silently clears any cached objects

compare_heaps

Description

Perform timing comparison between different kinds of heaps as well as with equivalent igraph routine distances. To do this, a random sub-graph containing a defined number of vertices is first selected. Alternatively, this random sub-graph can be pre-generated with the dodgr_sample function and passed directly.

Usage

compare_heaps(graph, nverts = 100, replications = 2)

Arguments

graph data.frame object representing the network graph (or a sub-sample selected with codedodgr_sample)
nverts Number of vertices used to generate random sub-graph. If a non-numeric value is given, the whole graph will be used.
replications Number of replications to be used in comparison
Value

Result of `rbenchmark::benchmark` comparison in `data.frame` form.

Note

`igraph` caches intermediate results of graph processing, so the `igraph` comparisons will be faster on subsequent runs. To obtain fair comparisons, run only once or re-start the current R session.

Examples

```r
graph <- weight_streetnet(hampi)
compare_heaps(graph, nverts = 1000, replications = 1)
```

Description

Distances on dual-weighted directed graphs using priority-queue shortest paths. Weighted directed graphs have weights from A to B which may differ from those from B to A. Dual-weighted directed graphs have two sets of such weights. A canonical example is a street network to be used for routing in which routes are calculated by weighting distances according to the type of way and mode of transport, yet lengths of routes must be calculated from direct distances.

The Main Function

- `dodgr_dists()`: Calculate pair-wise distances between specified pairs of points in a graph.

Functions to Obtain Graphs

- `dodgr_streetnet()`: Extract a street network in Simple Features (`sf`) form.
- `weight_streetnet()`: Convert an `sf`-formatted street network to a `dodgr` graph through applying specified weights to all edges.

Functions to Modify Graphs

- `dodgr_components()`: Number all graph edges according to their presence in distinct connected components.
- `dodgr_contract_graph()`: Contract a graph by removing redundant edges.

Miscellaneous Functions

- `dodgr_sample()`: Randomly sample a graph, returning a single connected component of a defined number of vertices.
- `dodgr_vertices()`: Extract all vertices of a graph.
- `compare_heaps()`: Compare the performance of different priority queue heap structures for a given type of graph.
**dodgr_cache_off**

---

dodgr_cache_off  dodgr_cache_off

---

**Description**

Turn off all dodgr caching in current session. This is useful if speed is paramount, and if graph contraction is not needed. Caching can be switched back on with `dodgr_cache_on`.

**Usage**

`dodgr_cache_off()`

**Value**

Nothing; the function invisibly returns `TRUE` if successful.

---

dodgr_cache_on  dodgr_cache_on

---

**Description**

Turn on all dodgr caching in current session. This will only have an effect after caching has been turned off with `dodgr_cache_off`.

**Usage**

`dodgr_cache_on()`

**Value**

Nothing; the function invisibly returns `TRUE` if successful.
dodgr_centrality

dodgr_centrality
dodgr_centrality

dodgr_centrality
dodgr_centrality

Description

Calculate betweenness centrality for a 'dodgr' network, in either vertex- or edge-based form.

Usage

dodgr_centrality(
  graph,
  contract = TRUE,
  edges = TRUE,
  dist_threshold = NULL,
  heap = "BHeap"
)

Arguments

graph 'data.frame' or equivalent object representing the network graph (see Details)
contract If 'TRUE', centrality is calculated on contracted graph before mapping back on to the original full graph. Note that for street networks, in particular those obtained from the osmdata package, vertex placement is effectively arbitrary except at junctions; centrality for such graphs should only be calculated between the latter points, and thus 'contract' should always be 'TRUE'.
edges If 'TRUE', centrality is calculated for graph edges, returning the input 'graph' with an additional 'centrality' column; otherwise centrality is calculated for vertices, returning the equivalent of 'dodgr_vertices(graph)', with an additional vertex-based 'centrality' column.
dist_threshold If not 'NULL', only calculate centrality for each point out to specified threshold. Setting values for this will result in approximate estimates for centrality, yet with considerable gains in computational efficiency. For sufficiently large values, approximations will be accurate to within some constant multiplier. Appropriate values can be established via the estimate_centrality_threshold function.
heap Type of heap to use in priority queue. Options include Fibonacci Heap (default; 'FHeap'), Binary Heap ('BHeap'), Trinomial Heap ('TriHeap'), Extended Trinomial Heap ('TriHeapExt', and 2-3 Heap ('Heap23')

Value

Modified version of graph with additional 'centrality' column added.

Note

Centrality is calculated by default using parallel computation with the maximal number of available cores or threads. This number can be reduced by specifying a value via RcppParallel::setThreadOptions (numThreads = <desired_number>).
Examples

```r
dodgr_centrality

Examples

dodgr_centrality

```
dodgr_components

Description
Identify connected components of graph and add corresponding component column to data.frame.

Usage
dodgr_components(graph)

Arguments
graph A data.frame of edges

Value
Equivalent graph with additional component column, sequentially numbered from 1 = largest component.

Examples
graph <- weight_streetnet(hampi)
graph <- dodgr_components(graph)

dodgr_contract_graph

Description
Removes redundant (straight-line) vertices from graph, leaving only junction vertices.

Usage
dodgr_contract_graph(graph, verts = NULL)

Arguments
graph A flat table of graph edges. Must contain columns labelled from and to, or start and stop. May also contain similarly labelled columns of spatial coordinates (for example from_x or stop_lon).
verts Optional list of vertices to be retained as routing points. These must match the from and to columns of graph.
Value
A contracted version of the original graph, containing the same number of columns, but with each row representing an edge between two junction vertices (or between the submitted verts, which may or may not be junctions).

Examples
```
graph <- weight_streetnet (hampi)
nrow (graph) # 5,973
graph <- dodgr_contract_graph (graph)
nrow (graph) # 662
```

Description
Alias for dodgr_dists

Usage
```
dodgr_distances(
  graph,
  from = NULL,
  to = NULL,
  shortest = TRUE,
  pairwise = FALSE,
  heap = "BHeap",
  parallel = TRUE,
  quiet = TRUE
)
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>graph</td>
<td>data.frame or equivalent object representing the network graph (see Notes)</td>
</tr>
<tr>
<td>from</td>
<td>Vector or matrix of points from which route distances are to be calculated (see Notes)</td>
</tr>
<tr>
<td>to</td>
<td>Vector or matrix of points to which route distances are to be calculated (see Notes)</td>
</tr>
<tr>
<td>shortest</td>
<td>If FALSE, calculate distances along the fastest rather than shortest routes (see Notes).</td>
</tr>
<tr>
<td>pairwise</td>
<td>If TRUE, calculate distances only between the ordered pairs of from and to.</td>
</tr>
<tr>
<td>heap</td>
<td>Type of heap to use in priority queue. Options include Fibonacci Heap (default; FHeap), Binary Heap (BHeap), Trinomial Heap (TriHeap), Extended Trinomial Heap (TriHeapExt, and 2-3 Heap (Heap23)).</td>
</tr>
<tr>
<td>parallel</td>
<td>If TRUE, perform routing calculation in parallel (see details)</td>
</tr>
<tr>
<td>quiet</td>
<td>If FALSE, display progress messages on screen.</td>
</tr>
</tbody>
</table>
Value

square matrix of distances between nodes

Examples

# A simple graph
graph <- data.frame (from = c ("A", "B", "B", "C", "C", "D", "D"),
                      d = c (1, 2, 1, 3, 2, 1, 2, 1))
dodgr_dists (graph)

# A larger example from the included [hampi()] data.
graph <- weight_streetnet (hampi)
from <- sample (graph$from_id, size = 100)
to <- sample (graph$to_id, size = 50)
d <- dodgr_dists (graph, from = from, to = to)
# d is a 100-by-50 matrix of distances between `from` and `to`

## Not run:
# a more complex street network example, thanks to @chrijo; see
# https://github.com/ATFutures/dodgr/issues/47
xy <- rbind (c (7.005994, 51.45774), # limbeckerplatz 1 essen germany
             c (7.012874, 51.45041)) # hauptbahnhof essen germany
xy <- data.frame (lon = xy [, 1], lat = xy [, 2])
essen <- dodgr_streetnet (pts = xy, expand = 0.2, quiet = FALSE)
graph <- weight_streetnet (essen, wt_profile = "foot")
d <- dodgr_dists (graph, from = xy, to = xy)
# First reason why this does not work is because the graph has multiple,
# disconnected components.
table (graph$component)
# reduce to largest connected component, which is always number 1
graph <- graph [which (graph$component == 1),]
d <- dodgr_dists (graph, from = xy, to = xy)
# should work, but even then note that
table (essen$level)
# There are parts of the network on different building levels (because of
# shopping malls and the like). These may or may not be connected, so it may
# be necessary to filter out particular levels
index <- which (! (essen$level == "-1" | essen$level == "1")) # for example
library (sf) # needed for following sub-select operation
essen <- essen [index,]
graph <- weight_streetnet (essen, wt_profile = "foot")
graph <- graph [which (graph$component == 1),]
d <- dodgr_dists (graph, from = xy, to = xy)

## End(Not run)
dodgr_dists

Description

Calculate matrix of pair-wise distances between points.

Usage

dodgr_dists(
  graph,
  from = NULL,
  to = NULL,
  shortest = TRUE,
  pairwise = FALSE,
  heap = "BHeap",
  parallel = TRUE,
  quiet = TRUE
)

Arguments

graph   data.frame or equivalent object representing the network graph (see Notes)
from    Vector or matrix of points from which route distances are to be calculated (see Notes)
to      Vector or matrix of points to which route distances are to be calculated (see Notes)
shortest    If FALSE, calculate distances along the fastest rather than shortest routes (see Notes).
pairwise   If TRUE, calculate distances only between the ordered pairs of from and to.
heap      Type of heap to use in priority queue. Options include Fibonacci Heap (default; FHeap), Binary Heap (BHeap), Trinomial Heap (TriHeap), Extended Trinomial Heap (TriHeapExt), and 2-3 Heap (Heap23').
parallel  If TRUE, perform routing calculation in parallel (see details)
quiet     If FALSE, display progress messages on screen.

Value

square matrix of distances between nodes

Note

graph must minimally contain three columns of from, to, dist. If an additional column named weight or wt is present, shortest paths are calculated according to values specified in that column; otherwise according to dist values. Either way, final distances between from and to points are calculated by default according to values of dist. That is, paths between any pair of points will be calculated according to the minimal total sum of weight values (if present), while reported distances will be total sums of dist values.

For street networks produced with weight_streetnet, distances may also be calculated along the fastest routes with the shortest = FALSE option. Graphs must in this case have columns of time.
and time_weighted. Note that the fastest routes will only be approximate when derived from sf-format data generated with the `osmdata` function `osmdata_sf()`, and will be much more accurate when derived from sc-format data generated with `osmdata_sc()`. See `weight_streetnet` for details.

The `from` and `to` columns of `graph` may be either single columns of numeric or character values specifying the numbers or names of graph vertices, or combinations to two columns specifying geographical (longitude and latitude) coordinates. In the latter case, almost any sensible combination of names will be accepted (for example, fromx, fromy, from_x, from_y, or fr_lat, fr_lon.)

`from` and `to` values can be either two-column matrices of equivalent of longitude and latitude coordinates, or else single columns precisely matching node numbers or names given in `graph$from` or `graph$to`. If `to` is NULL, pairwise distances are calculated between all points specified in `from`. If both `from` and `to` are NULL, pairwise distances are calculated between all nodes in `graph`.

Calculations in parallel (`parallel = TRUE`) ought very generally be advantageous. For small graphs, calculating distances in parallel is likely to offer relatively little gain in speed, but increases from parallel computation will generally markedly increase with increasing graph sizes. By default, parallel computation uses the maximal number of available cores or threads. This number can be reduced by specifying a value via `RcppParallel::setThreadOptions(numThreads = <desired_number>).

### Examples

```r
# A simple graph
graph <- data.frame (from = c("A", "B", "B", "B", "C", "C", "D", "D"),
                     d = c(1, 2, 1, 3, 2, 1, 2, 1))
dodgr_dists (graph)

# A larger example from the included [hampi()] data.
graph <- weight_streetnet (hampi)
from <- sample (graph$from_id, size = 100)
to <- sample (graph$to_id, size = 50)
d <- dodgr_dists (graph, from = from, to = to)
# d is a 100-by-50 matrix of distances between `from` and `to`

# Not run:
# a more complex street network example, thanks to @chrijo; see
# https://github.com/ATFutures/dodgr/issues/47
xy <- rbnd (c (7.005994, 51.45774), # limbeckerplatz 1 essen germany
            c (7.012874, 51.45041)) # hauptbahnhof essen germany
xy <- data.frame (lon = xy [, 1], lat = xy [, 2])
essen <- dodgr_streetnet (pts = xy, expand = 0.2, quiet = FALSE)
graph <- weight_streetnet (essen, wt_profile = "foot")
d <- dodgr_dists (graph, from = xy, to = xy)
# First reason why this does not work is because the graph has multiple,
# disconnected components.
table (graph$component)
# reduce to largest connected component, which is always number 1
graph <- graph [which (graph$component == 1), ]
d <- dodgr_dists (graph, from = xy, to = xy)
# should work, but even then note that
table (essen$level)
```
# There are parts of the network on different building levels (because of
# shopping malls and the like). These may or may not be connected, so it may
# be necessary to filter out particular levels
index <- which (! (essen$level == "-1" | essen$level == "1")) # for example
library (sf) # needed for following sub-select operation
essen <- essen [index, ]
graph <- weight_streetnet (essen, wt_profile = "foot")
graph <- graph [which (graph$component == 1), ]
d <- dodgr_dists (graph, from = xy, to = xy)

## End(Not run)

dodgr_flowmap

---

**dodgr_flowmap**

**Description**

Map the output of *dodgr_flows_aggregate* or *dodgr_flows_disperse*

**Usage**

```
dodgr_flowmap(net, bbox = NULL, linescale = 1)
```

**Arguments**

- **net**
  - A street network with a `flow` column obtained from *dodgr_flows_aggregate* or
    *dodgr_flows_disperse*

- **bbox**
  - If given, scale the map to this bbox, otherwise use entire extend of net

- **linescale**
  - Maximal thickness of plotted lines

**Note**

`net` should be first passed through `merge_directed_graph` prior to plotting, otherwise lines for
different directions will be overlaid.

**Examples**

```
graph <- weight_streetnet (hampi)
from <- sample (graph$from_id, size = 10)
to <- sample (graph$to_id, size = 5)
to <- to [!to %in% from]
flows <- matrix (10 * runif (length (from) * length (to)),
               nrow = length (from))
graph <- dodgr_flows_aggregate (graph, from = from, to = to, flows = flows)
# graph then has an additional 'flows' column of aggregate flows along all
# edges. These flows are directed, and can be aggregated to equivalent
# undirected flows on an equivalent undirected graph with:
graph_undir <- merge_directed_graph (graph)
## Not run:
```
dodgr_flows_aggregate (graph_undir)

## End(Not run)

dodgr_flows_aggregate
dodgr_flows_aggregate

### Description

Aggregate flows throughout a network based on an input matrix of flows between all pairs of from and to points.

### Usage

```r
dodgr_flows_aggregate(
  graph,
  from,
  to,
  flows,
  contract = FALSE,
  heap = "BHeap",
  tol = 0.000000000001,
  norm_sums = TRUE,
  quiet = TRUE
)
```

### Arguments

- **graph**: data.frame or equivalent object representing the network graph (see Details)
- **from**: Vector or matrix of points from which aggregate flows are to be calculated (see Details)
- **to**: Vector or matrix of points to which aggregate flows are to be calculated (see Details)
- **flows**: Matrix of flows with `nrow(flows)==length(from)` and `ncol(flows)==length(to)`.  
- **contract**: If TRUE, calculate flows on contracted graph before mapping them back on to the original full graph (recommended as this will generally be much faster).  
- **heap**: Type of heap to use in priority queue. Options include Fibonacci Heap (default; FHeap), Binary Heap (BHeap), Trinomial Heap (TriHeap), Extended Trinomial Heap (TriHeapExt), and 2-3 Heap (Heap23).  
- **tol**: Relative tolerance below which flows towards to vertices are not considered.  
  This will generally have no effect, but can provide speed gains when flow matrices represent spatial interaction models, in which case this parameter effectively reduces the radius from each from point over which flows are aggregated. To remove any such effect, set `tol = 0`.  
- **norm_sums**: Standardise sums from all origin points, so sum of flows throughout entire network equals sum of densities from all origins (see Note).  
- **quiet**: If FALSE, display progress messages on screen.
Value

Modified version of graph with additional flow column added.

Note

Spatial Interaction models are often fitted through trialling a range of values of 'k'. The specification above allows fitting multiple values of 'k' to be done with a single call, in a way that is far more efficient than making multiple calls. A matrix of 'k' values may be entered, with each column holding a different vector of values, one for each 'from' point. For a matrix of 'k' values having 'n' columns, the return object will be a modified version in the input 'graph', with an additional 'n' columns, named 'flow1', 'flow2', ... up to 'n'. These columns must be subsequently matched by the user back on to the corresponding columns of the matrix of 'k' values.

The norm_sums parameter should be used whenever densities at origins and destinations are absolute values, and ensures that the sum of resultant flow values throughout the entire network equals the sum of densities at all origins. For example, with norm_sums = TRUE (the default), a flow from a single origin with density one to a single destination along two edges will allocate flows of one half to each of those edges, such that the sum of flows across the network will equal one, or the sum of densities from all origins. The norm_sums = TRUE option is appropriate where densities are relative values, and ensures that each edge maintains relative proportions. In the above example, flows along each of two edges would equal one, for a network sum of two, or greater than the sum of densities.

With norm_sums = TRUE, the sum of network flows (sum(output$flow)) should equal the sum of origin densities (sum(dens_from)). This may nevertheless not always be the case, because origin points may simply be too far from any destination (to) points for an exponential model to yield non-zero values anywhere in a network within machine tolerance. Such cases may result in sums of output flows being less than sums of input densities.

Flows are calculated by default using parallel computation with the maximal number of available cores or threads. This number can be reduced by specifying a value via RcppParallel::setThreadOptions (numThreads = <desired_number>).

Examples

```r
graph <- weight_streetnet (hampi)
from <- sample (graph$from_id, size = 10)
to <- sample (graph$to_id, size = 5)
to <- to [!to %in% from]
flows <- matrix (10 * runif (length (from) * length (to)),
               nrow = length (from))
graph <- dodgr_flows_aggregate (graph, from = from, to = to, flows = flows)
# graph then has an additional 'flows' column of aggregate flows along all # edges. These flows are directed, and can be aggregated to equivalent # undirected flows on an equivalent undirected graph with:
graph_undir <- merge_directed_graph (graph)
# This graph will only include those edges having non-zero flows, and so:
nrow (graph); nrow (graph_undir) # the latter is much smaller

# The following code can be used to convert the resultant graph to an `sf`
# object suitable for plotting
## Not run:
gsf <- dodgr_to_sf (graph_undir)
```
```r
# example of plotting with the 'mapview' package
library (mapview)
flow <- gsf$flow / max (gsf$flow)
ncols <- 30
cols <- colorRampPalette (c ("lawngreen", "red")) (ncols) [ceiling (ncols * flow)]
mapview (gsf, color = cols, lwd = 10 * flow)

## End(Not run)

# An example of flow aggregation across a generic (non-OSM) highway,
# represented as the 'routes_fast' object of the \pkg{stplanr} package,
# which is a SpatialLinesDataFrame containing commuter densities along
# components of a street network.
## Not run:
library (stplanr)
# merge all of the 'routes_fast' lines into a single network
r <- overline (routes_fast, attrib = "length", buff_dist = 1)
r <- sf::st_as_sf (r)
# then extract the start and end points of each of the original 'routes_fast'
# lines and use these for routing with 'dodgr'
l <- lapply (routes_fast@lines, function (i)
  c (sp::coordinates (i) [[1]] [1,],
     tail (sp::coordinates (i) [[1]], 1)))
l <- do.call (rbind, l)
xy_start <- l [, 1:2]
xy_end <- l [, 3:4]
# Then just specify a generic OD matrix with uniform values of 1:
flows <- matrix (1, nrow = nrow (l), ncol = nrow (l))
# We need to specify both a 'type' and 'id' column for the
# \link{weight_streetnet} function.
r$type <- 1
r$id <- seq (nrow (r))
graph <- weight_streetnet (r, type_col = "type", id_col = "id",
                          wt_profile = 1)
f <- dodgr_flows_aggregate (graph, from = xy_start, to = xy_end,
                           flows = flows)
# Then merge directed flows and convert to \pkg{sf} for plotting as before:
f <- merge_directed_graph (f)
geoms <- dodgr_to_sfc (f)
gc <- dodgr_contract_graph (f)
gsf <- sf::st_sf (geoms)
gsf$flow <- gc$flow
# sf plot:
plot (gsf ["flow"])

## End(Not run)
```
**dodgr_flows_disperse**

**Description**

Disperse flows throughout a network based on a input vectors of origin points and associated densities

**Usage**

```r
dodgr_flows_disperse(
  graph,  
  from,   
  dens,   
  k = 500,  
  contract = FALSE,  
  heap = "BHeap",  
  tol = 0.000000000001,  
  quiet = TRUE
)
```

**Arguments**

- `graph` data.frame or equivalent object representing the network graph (see Details)
- `from` Vector or matrix of points from which aggregate dispersed flows are to be calculated (see Details)
- `dens` Vectors of densities corresponding to the from points
- `k` Width coefficient of exponential diffusion function defined as \( \exp(-d/k) \), in units of distance column of graph (metres by default). Can also be a vector with same length as from, giving dispersal coefficients from each point. If value of \( k<0 \) is given, a standard logistic polynomial will be used.
- `contract` If TRUE, calculate flows on contracted graph before mapping them back on to the original full graph (recommended as this will generally be much faster).
- `heap` Type of heap to use in priority queue. Options include Fibonacci Heap (default; FHeap), Binary Heap (BHeap), Trinomial Heap (TriHeap), Extended Trinomial Heap (TriHeapExt), and 2-3 Heap (Heap23).
- `tol` Relative tolerance below which dispersal is considered to have finished. This parameter can generally be ignored; if in doubt, its effect can be removed by setting \( tol = 0 \).
- `quiet` If FALSE, display progress messages on screen.

**Value**

Modified version of graph with additional flow column added.

**Note**

Spatial Interaction models are often fitted through trialling a range of values of 'k'. The specification above allows fitting multiple values of 'k' to be done with a single call, in a way that is far more efficient than making multiple calls. A matrix of 'k' values may be entered, with each column holding a different vector of values, one for each 'from' point. For a matrix of 'k' values having
'n' columns, the return object will be a modified version in the input 'graph', with an additional 'n' columns, named 'flow1', 'flow2', ... up to 'n'. These columns must be subsequently matched by the user back on to the corresponding columns of the matrix of 'k' values.

Examples

```r
graph <- weight_streetnet (hampi)
from <- sample (graph$from_id, size = 10)
dens <- rep (1, length (from)) # Uniform densities
graph <- dodgr_flows_disperse (graph, from = from, dens = dens)
# graph then has an additional 'flows' column of aggregate flows along all
# edges. These flows are directed, and can be aggregated to equivalent
# undirected flows on an equivalent undirected graph with:
graphUndir <- merge_directed_graph (graph)
```

**Description**

Aggregate flows throughout a network based using an exponential Spatial Interaction (SI) model between a specified set of origin and destination points, and associated vectors of densities.

**Usage**

```r
dodgr_flows_si(
  graph, from, to,
  k = 500,
  dens_from = NULL, dens_to = NULL,
  contract = FALSE, norm_sums = TRUE,
  heap = "BHeap",
  tol = 0.00000000001,
  quiet = TRUE
)
```

**Arguments**

- `graph` data.frame or equivalent object representing the network graph (see Details)
- `from` Vector or matrix of points from which aggregate flows are to be calculated (see Details)
- `to` Vector or matrix of points to which aggregate flows are to be calculated (see Details)
Width of exponential spatial interaction function \( \exp(-d / k) \), in units of 'd', specified in one of 3 forms: (i) a single value; (ii) a vector of independent values for each origin point (with same length as 'from' points); or (iii) an equivalent matrix with each column holding values for each 'from' point, so 'nrow(k)==length(from)'. See Note.

Vector of densities at origin ('from') points

Vector of densities at destination ('to') points

If TRUE, calculate flows on contracted graph before mapping them back on to the original full graph (recommended as this will generally be much faster).

Standardise sums from all origin points, so sum of flows throughout entire network equals sum of densities from all origins (see Note).

Type of heap to use in priority queue. Options include Fibonacci Heap (default; FHeap), Binary Heap (BHeap), Trinomial Heap (TriHeap), Extended Trinomial Heap (TriHeapExt), and 2-3 Heap (Heap23).

Relative tolerance below which flows towards 'to' vertices are not considered. This will generally have no effect, but can provide speed gains when flow matrices represent spatial interaction models, in which case this parameter effectively reduces the radius from each 'from' point over which flows are aggregated. To remove any such effect, set tol = 0.

If FALSE, display progress messages on screen.

Modified version of graph with additional flow column added.

Spatial Interaction models are often fitted through trialling a range of values of 'k'. The specification above allows fitting multiple values of 'k' to be done with a single call, in a way that is far more efficient than making multiple calls. A matrix of 'k' values may be entered, with each column holding a different vector of values, one for each 'from' point. For a matrix of 'k' values having 'n' columns, the return object will be a modified version in the input 'graph', with an additional 'n' columns, named 'flow1', 'flow2', .. up to 'n'. These columns must be subsequently matched by the user back on to the corresponding columns of the matrix of 'k' values.

The norm_sums parameter should be used whenever densities at origins and destinations are absolute values, and ensures that the sum of resultant flow values throughout the entire network equals the sum of densities at all origins. For example, with norm_sums = TRUE (the default), a flow from a single origin with density one to a single destination along two edges will allocate flows of one half to each of those edges, such that the sum of flows across the network will equal one, or the sum of densities from all origins. The norm_sums = TRUE option is appropriate where densities are relative values, and ensures that each edge maintains relative proportions. In the above example, flows along each of two edges would equal one, for a network sum of two, or greater than the sum of densities.

With norm_sums = TRUE, the sum of network flows (sum(output$flow)) should equal the sum of origin densities (sum(dens_from)). This may nevertheless not always be the case, because origin points may simply be too far from any destination ('to') points for an exponential model to yield
non-zero values anywhere in a network within machine tolerance. Such cases may result in sums of output flows being less than sums of input densities.

Examples

```r
graph <- weight_streetnet (hampi)
from <- sample (graph$from_id, size = 10)
to <- sample (graph$to_id, size = 5)
to <- to [!to %in% from]
flows <- matrix (10 * runif (length (from) * length (to)),
                   nrow = length (from))
graph <- dodgr_flows_aggregate (graph, from = from, to = to, flows = flows)
# graph then has an additional 'flows' column of aggregate flows along all
# edges. These flows are directed, and can be aggregated to equivalent
# undirected flows on an equivalent undirected graph with:
graph_undir <- merge_directed_graph (graph)
# This graph will only include those edges having non-zero flows, and so:
nrow (graph); nrow (graph_undir) # the latter is much smaller
```

---

dodgr_full_cycles  dodgr_full_cycles

Description

Calculate fundamental cycles on a FULL (that is, non-contracted) graph.

Usage

```r
dodgr_full_cycles(graph, graph_max_size = 10000, expand = 0.05)
```

Arguments

- **graph**: data.frame or equivalent object representing the contracted network graph (see Details).
- **graph_max_size**: Maximum size submitted to the internal C++ routines as a single chunk. Warning: Increasing this may lead to computer meltdown!
- **expand**: For large graphs which must be broken into chunks, this factor determines the relative overlap between chunks to ensure all cycles are captured. (This value should only need to be modified in special cases.)

Note

This function converts the graph to its contracted form, calculates the fundamental cycles on that version, and then expands these cycles back onto the original graph. This is far more computationally efficient than calculating fundamental cycles on a full (non-contracted) graph.
**Examples**

```r
## Not run:
net <- weight_streetnet (hampi)
graph <- dodgr_contract_graph (net)
cyc1 <- dodgr_fundamental_cycles (graph)
cyc2 <- dodgr_full_cycles (net)
## End(Not run)

# cyc2 has same number of cycles, but each one is generally longer, through
# including all points intermediate to junctions; cyc1 has cycles composed of
# junction points only.
```

**dodgr_fundamental_cycles**

**Description**

Calculate fundamental cycles in a graph.

**Usage**

```r
dodgr_fundamental_cycles(
  graph,
  vertices = NULL,
  graph_max_size = 10000,
  expand = 0.05
)
```

**Arguments**

- `graph` : data.frame or equivalent object representing the contracted network graph (see Details).
- `vertices` : data.frame returned from `dodgr_vertices(graph)`. Will be calculated if not provided, but it's quicker to pass this if it has already been calculated.
- `graph_max_size` : Maximum size submitted to the internal C++ routines as a single chunk. Warning: Increasing this may lead to computer meltdown!
- `expand` : For large graphs which must be broken into chunks, this factor determines the relative overlap between chunks to ensure all cycles are captured. (This value should only need to be modified in special cases.)

**Value**

List of cycle paths, in terms of vertex IDs in graph and, for spatial graphs, the corresponding coordinates.
Note

Calculation of fundamental cycles is VERY computationally demanding, and this function should only be executed on CONTRACTED graphs (that is, graphs returned from `dodgr_contract_graph`), and even than may take a long time to execute. Results for full graphs can be obtained with the function `dodgr_full_cycles`. The computational complexity can also not be calculated in advance, and so the parameter `graph_max_size` will lead to graphs larger than that (measured in numbers of edges) being cut into smaller parts. (Note that this is only possible for spatial graphs, meaning that it is not at all possible to apply this function to large, non-spatial graphs.) Each of these smaller parts will be expanded by the specified amount (`expand`), and cycles found within. The final result is obtained by aggregating all of these cycles and removing any repeated ones arising due to overlap in the expanded portions. Finally, note that this procedure of cutting graphs into smaller, computationally manageable sub-graphs provides only an approximation and may not yield all fundamental cycles.

Examples

```r
net <- weight_streetnet(hampi)
graph <- dodgr_contract_graph(net)
verts <- dodgr_vertices(graph)
cyc <- dodgr_fundamental_cycles(graph, verts)
```

---

**Description**

Insert a new node or vertex into a network

**Usage**

```r
dodgr_insert_vertex(graph, v1, v2, x = NULL, y = NULL)
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>graph</td>
<td>A flat table of graph edges. Must contain columns labelled <code>from</code> and <code>to</code>, or <code>start</code> and <code>stop</code>. May also contain similarly labelled columns of spatial coordinates (for examples <code>from_x</code>) or <code>stop_lon</code>.</td>
</tr>
<tr>
<td>v1</td>
<td>Vertex defining start of graph edge along which new vertex is to be inserted</td>
</tr>
<tr>
<td>v2</td>
<td>Vertex defining end of graph edge along which new vertex is to be inserted (order of v1 and v2 is not important).</td>
</tr>
<tr>
<td>x</td>
<td>The x-coordinate of new vertex. If not specified, vertex is created half-way between v1 and v2.</td>
</tr>
<tr>
<td>y</td>
<td>The y-coordinate of new vertex. If not specified, vertex is created half-way between v1 and v2.</td>
</tr>
</tbody>
</table>
Value

A modified graph with specified edge between defined start and end vertices split into two edges either side of new vertex.

Examples

graph <- weight_streetnet (hampi)
e1 <- sample (nrow (graph), 1)
v1 <- graph$from_id [e1]
v2 <- graph$to_id [e1]
# insert new vertex in the middle of that randomly-selected edge:
graph2 <- dodgr_insert_vertex (graph, v1, v2)
nrow (graph); nrow (graph2) # new edges added to graph2

Description

Calculate isochrone contours from specified points. Function is fully vectorized to calculate accept vectors of central points and vectors defining multiple isochrone thresholds.

Usage

dodgr_isochrones(graph, from = NULL, tlim = NULL, heap = "BHeap")

Arguments

graph  data.frame or equivalent object representing the network graph (see Notes)
from  Vector or matrix of points from which isochrones are to be calculated.
tlim  Vector of desired limits of isochrones in seconds
heap  Type of heap to use in priority queue. Options include Fibonacci Heap (default; FHeap), Binary Heap (BHeap), Trinomial Heap (TriHeap), Extended Trinomial Heap (TriHeapExt), and 2-3 Heap (Heap23).

Value

A single data.frame of isochrones as points sorted anticlockwise around each origin (from) point, with columns denoting the from points and tlim value(s). The isochrones are given as id values and associated coordinates of the series of points from each from point at the specified isochrone times.

Isochrones are calculated by default using parallel computation with the maximal number of available cores or threads. This number can be reduced by specifying a value via RcppParallel::setThreadOptions (numThreads = <...>).
Examples

```r
## Not run:
# Use osmdata package to extract 'SC'-format data:
library (osmdata)

dat <- opq("hampi india") %>%
  add_osm_feature (key = "highway") %>%
  osmdata_sc ()

graph <- weight_streetnet (dat)
from <- sample (graph$.vx0, size = 100)
tlim <- c (5, 10, 20, 30, 60) * 60 # times in seconds
x <- dodgr_isochrones (graph, from = from, tlim)

## End(Not run)
```

### Description

Calculate isodistance contours from specified points. Function is fully vectorized to calculate accept vectors of central points and vectors defining multiple isodistances.

### Usage

```r
dodgr_isodists(graph, from = NULL, dlim = NULL, heap = "BHeap")
```

### Arguments

- **graph**: data.frame or equivalent object representing the network graph (see Notes)
- **from**: Vector or matrix of points from which isodistances are to be calculated.
- **dlim**: Vector of desired limits of isodistances in metres.
- **heap**: Type of heap to use in priority queue. Options include Fibonacci Heap (default; FHeap), Binary Heap (BHeap), Trinomial Heap (TriHeap), Extended Trinomial Heap (TriHeapExt, and 2-3 Heap (Heap23)).

### Value

A single data.frame of isodistances as points sorted anticlockwise around each origin (from) point, with columns denoting the from points and dlim value(s). The isodistance contours are given as id values and associated coordinates of the series of points from each from point at the specified isodistances.

### Note

Isodists are calculated by default using parallel computation with the maximal number of available cores or threads. This number can be reduced by specifying a value via RcppParallel::setThreadOptions (numThreads = <desired_number>).
**dodgr_isoverts**

**Examples**

```r
# Example code
graph <- weight_streetnet (hampi)
from <- sample (graph$from_id, size = 100)
dlim <- c (1, 2, 5, 10, 20) * 100
d <- dodgr_isodists (graph, from = from, dlim)
```

---

**Description**

Calculate isodistance or isochrone contours from specified points, and return lists of all network vertices contained within the contours. Function is fully vectorized to calculate accept vectors of central points and vectors defining multiple isochrone thresholds. Provide one or more dlim values for isodistances, or one or more tlim values for isochrones.

**Usage**

```r
dodgr_isoverts (graph, from = NULL, dlim = NULL, tlim = NULL, heap = "BHeap")
```

**Arguments**

- `graph`: data.frame or equivalent object representing the network graph (see Notes)
- `from`: Vector or matrix of points from which isodistances or isochrones are to be calculated.
- `dlim`: Vector of desired limits of isodistances in metres.
- `tlim`: Vector of desired limits of isochrones in seconds
- `heap`: Type of heap to use in priority queue. Options include Fibonacci Heap (default; FHeap), Binary Heap (BHeap), Trinomial Heap (TriHeap), Extended Trinomial Heap (TriHeapExt), and 2-3 Heap (Heap23).

**Value**

A single data.frame of vertex IDs, with columns denoting the from points and tlim value(s). The isochrones are given as id values and associated coordinates of the series of points from each from point at the specified isochrone times.

Isoverts are calculated by default using parallel computation with the maximal number of available cores or threads. This number can be reduced by specifying a value via RcppParallel::setThreadOptions (numThreads = <desired_number>).

**Examples**

```r
# Not run:
# Use osmdata package to extract 'SC'-format data:
library (osmdata)
dat <- opq ("hampi india") %>%
  add_osm_feature (key = "highway") %>%
```
osmdata_sc()
graph <- weight_streetnet(dat)
from <- sample(graph$.vx0, size = 100)
tlim <- c(5, 10, 20, 30, 60) * 60 # times in seconds
x <- dodgr_isoverts(graph, from = from, tlim)

## End(Not run)

dodgr_paths  dodgr_paths

**Description**

Calculate lists of pair-wise shortest paths between points.

**Usage**

```r
dodgr_paths(
  graph,
  from,
  to,
  vertices = TRUE,
  pairwise = FALSE,
  heap = "BHeap",
  quiet = TRUE
)
```

**Arguments**

- `graph`: data.frame or equivalent object representing the network graph (see Details)
- `from`: Vector or matrix of points from which route paths are to be calculated (see Details)
- `to`: Vector or matrix of points to which route paths are to be calculated (see Details)
- `vertices`: If TRUE, return lists of lists of vertices for each path, otherwise return corresponding lists of edge numbers from graph.
- `pairwise`: If TRUE, calculate paths only between the ordered pairs of from and to. In this case, each of these must be the same length, and the output will contain paths the i-th members of each, and thus also be of that length.
- `heap`: Type of heap to use in priority queue. Options include Fibonacci Heap (default; FHeap), Binary Heap (BHeap), Radix, Trinomial Heap (TriHeap), Extended Trinomial Heap (TriHeapExt, and 2-3 Heap (Heap23).
- `quiet`: If FALSE, display progress messages on screen.

**Value**

List of list of paths tracing all connections between nodes such that if `x <- dodgr_paths(graph, from, to)`, then the path between `from[i]` and `to[j]` is `x[[i]][[j]]`.
Note

graph must minimally contain four columns of from, to, dist. If an additional column named weight or wt is present, shortest paths are calculated according to values specified in that column; otherwise according to dist values. Either way, final distances between from and to points are calculated according to values of dist. That is, paths between any pair of points will be calculated according to the minimal total sum of weight values (if present), while reported distances will be total sums of dist values.

The from and to columns of graph may be either single columns of numeric or character values specifying the numbers or names of graph vertices, or combinations to two columns specifying geographical (longitude and latitude) coordinates. In the latter case, almost any sensible combination of names will be accepted (for example, fromx, fromy, from_x, from_y, or fr_lat, fr_lon.)

from and to values can be either two-column matrices of equivalent of longitude and latitude coordinates, or else single columns precisely matching node numbers or names given in graph$from or graph$to. If to is missing, pairwise distances are calculated between all points specified in from. If neither from nor to are specified, pairwise distances are calculated between all nodes in graph.

Examples

```r
graph <- weight_streetnet (hampi)
from <- sample (graph$from_id, size = 100)
to <- sample (graph$to_id, size = 50)
dp <- dodgr_paths (graph, from = from, to = to)
# dp is a list with 100 items, and each of those 100 items has 30 items, each
# of which is a single path listing all vertex IDs as taken from 'graph'.

# it is also possible to calculate paths between pairwise start and end
# points
from <- sample (graph$from_id, size = 5)
to <- sample (graph$to_id, size = 5)
dp <- dodgr_paths (graph, from = from, to = to, pairwise = TRUE)
# dp is a list of 5 items, each of which just has a single path between each
# pairwise from and to point.
```

Description

Sample a random but connected sub-component of a graph

Usage

```r
dodgr_sample(graph, nverts = 1000)
```
Arguments

- **graph**: A flat table of graph edges. Must contain columns labelled `from` and `to`, or `start` and `stop`. May also contain similarly labelled columns of spatial coordinates (for example `from_x`) or `stop_lon`.
- **nverts**: Number of vertices to sample

Value

A connected sub-component of `graph`

Note

Graphs may occasionally have `nverts + 1` vertices, rather than the requested `nverts`.

Examples

```r
graph <- weight_streetnet(hampi)
nrow(graph) # 5,742
graph <- dodgr_sample(graph, nverts = 200)
nrow(graph) # generally around 400 edges
nrow(dodgr_vertices(graph)) # 200
```

---

dodgr_sflines_to_poly  dodgr_sflines_to_poly

Description

Convert `sf` `LINESTRING` objects to `POLYGON` objects representing all fundamental cycles within the `LINESTRING` objects.

Usage

```
dodgr_sflines_to_poly(sflines, graph_max_size = 10000, expand = 0.05)
```

Arguments

- **sflines**: An `sf` `LINESTRING` object representing a network.
- **graph_max_size**: Maximum size submitted to the internal C++ routines as a single chunk. Warning: Increasing this may lead to computer meltdown!
- **expand**: For large graphs which must be broken into chunks, this factor determines the relative overlap between chunks to ensure all cycles are captured. (This value should only need to be modified in special cases.)

Value

An `sf::sfc` collection of `POLYGON` objects.
**dodgr_streetnet**

---

**Description**

Use the osmdata package to extract the street network for a given location. For routing between a given set of points (passed as `pts`), the `bbox` argument may be omitted, in which case a bounding box will be constructed by expanding the range of `pts` by the relative amount of `expand`.

**Usage**

```r
dodgr_streetnet(bbox, pts = NULL, expand = 0.05, quiet = TRUE)
```

**Arguments**

- **bbox**: Bounding box as vector or matrix of coordinates, or location name. Passed to `osmdata::getbb`.
- **pts**: List of points presumably containing spatial coordinates
- **expand**: Relative factor by which street network should extend beyond limits defined by `pts` (only if `bbox` not given).
- **quiet**: If `FALSE`, display progress messages

**Value**

A Simple Features (`sf`) object with coordinates of all lines in the street network.

**Examples**

```r
## Not run:
streetnet <- dodgr_streetnet("hampi india", expand = 0)
# convert to form needed for `dodgr` functions:
graph <- weight_streetnet(streetnet)
nrow(graph) # around 5,900 edges

# Alternative ways of extracting street networks by using a small selection
# of graph vertices to define bounding box:
verts <- dodgr_vertices(graph)
verts <- verts[sample(nrow(verts), size = 200),]
streetnet <- dodgr_streetnet(pts = verts, expand = 0)
graph <- weight_streetnet(streetnet)
nrow(graph)
# This will generally have many more rows because most street networks
# include streets that extend considerably beyond the specified bounding box.

# bbox can also be a polygon:
bb <- osmdata::getbb("gent belgium") # rectangular bbox
nrow(dodgr_streetnet(bbox = bb)) # around 30,000
bb <- osmdata::getbb("gent belgium", format_out = "polygon")
nrow(dodgr_streetnet(bbox = bb)) # around 17,000
```
# The latter has fewer rows because only edges within polygon are returned

## End(Not run)

---

**dodgr_streetnet_sc  dodgr_streetnet_sc**

### Description

Use the osmdata package to extract the street network for a given location and return it in SC-format. For routing between a given set of points (passed as `pts`), the `bbox` argument may be omitted, in which case a bounding box will be constructed by expanding the range of `pts` by the relative amount of `expand`.

### Usage

```r
dodgr_streetnet_sc(bbox, pts = NULL, expand = 0.05, quiet = TRUE)
```

### Arguments

- **bbox**: Bounding box as vector or matrix of coordinates, or location name. Passed to osmdata::getbb.
- **pts**: List of points presumably containing spatial coordinates
- **expand**: Relative factor by which street network should extend beyond limits defined by `pts` (only if `bbox` not given).
- **quiet**: If `FALSE`, display progress messages

### Value

A Simple Features (sf) object with coordinates of all lines in the street network.

### Examples

```r
## Not run:
streetnet <- dodgr_streetnet ("hampi india", expand = 0)
# convert to form needed for `dodgr` functions:
graph <- weight_streetnet (streetnet)
nrow (graph) # around 5,900 edges
# Alternative ways of extracting street networks by using a small selection
# of graph vertices to define bounding box:
verts <- dodgr_vertices (graph)
verts <- verts [sample (nrow (verts), size = 200), ]
streetnet <- dodgr_streetnet (pts = verts, expand = 0)
graph <- weight_streetnet (streetnet)
nrow (graph)
# This will generally have many more rows because most street networks
# include streets that extend considerably beyond the specified bounding box.
```
# bbox can also be a polygon:
bb <- osmdata::getbb("gent belgium") # rectangular bbox
nrow(dodgr_streetnet(bbox = bb)) # around 30,000
bb <- osmdata::getbb("gent belgium", format_out = "polygon")
nrow(dodgr_streetnet(bbox = bb)) # around 17,000
# The latter has fewer rows because only edges within polygon are returned
## End(Not run)

dodgr_times
dodgr_times
dodgr_times
dodgr_times

### Description
Calculate matrix of pair-wise travel times between points.

### Usage
dodgr_times(graph, from = NULL, to = NULL, shortest = FALSE, heap = "BHeap")

### Arguments
- **graph**: A dodgr network returned from the `weight_streetnet` function using a network obtained with the `osmdata` function, possibly contracted with `dodgr_contract_graph`.
- **from**: Vector or matrix of points from which route distances are to be calculated (see Notes)
- **to**: Vector or matrix of points to which route distances are to be calculated (see Notes)
- **shortest**: If TRUE, calculate times along the shortest rather than fastest paths.
- **heap**: Type of heap to use in priority queue. Options include Fibonacci Heap (default; FHeap), Binary Heap (BHeap), Trinomial Heap (TriHeap), Extended Trinomial Heap (TriHeapExt), and 2:3 Heap (Heap23).

### Value
- square matrix of distances between nodes

### Examples
# A simple graph
graph <- data.frame(from = c("A", "B", "B", "B", "C", "C", "D", "D"),
                    d = c(1, 2, 1, 3, 2, 1, 2, 1))
dodgr_dists(graph)

# A larger example from the included [hampi()] data.
graph <- weight_streetnet(hampi)
from <- sample(graph$from_id, size = 100)
to <- sample (graph$to_id, size = 50)
d <- dodgr_dists (graph, from = from, to = to)
# d is a 100-by-50 matrix of distances between `from` and `to`

## Not run:
# a more complex street network example, thanks to @chrijo; see
# https://github.com/ATFutures/dodgr/issues/47
xy <- rbind (c (7.005994, 51.45774), # limbeckerplatz 1 essen germany
c (7.012874, 51.45041)) # hauptbahnhof essen germany
xy <- data.frame (lon = xy [, 1], lat = xy [, 2])
essen <- dodgr_streetnet (pts = xy, expand = 0.2, quiet = FALSE)
graph <- weight_streetnet (essen, wt_profile = "foot")
d <- dodgr_dists (graph, from = xy, to = xy)
# First reason why this does not work is because the graph has multiple,
# disconnected components.
table (graph$component)
# reduce to largest connected component, which is always number 1
graph <- graph [which (graph$component == 1), ]
d <- dodgr_dists (graph, from = xy, to = xy)
# should work, but even then note that

dodgr_to_igraph

dodgr_to_igraph

**Description**

Convert a dodgr graph to an igraph.

**Usage**

```
dodgr_to_igraph(graph, weight_column = "d")
```

**Arguments**

- **graph**: A dodgr graph
- **weight_column**: The column of the dodgr network to use as the edge weights in the igraph representation.
**Value**

The igraph equivalent of the input. Note that this will *not* be a dual-weighted graph.

**See Also**

igraph_to_dodgr

**Examples**

```r
graph <- weight_streetnet(hampi)
graphi <- dodgr_to_igraph(graph)
```

---

**Description**

Convert a dodgr graph into an equivalent sf object. Works by aggregating edges into LINESTRING objects representing longest sequences between all junction nodes. The resultant objects will generally contain more LINESTRING objects than the original sf object, because the former will be bisected at every junction point.

**Usage**

```r
dodgr_to_sf(graph)
```

**Arguments**

- `graph` A dodgr graph

**Value**

Equivalent object of class sf.

**Note**

Requires the sf package to be installed.

**Examples**

```r
hw <- weight_streetnet(hampi)
nrow(hw) # 5,729 edges
xy <- dodgr_to_sf(hw)
dim(xy) # 764 edges; 14 attributes
```
**dodgr_to_sfc**

**Description**

Convert a dodgr graph into a list composed of two objects: dat, a data.frame; and geometry, an sfc object from the (sf) package. Works by aggregating edges into LINESTRING objects representing longest sequences between all junction nodes. The resultant objects will generally contain more LINESTRING objects than the original sf object, because the former will be bisected at every junction point.

**Usage**

```r
dodgr_to_sfc(graph)
```

**Arguments**

- `graph` A dodgr graph

**Value**

A list containing (1) A data.frame of data associated with the sf geometries; and (ii) A Simple Features Collection (sfc) list of LINESTRING objects.

**Note**

The output of this function corresponds to the edges obtained from dodgr_contract_graph. This function does not require the sf package to be installed; the corresponding function that creates a full sf object - dodgr_to_sf does requires sf to be installed.

**Examples**

```r
hw <- weight_streetnet (hampi)
nrow(hw)
xy <- dodgr_to_sfc (hw)
dim (hw) # 5.845 edges
length (xy$geometry) # more linestrings aggregated from those edges
nrow (hampi) # than the 191 linestrings in original sf object
dim (xy$dat) # same number of rows as there are geometries
# The dodgr_to_sf function then just implements this final conversion:
# sf::st_sf (xy$dat, geometry = xy$geometry, crs = 4326)
```
**dodgr_to_tidygraph**

---

**Description**
Convert a **dodgr** graph to a **tidygraph**.

**Usage**

dodgr_to_tidygraph(graph)

**Arguments**

- **graph**
  A **dodgr** graph

**Value**

The **tidygraph** equivalent of the input

**Examples**

graph <- weight_streetnet(hampi)
grapht <- dodgr_to_tidygraph(graph)

---

**dodgr_uncontract_graph**

---

**Description**
Revert a contracted graph created with **dodgr_contract_graph** back to the full, uncontracted version. This function is mostly used for the side effect of mapping any new columns inserted on the contracted graph back on to the original graph, as demonstrated in the example.

**Usage**

dodgr_uncontract_graph(graph)

**Arguments**

- **graph**
  A contracted graph created from **dodgr_contract_graph**.

**Value**

A single **data.frame** representing the equivalent original, uncontracted graph.
**dodgr_vertices**

### Description

Extract vertices of graph, including spatial coordinates if included

### Usage

```r
dodgr_vertices(graph)
```

### Arguments

- **graph**
  
  A flat table of graph edges. Must contain columns labelled `from` and `to`, or `start` and `stop`. May also contain similarly labelled columns of spatial coordinates (for example `from_x`) or `stop_lon`).

### Value

A `data.frame` of vertices with unique numbers (`n`).

### Note

Values of `n` are 0-indexed

### Examples

```r
graph <- weight_streetnet (hampi)
v <- dodgr_vertices (graph)
```
**Description**

Estimate a value for the 'dist_threshold' parameter of the `dodgr_centrality` function. Providing distance thresholds to this function generally provides considerably speed gains, and results in approximations of centrality. This function enables the determination of values of 'dist_threshold' corresponding to specific degrees of accuracy.

**Usage**

```
estimate_centrality_threshold(graph, tolerance = 0.001)
```

**Arguments**

- `graph` 'data.frame' or equivalent object representing the network graph (see Details)
- `tolerance` Desired maximal degree of inaccuracy in centrality estimates
  - values will be accurate to within this amount, subject to a constant scaling factor. Note that threshold values increase non-linearly with decreasing values of 'tolerance'

**Value**

A single value for 'dist_threshold' giving the required tolerance.

**Note**

This function may take some time to execute. While running, it displays ongoing information on screen of estimated values of 'dist_threshold' and associated errors. Thresholds are progressively increased until the error is reduced below the specified tolerance.

**Description**

The 'dodgr' centrality functions are designed to be applied to potentially very large graphs, and may take considerable time to execute. This helper function estimates how long a centrality function may take for a given graph and given value of 'dist_threshold' estimated via the `estimate_centrality_threshold` function.
Usage

```r
estimate_centrality_time(
  graph,
  contract = TRUE,
  edges = TRUE,
  dist_threshold = NULL,
  heap = "BHeap"
)
```

Arguments

- **graph**: 'data.frame' or equivalent object representing the network graph (see Details)
- **contract**: If 'TRUE', centrality is calculated on contracted graph before mapping back on to the original full graph. Note that for street networks, in particular those obtained from the `osmdata` package, vertex placement is effectively arbitrary except at junctions; centrality for such graphs should only be calculated between the latter points, and thus 'contract' should always be 'TRUE'.
- **edges**: If 'TRUE', centrality is calculated for graph edges, returning the input 'graph' with an additional 'centrality' column; otherwise centrality is calculated for vertices, returning the equivalent of 'dodgr_vertices(graph)', with an additional vertex-based 'centrality' column.
- **dist_threshold**: If not 'NULL', only calculate centrality for each point out to specified threshold. Setting values for this will result in approximate estimates for centrality, yet with considerable gains in computational efficiency. For sufficiently large values, approximations will be accurate to within some constant multiplier. Appropriate values can be established via the `estimate_centrality_threshold` function.
- **heap**: Type of heap to use in priority queue. Options include Fibonacci Heap (default; 'FHeap'), Binary Heap ('BHeap'), Trinomial Heap ('TriHeap'), Extended Trinomial Heap ('TriHeapExt', and 2-3 Heap ('Heap23').

Value

An estimated calculation time for calculating centrality for the given value of 'dist_threshold'

Note

This function may take some time to execute. While running, it displays ongoing information on screen of estimated values of 'dist_threshold' and associated errors. Thresholds are progressively increased until the error is reduced below the specified tolerance.

Description

A sample street network from the township of Hampi, Karnataka, India.
igraph_to_dodgr

Format

A Simple Features sf data.frame containing the street network of Hampi.

Note

Can be re-created with the following command, which also removes extraneous columns to reduce size:

Examples

```r
## Not run:
hampi <- dodgr_streetnet("hampi india")
cols <- c("osm_id", "highway", "oneway", "geometry")
hampi <- hampi [, which (names (hampi) %in% cols)]

## End(Not run)

# this 'sf data.frame' can be converted to a 'dodgr' network with
net <- weight_streetnet (hampi, wt_profile = 'foot')
```

igraph_to_dodgr

Description

Convert a igraph network to an equivalent dodgr representation.

Usage

```r
igraph_to_dodgr(graph)
```

Arguments

- **graph**: An igraph network

Value

The dodgr equivalent of the input.

See Also

dodgr_to_igraph

Examples

```r
graph <- weight_streetnet (hampi)
graph1 <- dodgr_to_igraph (graph)
graph2 <- igraph_to_dodgr (graph1)
identical (graph2, graph) # FALSE
```
**match_points_to_graph**

**Description**

Alias for `match_points_to_graph`

**Usage**

```r
match_points_to_graph(verts, xy, connected = FALSE)
```

**Arguments**

- `verts`: A data.frame of vertices obtained from `dodgr_vertices(graph)`.
- `xy`: coordinates of points to be matched to the vertices, either as matrix or sf-formatted data.frame.
- `connected`: Should points be matched to the same (largest) connected component of graph? If FALSE and these points are to be used for a dodgr routing routine (dodgr_dists, dodgr_paths, or dodgr_flows_aggregate), then results may not be returned if points are not part of the same connected component. On the other hand, forcing them to be part of the same connected component may decrease the spatial accuracy of matching.

**Value**

A vector index into verts

**Examples**

```r
net <- weight_streetnet(hampi, wt_profile = "foot")
verts <- dodgr_vertices(net)
# Then generate some random points to match to graph
npts <- 10
xy <- data.frame(
  x = min(verts$x) + runif(npts) * diff(range(verts$x)),
  y = min(verts$y) + runif(npts) * diff(range(verts$y))
)
pts <- match_pts_to_graph(verts, xy)
pts # an index into verts
pts <- verts$id[pts]
pts # names of those vertices
```
**Description**

Match spatial points to a spatial graph which contains vertex coordinates.

**Usage**

```r
match_pts_to_graph(verts, xy, connected = FALSE)
```

**Arguments**

- `verts`: A `data.frame` of vertices obtained from `dodgr_vertices(graph)`.
- `xy`: Coordinates of points to be matched to the vertices, either as matrix or `sf`-formatted `data.frame`.
- `connected`: Should points be matched to the same (largest) connected component of graph? If `FALSE` and these points are to be used for a dodgr routing routine (`dodgr_dists`, `dodgr_paths`, or `dodgr_flows_aggregate`), then results may not be returned if points are not part of the same connected component. On the other hand, forcing them to be part of the same connected component may decrease the spatial accuracy of matching.

**Value**

A vector index into `verts`.

**Examples**

```r
net <- weight_streetnet(hampi, wt_profile = "foot")
verts <- dodgr_vertices(net)
# Then generate some random points to match to graph
npts <- 10
xy <- data.frame(
  x = min(verts$x) + runif(npts) * diff(range(verts$x)),
  y = min(verts$y) + runif(npts) * diff(range(verts$y))
)
pts <- match_pts_to_graph(verts, xy)
pts # an index into verts
pts <- verts$id[pts]
pts # names of those vertices
```
Description

Merge directed edges into equivalent undirected values by aggregating across directions. This function is primarily intended to aid visualisation of directed graphs, particularly visualising the results of the `dodgr_flows_aggregate` and `dodgr_flows_disperse` functions, which return columns of aggregated flows directed along each edge of a graph.

Usage

```r
merge_directed_graph(graph, col_names = c("flow"))
```

Arguments

- `graph`: A undirected graph in which directed edges of the input graph have been merged through aggregation to yield a single, undirected edge between each pair of vertices.
- `col_names`: Names of columns to be merged through aggregation. Values for these columns in resultant undirected graph will be aggregated from directed values.

Value

An equivalent graph in which all directed edges have been reduced to single, undirected edges, and all values of the specified column(s) have been aggregated across directions to undirected values.

Examples

```r
graph <- weight_streetnet(hampi)
from <- sample(graph$from_id, size = 10)
to <- sample(graph$to_id, size = 5)
to <- to [!to %in% from]
flows <- matrix(10 * runif(length(from) * length(to)),
               nrow = length(from))
graph <- dodgr_flows_aggregate(graph, from = from, to = to, flows = flows)
# graph then has an additional 'flows' column of aggregate flows along all edges. These flows are directed, and can be aggregated to equivalent undirected flows on an equivalent undirected graph with:
graph_undir <- merge_directed_graph(graph)
# This graph will only include those edges having non-zero flows, and so:
nrow(graph); nrow(graph_undir) # the latter is much smaller
```
Description

A sample street network for Bristol, U.K., from the Ordnance Survey.

Format

A Simple Features sf data.frame representing motorways in Bristol, UK.

Note

Input data downloaded from https://www.ordnancesurvey.co.uk/opendatadownload/products.html. To download the data from that page click on the tick box next to 'OS Open Roads', scroll to the bottom, click 'Continue' and complete the form on the subsequent page. This dataset is open access and can be used under the Open Government License and must be cited as follows: Contains OS data © Crown copyright and database right (2017)

Examples

```r
## Not run:
library(sf)
library(dplyr)
# os_roads <- sf::read_sf("~/data/ST_RoadLink.shp") # data must be unzipped here
# u <- "https://opendata.arcgis.com/datasets/686603e943f948acaa13fb5d2b0f1275_4.kml"
# lads <- sf::read_sf(u)
# mapview::mapview(lads)
# bristol_pol <- dplyr::filter(lads, grepl("Bristol", lad16nm))
# os_roads <- st_transform(os_roads, st_crs(lads))
# os_roads_bristol <- os_roads[bristol_pol, ]
# dplyr::filter(class == "Motorway" & roadNumber != "M32")
# st_zm(drop = TRUE)
# mapview::mapview(os_roads_bristol)
## End(Not run)

# Converting this 'sf data.frame' to a 'dodgr' network requires manual
# specification of weighting profile:
# colnm <- "formOfWay" # name of column used to determine weights
# wts <- data.frame (name = "custom",
# way = unique (os_roads_bristol [[colnm]])
# value = c (0.1, 0.2, 0.8, 1))
# net <- weight_streetnet (os_roads_bristol, wt_profile = wts,
# type_col = colnm, id_col = "identifier")
# 'id_col' tells the function which column to use to attribute IDs of ways
```
weighting_profiles

Description
Collection of weighting profiles used to adjust the routing process to different means of transport. Modified from data taken from the Routino project, with additional tables for average speeds, dependence of speed on type of surface, and waiting times in seconds at traffic lights.

Format
List of data.frame objects with profile names, means of transport and weights.

References
https://www.routino.org/xml/routino-profiles.xml

weight_railway

Description
Weight (or re-weight) an sf-formatted OSM street network for routing along railways.

Usage
weight_railway(
  sf_lines,
  type_col = "railway",
  id_col = "osm_id",
  keep_cols = c("maxspeed"),
  excluded = c("abandoned", "disused", "proposed", "razed")
)

Arguments
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf_lines</td>
<td>A street network represented as sf LINESTRING objects, typically extracted with dodgr_streetnet</td>
</tr>
<tr>
<td>type_col</td>
<td>Specify column of the sf data.frame object which designates different types of railways to be used for weighting (default works with osmdata objects).</td>
</tr>
<tr>
<td>id_col</td>
<td>Specify column of the codesf data.frame object which provides unique identifiers for each railway (default works with osmdata objects).</td>
</tr>
<tr>
<td>keep_cols</td>
<td>Vectors of columns from sf_lines to be kept in the resultant dodgr network; vector can be either names or indices of desired columns.</td>
</tr>
<tr>
<td>excluded</td>
<td>Types of railways to exclude from routing.</td>
</tr>
</tbody>
</table>
**Value**

A data.frame of edges representing the rail network, along with a column of graph component numbers.

**Note**

Default railway weighting is by distance. Other weighting schemes, such as by maximum speed, can be implemented simply by modifying the `d_weighted` column returned by this function accordingly.

**Examples**

```r
## Not run:
# sample railway extraction with the 'osmdata' package
library (osmdata)
dat <- opq ("shinjuku") %>%
    add_osm_feature (key = "railway") %>%
    osmdata_sf (quiet = FALSE)
graph <- weight_railway (dat$osm_lines)
## End(Not run)
```

---

**Description**

Weight (or re-weight) an sf or SC (silicate)-formatted OSM street network according to a named profile, selected from (foot, horse, wheelchair, bicycle, moped, motorcycle, motorcar, goods, hgv, psv).

**Usage**

```r
weight_streetnet(
  x,
  wt_profile = "bicycle",
  wt_profile_file = NULL,
  turn_penalty = FALSE,
  type_col = "highway",
  id_col = "osm_id",
  keep_cols = NULL,
  left_side = FALSE
)
```

```r
## Default S3 method:
weight_streetnet(
  x,
  wt_profile = "bicycle",
```
wt_profile_file = NULL,
    turn_penalty = FALSE,
    type_col = "highway",
    id_col = "osm_id",
    keep_cols = NULL,
    left_side = FALSE
)

## S3 method for class 'sf'
weight_streetnet(
    x,
    wt_profile = "bicycle",
    wt_profile_file = NULL,
    turn_penalty = FALSE,
    type_col = "highway",
    id_col = "osm_id",
    keep_cols = NULL,
    left_side = FALSE
)

## S3 method for class 'sc'
weight_streetnet(
    x,
    wt_profile = "bicycle",
    wt_profile_file = NULL,
    turn_penalty = FALSE,
    type_col = "highway",
    id_col = "osm_id",
    keep_cols = NULL,
    left_side = FALSE
)

Arguments

x A street network represented either as sf LINESTRING objects, typically extracted with dodgr_streetnet, or as an SC (silicate) object typically extracted with the dodgr_streetnet_sc.

wt_profile Name of weighting profile, or data.frame specifying custom values (see Details)

wt_profile_file Name of locally-stored, .json-formatted version of dodgr::weighting_profiles, created with write_dodgr_wt_profile, and modified as desired.

turn_penalty Including time penalty on edges for turning across oncoming traffic at intersections (see Note).

type_col Specify column of the sf data.frame object which designates different types of highways to be used for weighting (default works with osmdata objects).

id_col For sf-formatted data only: Specify column of the codesf data.frame object
weight_streetnet

which provides unique identifiers for each highway (default works with osmdata objects).

keep_cols  Vectors of columns from x to be kept in the resultant dodgr network; vector can be either names or indices of desired columns.

left_side  Does traffic travel on the left side of the road (TRUE) or the right side (FALSE)? - only has effect on turn angle calculations for edge times.

Value

A data.frame of edges representing the street network, with distances in metres and times in seconds, along with a column of graph component numbers. Times for sf-formatted street networks are only approximate, and do not take into account traffic lights, turn angles, or elevation changes. Times for sc-formatted street networks take into account all of these factors, with elevation changes automatically taken into account for networks generated with the osmdata function osm_elevation().

Note

Names for the wt_profile parameter are taken from weighting_profiles, which is a list including a data.frame also called weighting_profiles of weights for different modes of transport. Values for wt_profile are taken from current modes included there, which are "bicycle", "foot", "goods", "hgv", "horse", "moped", "motorcar", "motorcycle", "psv", and "wheelchair". Railway routing can be implemented with the separate function weight_railway. Alternatively, the entire weighting_profile structures can be written to a local .json-formatted file with write_dodgr_wt_profile, the values edited as desired, and the name of this file passed as the wt_profile_file parameter. Construction of custom weighting profiles is illustrated in the following example.

Calculating edge times to account for turn angles (that is, with turn_penalty = TRUE) involves calculating the temporal delay involving in turning across oncoming traffic. Resultant graphs are fundamentally different from the default for distance-based routing. The result of weight_streetnet(..., turn_penalty = TRUE) should thus only be used to submit to the dodgr_times function, and not for any other dodgr functions nor forms of network analysis.

The resultant graph includes only those edges for which the given weighting profile specifies finite edge weights. Any edges of types not present in a given weighting profile are automatically removed from the weighted streetnet.

If the resultant graph is to be contracted via dodgr_contract_graph, and if the columns of the graph have been, or will be, modified, then automatic caching must be switched off with dodgr_cache_off. If not, the dodgr_contract_graph function will return the automatically cached version, which is the contracted version of the full graph prior to any modification of columns.

See Also

write_dodgr_wt_profile, dodgr_times

Examples

# hampi is included with package as an 'osmdata' sf-formatted street network
net <- weight_streetnet(hampi)
class(net) # data.frame
dim(net) # 6096 11; 6096 streets
# os_roads_bristol is also included as an sf data.frame, but in a different
# format requiring identification of columns and specification of custom
# weighting scheme.
colnm <- "formOfWay"
wts <- data.frame(name = "custom",
                   way = unique(os_roads_bristol[[colnm]]),
                   value = c(0.1, 0.2, 0.8, 1))
net <- weight_streetnet(os_roads_bristol, wt_profile = wts,
                        type_col = colnm, id_col = "identifier")
dim(net) # 406 11; 406 streets

# An example for a generic (non-OSM) highway, represented as the
# 'routes_fast' object of the \pkg{stplanr} package, which is a
# \code{SpatialLinesDataFrame}.
## Not run:
library(stplanr)
# merge all of the 'routes_fast' lines into a single network
r <- overline(routes_fast, attrib = "length", buff_dist = 1)
r <- sf::st_as_sf(r, crs = 4326)
# We need to specify both a 'type' and 'id' column for the
# \link{weight_streetnet} function.
rt$type <- 1
rt$id <- seq(nrow(r))
graph <- weight_streetnet(r, type_col = "type", id_col = "id",
                          wt_profile = 1)
## End(Not run)

---

**write_dodgr_wt_profile**

**Description**

Write the dodgr street network weighting profiles to a local .json-formatted file for manual editing and subsequent re-reading.

**Usage**

```r
write_dodgr_wt_profile(file = NULL)
```

**Arguments**

- `file` Full name (including path) of file to which to write. The .json suffix will be automatically appended.

**Value**

TRUE if writing successful.
write_dodgr_wt_profile

See Also

weight_streetnet
Index

*Topic datasets
  hampi, 38
  os_roads_bristol, 43
  weighting_profiles, 44

clear_dodgr_cache, 3
compare_heaps, 3
compare_heaps(), 4
dodgr, 4
dodgr_cache_off, 5, 5, 47
dodgr_cache_on, 5, 5
dodgr_centrality, 6, 37
dodgr_components, 8
dodgr_components(), 4
dodgr_contract_graph, 8, 22, 31, 35, 47
dodgr_contract_graph(), 4
dodgr_distances, 9
dodgr_dist, 9, 10, 40, 41
dodgr_dist(), 4
dodgr_flowmap, 13
dodgr_flows_aggregate, 13, 14, 40–42
dodgr_flows_disperse, 13, 16, 42
dodgr_flows_si, 18
dodgr_full_cycles, 20, 22
dodgr_fundamental_cycles, 21
dodgr_insert_vertex, 22
dodgr_isochrones, 23
dodgr_isodists, 24
dodgr_isoverts, 25
dodgr_paths, 26, 40, 41
dodgr_sample, 27
dodgr_sample(), 4
dodgr_sflines_to_poly, 28
dodgr_streetnet, 29, 46
dodgr_streetnet(), 4
dodgr_streetnet_sc, 30, 46
dodgr_times, 31, 47
dodgr_to_igraph, 32, 39
dodgr_to_sf, 33, 34
dodgr_to_sfc, 34
dodgr_to_tidygraph, 35
dodgr_uncontract_graph, 35
dodgr_vertices, 21, 36
dodgr_vertices(), 4
estimate_centrality_threshold, 6, 37, 37,
38
estimate_centrality_time, 37
hampi, 38
igraph_to_dodgr, 33, 39
match_points_to_graph, 40, 40
match_pts_to_graph, 41
merge_directed_graph, 42
os_roads_bristol, 43
weight_railway, 44, 47
weight_streetnet, 11, 12, 31, 45, 49
weight_streetnet(), 4
weighting_profiles, 44, 47
write_dodgr_wt_profile, 46, 47, 48