Package: cppRouting (via r-universe)

November 12, 2026

Type Package

Title Algorithms for Routing and Solving the Traffic Assignment Problem

Version 4.1

Date 2026-11-12

Description Calculation of distances, shortest paths and isochrones on weighted graphs using several variants of Dijkstra algorithm. Proposed algorithms are unidirectional Dijkstra (Dijkstra, E. W. (1959) <doi:10.1007/BF01386390>), bidirectional Dijkstra (Goldberg, Andrew & Fonseca F. Werneck, Renato (2005) <https://www.cs.princeton.edu/courses/archive/spr06/cos423/Handouts/EPP%20shortest%20path%20algorithms.pdf>), A* search (P. E. Hart, N. J. Nilsson et B. Raphael (1968) <doi:10.1109/TSSC.1968.300136>), new bidirectional A* (Pijls & Post (2009) <https://repub.eur.nl/pub/16100/ei2009-10.pdf>), Contraction hierarchies (R. Geisberger, P. Sanders, D. Schultes and D. Delling (2008) <doi:10.1007/978-3-540-68552-4_24>), PHAST (D. Delling, A.Goldberg, A. Nowatzyk, R. Werneck (2011) <doi:10.1016/j.jpdc.2012.02.007>). Algorithms for solving the traffic assignment problem are All-or-Nothing assignment, Method of Successive Averages, Frank-Wolfe algorithm (M. Fukushima (1984) <doi:10.1016/0191-2615(84)90029-8>), Conjugate and Bi-Conjugate Frank-Wolfe algorithms (M. Mitradjieva, P. O. Lindberg (2012) <doi:10.1287/trsc.1120.0409>), Algorithm-B (R. B. Dial (2006) <doi:10.1016/j.trb.2006.02.008>), Customizable Contraction Hierarchies (J. Dibbelt, B. Strasser, D. Wagner (2014) <doi:10.1145/2886843>).

License GPL (>= 2)

Encoding UTF-8

Imports Rcpp (>= 1.0.7), RcppParallel, data.table

LinkingTo Rcpp, RcppParallel, RcppProgress

SystemRequirements GNU make, zlib

URL https://github.com/vlarmet/cppRouting

Suggests knitr, rmarkdown, igraph, sf, osmextract, testthat (>= 3.0.0)

VignetteBuilder knitr

NeedsCompilation yes

Config/roxygen2/version 8.0.0

Config/testthat/edition 3

Author Vincent Larmet [aut, cre]

Maintainer Vincent Larmet <larmet.vincent@gmail.com>

Config/pak/sysreqs make zlib1g-dev

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

Date/Publication 2026-11-12 14:10:02 UTC

RemoteUrl https://github.com/cran/cppRouting

RemoteRef HEAD

RemoteSha 4c94b1008e723e9c1c4d84260f9839adbbc29b1a

Contents

assign_traffic . . . 3
cpp_contract . . . 6
cpp_customize . . . 8
cpp_extract . . . 9
cpp_simplify . . . 13
get_aon . . . 14
get_detour . . . 16
get_distance_matrix . . . 18
get_distance_pair . . . 20
get_isochrone . . . 22
get_multi_paths . . . 23
get_path_pair . . . 24
makegraph . . . 26
osm_bike_profile . . . 28
osm_car_profile . . . 28
osm_foot_profile . . . 29
osm_hgv_profile . . . 29
to_df . . . 30
to_igraph . . . 31

Index **32**

assign_traffic *Algorithms for solving the Traffic Assignment Problem (TAP).*

Description

Estimation of the User Equilibrium (UE)

Usage

```
assign_traffic(
  Graph,
  from,
  to,
  demand,
  algorithm = "bfw",
  max_gap = 0.001,
  max_it = .Machine$integer.max,
  aon_method = "bi",
  cch = NULL,
  constant = 1,
  dial_params = NULL,
  verbose = TRUE
)
```

Arguments

Graph	An object generated by makegraph function.
from	A vector of origins
to	A vector of destinations.
demand	A vector describing the flow between each origin-destination pair.
algorithm	character. msa, fw, cfw, bfw or dial. Default to bfw. See details.
max_gap	Numeric. Relative gap to achieve. Default to 0.001.
max_it	Numeric. Maximum number of iterations. Default to .Machine$integer.max
aon_method	Character.d, bi, nba, cphast, cbi or cch. Default to bi. See details.
cch	Optional CCH topology generated by cpp_contract(Graph, customizable = TRUE). Use it with aon_method = "cch" to prepare the road topology once and reuse it across many congestion runs.
constant	numeric. Constant to maintain the heuristic function admissible in NBA* algorithm. Default to 1, when cost is expressed in the same unit than coordinates. See details
dial_params	List. Named list of hyperparameters for dial algorithm. See details.
verbose	Logical. If TRUE (default), progression is displayed.

Details

The most well-known assumptions in traffic assignment models are the ones following Wardrop's first principle. Traffic assignment models are used to estimate the traffic flows on a network. These models take as input a matrix of flows that indicate the volume of traffic between origin and destination (O-D) pairs. Unlike All-or-Nothing assignment (see get_aon), edge congestion is modeled through the **Volume Decay Function (VDF)**. The Volume Decay Function used is the most popular in literature, from the Bureau of Public Roads :

t = t0 * (1 + a * (V/C)^b) with t = actual travel time (minutes), t0 = free-flow travel time (minutes), a = alpha parameter (unitless), b = beta parameter (unitless), V = volume or flow (veh/hour) C = edge capacity (veh/hour)

Traffic Assignment Problem is a convex problem and solving algorithms can be divided into two categories :

- link-based : **Method of Successive Average** (`msa`) and **Frank-Wolfe variants** (normal : `fw`, conjugate : `cfw` and bi-conjugate : `bfw`). These algorithms uses the descent direction given by AON assignment at each iteration, all links are updated at the same time.
- bush-based : **Algorithm-B** (`dial`) The problem is decomposed into sub-problems, corresponding to each origin of the OD matrix, that operate on acyclic sub-networks of the original transportation network, called bushes. Link flows are shifted from the longest path to the shortest path recursively within each bush using Newton method.

Link-based algorithms are historically the first algorithms developed for solving the traffic assignment problem. It require low memory and are known to tail in the vicinity of the optimum and usually cannot be used to achieve highly precise solutions. Algorithm B is more recent, and is better suited for achieve the highest precise solution. However, it require more memory and can be time-consuming according the network size and OD matrix size. In `cppRouting`, the implementation of algorithm-B allow "batching", i.e. bushes are temporarily stored on disk if memory limit, defined by the user, is exceeded. Please see the package website for practical example and deeper explanations about algorithms. (https://github.com/vlarmet/cppRouting/blob/master/README.md)

Convergence criterion can be set by the user using max_gap argument, it is the relative gap which can be written as : **abs(TSTT/SPTT - 1)** with TSTT (Total System Travel Time) = sum(flow * cost), SPTT (Shortest Path Travel Time) = sum(aon * cost)

Especially for link-based algorithms (msa, *fw), the larger part of computation time rely on AON assignment. So, choosing the right AON algorithm is crucial for fast execution time. Contracting the network on-the-fly before AON computing can be faster for large network and/or large OD matrix.

AON algorithms are :

- `bi` : bidirectional Dijkstra algorithm
- `nba` : bidirectional A* algorithm, nodes coordinates and constant parameter are needed
- `d` : Dijkstra algorithm
- `cbi` : contraction hierarchies + bidirectional search
- `cphast` : contraction hierarchies + phast algorithm
- `cch` : customizable contraction hierarchies with RoutingKit-style elimination-tree grouped queries. The topology is prepared once, then edge costs are customized at each assignment iteration.

These AON algorithm can be decomposed into two families, depending the sparsity of origin-destination matrix :

- recursive pairwise : `bi`, `nba` and `cbi`. Optimal for high sparsity. One-to-one algorithm is called N times, with N being the length of from.
- recursive one-to-many : `d`, `cphast` and `cch`. Optimal for dense or repeated-endpoint matrices. One-to-many algorithm is called N times, with N being the number of unique from (or to) nodes.

CCH is most useful when the same graph topology is reused with changing edge costs, for example during congestion assignment. In that case, run `cch <- cpp_contract(graph, customizable = TRUE)` once, save it if needed, and pass `cch = cch` with `aon_method = "cch"`. The assignment will then customize the CCH at each iteration instead of rebuilding a full contraction hierarchy.

For large instance, it may be appropriate to test different `aon_method` for few iterations and choose the fastest one for the final estimation.

Hyperparameters for algorithm-b are :

- `inneriter` : number of time bushes are equilibrated within each iteration. Default to 20
- `max_tol` : numerical tolerance. Flow is set to 0 if less than max_tol. Since flow shifting consist of iteratively adding or substracting double types, numerical error can occur and stop convergence. Default to 1e-11.
- `tmp_path` : Path for storing bushes during algorithm-B execution. Default using `tempdir()`
- `max_mem` : Maximum amount of RAM used by algorithm-B in gigabytes. Default to 8.

In New Bidirectional A star algorithm, euclidean distance is used as heuristic function. To understand the importance of constant parameter, see the package description : https://github.com/vlarmet/cppRouting/blob/master/README.md All algorithms are partly multithreaded (AON assignment).

Value

A `list` containing :

- The relative gap achieved
- Number of iteration
- A data.frame containing edges attributes, including equilibrated flows, new costs and free-flow travel times.

Note

`from`, `to` and `demand` must be the same length. `alpha`, `beta` and `capacity` must be filled in during network construction. See makegraph.

References

Wardrop, J. G. (1952). "Some Theoretical Aspects of Road Traffic Research".

M. Fukushima (1984). "A modified Frank-Wolfe algorithm for solving the traffic assignment problem".

R. B. Dial (2006). "A path-based user-equilibrium traffic assignment algorithm that obviates path storage and enumeration".

M. Mitradjieva, P. O. Lindberg (2012). "The Stiff Is Moving — Conjugate Direction Frank-Wolfe Methods with Applications to Traffic Assignment".

Examples

```
#Choose number of cores used by cppRouting
RcppParallel::setThreadOptions(numThreads = 1)

#Data describing edges of the graph
edges<-data.frame(from_vertex=c(0,0,1,1,2,2,3,4,4),
                  to_vertex=c(1,3,2,4,4,5,1,3,5),
                  cost=c(9,2,11,3,5,12,4,1,6))

# Origin-destination trips
trips <- data.frame(from = c(0,0,0,0,1,1,1,1,2,2,2,3,3,4,5,5,5,5,5),
                    to = c(1,2,5,3,2,5,2,4,2,5,2,3,5,2,0,0,3,5,1),
                    flow = c(10,30,15,5,5,2,3,6,4,15,20,2,3,6,2,1,4,5,3))

#Construct graph
graph <- makegraph(edges,directed=TRUE, alpha = 0.15, beta = 4, capacity = 5)


# Solve traffic assignment problem
## using Bi-conjugate Frank-Wolfe algorithm
traffic <- assign_traffic(Graph=graph,
                          from=trips$from, to=trips$to, demand = trips$flow,
                          algorithm = "bfw")
print(traffic$data)

## using algorithm-B
traffic2 <- assign_traffic(Graph=graph,
                           from=trips$from, to=trips$to, demand = trips$flow,
                           algorithm = "dial")
print(traffic2$data)
```

`cpp_contract` *Contraction hierarchies algorithm*

Description

Contract a graph by using contraction hierarchies algorithm.

Usage

```
cpp_contract(Graph, silent = FALSE, customizable = FALSE, order = NULL)
```

Arguments

`Graph`	An object generated by makegraph or cpp_simplify function.
`silent`	Logical. If `TRUE`, progress is not displayed.
`customizable`	Logical. If `TRUE`, build a customizable contraction hierarchy (CCH) instead of a classical one: only the graph topology is contracted, so shortcut costs can be recomputed cheaply with cpp_customize whenever edge weights change. See Details.
`order`	Optional node order, only used when `customizable = TRUE`. Values must be node references from `Graph$dict$ref`. If omitted, cppRouting computes a simple local order. Large road networks usually benefit from a stronger nested-dissection order prepared outside this function.

Details

Contraction hierarchies is a speed-up technique for finding shortest path in a graph. It consist of two steps : preprocessing phase and query. `cpp_contract()` preprocess the input graph to later use special query algorithm implemented in get_distance_pair, get_distance_matrix, get_aon and get_path_pair functions.

When `customizable = TRUE`, contraction is split into two phases instead:

- **Preparation** (this function): builds shortcuts from the road topology. This step is reusable as long as the set of edges does not change.
- **Customization** (cpp_customize): updates shortcut costs from the current edge weights. This step is repeated when congestion changes travel times.

For traffic assignment, prepare once and call `assign_traffic(..., aon_method = "cch", cch = cch)`. The CCH assignment path uses the elimination-tree query, which is the fast query variant for repeated origins or repeated destinations.

To see the benefits of using contraction hierarchies, see the package description : https://github.com/vlarmet/cppRouting/blob/master/README.md.

Value

A contracted graph, or (when `customizable = TRUE`) a CCH topology object. Run cpp_customize before distance queries on a CCH topology object, or pass it to assign_traffic through the `cch` argument to avoid rebuilding the topology at each model run.

See Also

cpp_simplify

Examples

```
#Data describing edges of the graph
edges<-data.frame(from_vertex=c(0,0,1,1,2,2,3,4,4),
                  to_vertex=c(1,3,2,4,4,5,1,3,5),
                  cost=c(9,2,11,3,5,12,4,1,6))
```

```
#Construct cppRouting graph
graph<-makegraph(edges,directed=TRUE)

#Contract graph
contracted_graph<-cpp_contract(graph,silent=TRUE)

#Build a customizable contraction hierarchy instead
cch <- cpp_contract(graph, customizable = TRUE)
metric <- cpp_customize(cch)
get_distance_pair(metric, from = 0, to = 3)
```

`cpp_customize` *Customize a CCH with edge weights*

Description

Update a prepared CCH with the edge weights that should be used for shortest path queries. Customization is much cheaper than preparing the CCH topology, so it is the step to repeat when congestion changes travel times.

Usage

```
cpp_customize(CCH, weights = NULL)
```

Arguments

`CCH`	An object generated by cpp_contract with `customizable = TRUE`.
`weights`	Optional edge weights. Defaults to the original graph cost. Give a single column name (`"dist"` or a column imported with `makegraph(aux = ...)`) to select a stored weight, or one numeric value per edge of `CCH$original$data`.

Details

The recommended way to customize a CCH is to import the weights up front with `makegraph` using the `aux` argument and select them by column name, e.g. `cpp_customize(cch, weights = "time")`. This is safer than writing a raw vector at query time: the edge table stored in the CCH doubles when the graph was built with `directed = FALSE`, so a raw vector is easy to get the wrong length or order for, and a mismatch is silently accepted. A raw numeric vector must then hold one value per edge of `CCH$original$data`, in that exact order.

Value

A customized CCH metric object usable by get_distance_pair, get_distance_matrix and get_aon.

Examples

```
RcppParallel::setThreadOptions(numThreads = 1)

# Travel time as the main cost, road length as an auxiliary weight
edges <- data.frame(from_vertex = c(0, 0, 1, 1, 2),
                    to_vertex = c(1, 2, 2, 3, 3),
                    cost = c(1, 4, 1, 2, 1))
graph <- makegraph(edges, directed = TRUE,
                   aux = data.frame(length = c(1, 3, 2, 4, 2)))
cch <- cpp_contract(graph, customizable = TRUE)

# First customization with the travel time (the default)
metric_time <- cpp_customize(cch)
get_distance_pair(metric_time, from = c(0, 1), to = c(3, 3))

# Second customization with the road length, selected by column name
metric_length <- cpp_customize(cch, weights = "length")
get_distance_pair(metric_length, from = c(0, 1), to = c(3, 3))
```

`cpp_extract` *Extract a routable graph from a geographic source*

Description

Single entry point to build a makegraph-ready graph from a geographic linestring source. The source is dispatched on automatically from `x`'s type (file extension, or object class). Exactly five forms of `x` are recognized – see the `x` parameter below for the precise contract of each, since passing the wrong shape of object for a given source (e.g. only part of an `osmdata` result) fails loudly rather than silently producing a wrong graph.

Usage

```
cpp_extract(
  x,
  output = c("graph", "data.frame", "csv"),
  out_prefix = NULL,
  coords = TRUE,
  epsg = NULL,
  keep_length = FALSE,
  snap_digits = NULL,
  osm_params = list(),
  geom_params = list()
)
```

Arguments

`x`	One of:

- a path (character) to a `.shp` shapefile – the companion `.dbf`/`.prj` must sit next to it, same base name;
- an in-memory `sf` object (`LINESTRING`/`MULTILINESTRING`), e.g. from `sf::st_read()` on a GeoPackage or any other OGR-readable format;
- a path (character) to an `.osm.pbf` file;
- the **full result of** `osmdata::osmdata_sf()` (class `"osmdata"`) – not just its `$osm_lines` element. `$osm_points` must also be present: it holds the `osm_id` of every node osmdata downloaded (including untagged vertices in the middle of a way), which is what lets `cpp_extract` recover real OSM topology and detect junctions;
- for `osmextract`: not a distinct input type – get the path to the **raw cached** `.pbf` with `osmextract::oe_get(..., download_only = TRUE, skip_vectortranslate = TRUE)` (**both** flags are required: `download_only` alone still runs the GDAL conversion and returns a `.gpkg` path instead) and pass that `.pbf` path like any other `.pbf` file. Do not let osmextract convert it to `sf`/GeoPackage via GDAL first: that conversion only keeps tagged nodes by default, which is not enough to reconstruct topology.

GeoPackage is not supported and has no plans to be: convert it to a shapefile first (e.g. via `sf::st_write(sf::st_read("file.gpkg"), "file.shp")`, or with any GIS tool) and use that instead.

`output` One of `"graph"` (default, returns a ready-to-use `cppRouting_graph`), `"data.frame"`, or `"csv"`. Common to every source.

`out_prefix` Path prefix used when `output = "csv"`. Common to every source.

`coords` logical. If `TRUE` (default), extract each node's coordinates: written to `<out_prefix>_coords.csv` (`output = "csv"`, including the `.pbf` `max_mem` disk-backed pipeline), returned as a `data.frame(node_id, X, Y)` (`output = "data.frame"`), or passed to `makegraph(coords = ...)` (`output = "graph"`). Set to `FALSE` to skip this – e.g. if you already have node coordinates elsewhere and only need the graph topology. For the `max_mem` pipeline this also skips an entire external sort/merge pass, a real saving in disk I/O and time at continent scale, not just a smaller output. If the source coordinates are geographic (lat/lon – always true for `.pbf`/`osmdata`) and `coords = TRUE`, `epsg` must be supplied (see below), otherwise this errors rather than silently returning coordinates that break the A*/NBA algorithms (see get_distance_pair). Common to every source.

`epsg` Optional. A projected EPSG code (or anything `sf::st_crs()` accepts) to reproject node coordinates into, via `sf::st_transform()` (requires the `sf` package). Only takes effect if `coords = TRUE` and the source CRS is geographic; already-projected coordinates (possible for shapefile/sf sources) are left as-is regardless. Together with `osm_params$max_mem` on a `.pbf` source, the reprojection runs in row chunks sized from the `max_mem` budget, so peak RAM stays within that budget even on the disk-backed pipeline. Common to every source.

`keep_length` logical. If `TRUE`, keep each edge's length (in km) as `aux`, regardless of how the main `cost` was actually resolved – e.g. to get a time-based graph while still reporting physical distance. Shapefile/sf sources: the geometric length (unreliable for a geographic CRS, same caveat as `geom_params$length_field` below – reproject or use `geom_params$keep_fields` with your own length

column instead in that case). OSM sources: the haversine length. `output = "csv"` carries the same information as `output = "data.frame"`: `aux` is written to `<out_prefix>_aux.csv`. Common to every source.

`snap_digits` integer. Shapefile/sf sources: decimal places used to round coordinates before deduplicating line endpoints into nodes. `osmdata` sources: decimal places used to match line vertex coordinates against `$osm_points` coordinates. Unused for `.pbf` (topology comes from OSM node ids directly, no coordinate matching needed). Defaults (`NULL`) to 7 for geographic (lat/lon) coordinates or 4 for projected (meter-scale) ones – geographic coordinates need many decimal places to distinguish nearby points, but that precision is far too tight for projected coordinates, where it starts picking up floating-point noise between independently-digitized but logically-identical points rather than real geometric differences. The one argument that doesn't sort cleanly into `osm_params`/`geom_params`: it applies to shapefile/sf sources and to `osmdata`, but not to `.pbf`.

`osm_params` A list of options for OSM sources (`.pbf`/`osmdata`/`osmextract`) only, ignored for shapefile/`sf`. Any subset may be supplied – omitted elements keep their default below; an unrecognized element name raises an error immediately (rather than silently keeping its default, which is what a typo'd list element would otherwise do). Recognized elements:

- `mode`: one of `"car"` (default), `"foot"`, `"bicycle"` or `"hgv"` – which access-tag/oneway resolution rules to apply (see `graph_from_osmpbf`'s `@details` for the exact tags checked per mode). If `profile` is left `NULL`, also selects the matching default profile (osm_car_profile/osm_foot_profile/osm_bike_profile/osm_hgv_profile).
- `profile`: a data.frame with columns `highway`, `clazz`, `speed` (see `osm_car_profile` and friends): which `highway` tag values to keep, and their class id / default speed (km/h) when not overridden by `maxspeed`. Defaults (`NULL`) to the profile matching `mode`; pass your own to override independently of `mode`.
- `use_maxspeed`: logical, default `TRUE`. If `TRUE`, a way's `maxspeed` tag overrides the profile's default speed when present and parseable (plain km/h, or `"<n> mph"`). No effect for `mode = "foot"`/`"bicycle"`.
- `assume_oneway_motorway`: logical, default `TRUE`. If `TRUE`, `motorway`/`motorway_link` ways are treated as one-way unless explicitly tagged `oneway=no`. Irrelevant for `mode = "foot"` (oneway is always ignored for that mode).
- `keep_clazz`: logical, default `FALSE`. If `TRUE`, keep each edge's resolved highway class id as `aux`.
- `max_mem`: `.pbf` only (ignored for `osmdata`). A budget in megabytes for a disk-backed pipeline that spills every intermediate table (not just the final result) to disk, keeping peak RAM close to this budget regardless of the input file's size – for files too large to fit in RAM even with the default pipeline (e.g. a full-planet extract). Requires `output = "csv"`. See `graph_from_osmpbf` for details and the (rare but possible) requirement that nodes appear in increasing id order in the file.
- `tmp_path`: `.pbf` only. Directory for temporary files when `max_mem` is set. Defaults to `tempdir()`.
- `progress`: logical, default `TRUE`. Print progress messages while reading/matching.

`geom_params` A list of options for shapefile/`sf` sources only, ignored for OSM sources. Any subset may be supplied – omitted elements keep their default below; an unrecognized element name raises an error immediately (same reasoning as `osm_params` above). Recognized elements:

- `cost_field`: optional, name of a numeric attribute column to use directly as edge cost.
- `speed_field`: optional, name of a numeric attribute column holding a speed in km/h. Used with `length_field` (in km) to derive a cost in minutes as `60 * length / speed`. Ignored if `cost_field` is provided.
- `length_field`: optional, name of a numeric attribute column holding each line's length, in km (to match the `60 * length / speed` formula above). If `NULL` and `speed_field` is used, length is instead computed from the line geometry itself and converted from the CRS's native unit (assumed to be meters, true for any typical projected CRS) to km.
- `direction_field`: optional, name of an attribute column describing the direction of travel allowed on each line (e.g. a "sens" column). Combined with `direction_values`.
- `direction_values`: optional, a list with up to three character vectors: `forward` (only from the line's first point to its last), `backward` (only from last to first) and `twoway` (both directions). Values of `direction_field` not listed anywhere default to `twoway`, with a warning.
- `directed`: logical, default `FALSE`. Used only when `direction_field` is `NULL`. If `TRUE`, every line is kept as digitized (first point to last point); if `FALSE`, every line is made bidirectional.
- `keep_fields`: optional character vector of extra attribute columns to keep as `aux` (see makegraph).

Details

Arguments are grouped by which family of source they apply to: arguments that work the same way regardless of source stay as their own top-level parameter; source-specific ones are grouped into the `osm_params` or `geom_params` list (see those two parameters below for the full list of what each one accepts) – so a shapefile/`sf` call never needs to mention `profile`/ `use_maxspeed`, and a `.pbf`/`osmdata`/`osmextract` call never needs to mention `cost_field`/`direction_field`.

Shapefile/sf sources are assumed to already be topologically clean, i.e. lines only meet at their endpoints. OSM sources (`.pbf` or `osmdata`) instead detect junctions in the middle of a way from OSM's own node ids (a node referenced by 2 or more kept ways, or a way's own endpoint); for `.pbf`, a way referencing a node outside the file (a regional extract cut through it) is dropped entirely, and for `osmdata`, a way with a vertex that can't be matched against `$osm_points` is dropped, with a warning. A plain `sf` object that looks OSM-derived (it has an `osm_id` column) but isn't the full `osmdata` result is refused with an explanatory error, rather than silently treated as topologically clean like a plain `sf` source and producing a wrong graph.

Value

Depending on `output`: a `cppRouting_graph`, a named list of two data.frames (`data`, `coords`), or (invisibly) the paths of the written CSV files. For `output = "data.frame"`, `from`/`to`/`node_id` are

`character` regardless of source: a node id is a label, not a quantity, matching what `makegraph()` itself already guarantees for `dict$ref`. For `output = "graph"`, that same real id is carried through `dict$ref`; `data$from`/`data$to` stay `makegraph()`'s own compact integer index (its adjacency list is always built on that, not on the original id).

`cpp_simplify` *Reduce the number of edges by removing non-intersection nodes, duplicated edges and isolated loops in the graph.*

Description

Reduce the number of edges by removing non-intersection nodes, duplicated edges and isolated loops in the graph.

Usage

```
cpp_simplify(
  Graph,
  keep = NULL,
  rm_loop = TRUE,
  iterate = FALSE,
  silent = TRUE
)
```

Arguments

`Graph`	An object generated by makegraph function.
`keep`	Character or integer vector. Nodes of interest that will not be removed. Default to `NULL`
`rm_loop`	Logical. if `TRUE`, isolated loops as removed. Default to `TRUE`
`iterate`	Logical. If `TRUE`, process is repeated until only intersection nodes remain in the graph. Default to FALSE
`silent`	Logical. If `TRUE` and iterate set to `TRUE`, number of iteration and number of removed nodes are printed to the console.

Details

To understand why process can be iterated, see the package description : https://github.com/vlarmet/cppRouting/blob/master/README.md

Value

The simplified cppRouting graph

Note

aux is preserved through simplification: when a chain of edges is collapsed into one, its aux value(s) are summed over the collapsed edges, the same way aggregate_aux sums them over a shortest path. alpha, beta and capacity are still removed. The first iteration usually eliminates the majority of non-intersection nodes and is therefore faster.

Examples

```
#Simple directed graph
edges<-data.frame(from=c(1,2,3,4,5,6,7,8),
                  to=c(0,1,2,3,6,7,8,5),
                  dist=c(1,1,1,1,1,1,1,1))

#Plot
if(requireNamespace("igraph",quietly = TRUE)){
igr<-igraph::graph_from_data_frame(edges)
plot(igr)
}

#Construct cppRouting graph
graph<-makegraph(edges,directed=TRUE)

#Simplify the graph, removing loop
simp<-cpp_simplify(graph, rm_loop=TRUE)

#Convert cppRouting graph to data frame
simp<-to_df(simp)

#Plot
if(requireNamespace("igraph",quietly = TRUE)){
igr<-igraph::graph_from_data_frame(simp)
plot(igr)
}

#Simplify the graph, keeping node 2 and keeping loop
simp<-cpp_simplify(graph,keep=2 ,rm_loop=FALSE)

#Convert cppRouting graph to data frame
simp<-to_df(simp)

#Plot
if(requireNamespace("igraph",quietly = TRUE)){
igr<-igraph::graph_from_data_frame(simp)
plot(igr)
}
```

---

get_aon *Given an origin-destination matrix, compute All-or-Nothing assignment.*

Description

Given an origin-destination matrix, compute All-or-Nothing assignment.

Usage

```
get_aon(Graph, from, to, demand, algorithm = "bi", constant = 1)
```

Arguments

Graph	An object generated by makegraph, or cpp_contract function.
from	A vector of origins
to	A vector of destinations.
demand	A vector describing the flow between each origin-destination pair.
algorithm	character. For contracted network : phast or bi. Otherwise : d, bi or nba. Default to bi. See details.
constant	numeric. Constant to maintain the heuristic function admissible in NBA* algorithm. Default to 1, when cost is expressed in the same unit than coordinates. See details

Details

All-or-Nothing assignment (AON) is the simplest method to load flow on a network, since it assume there is no congestion effects. The assignment algorithm itself is the procedure that loads the origin-destination matrix to the shortest path trees and produces the flows. Origin-destination matrix is represented via 3 vectors : from, to and demand.

There is two variants of algorithms, depending the **sparsity** of origin-destination matrix :

- recursive one-to-one : Bidirectional search (bi) and Bidirectional A* (nba). Optimal for high sparsity.
- recursive one-to-many : Dijkstra (d) and PHAST (phast). Optimal for dense matrix.

For large network and/or large OD matrix, this function is a lot faster on a contracted network. In New Bidirectional A star algorithm, euclidean distance is used as heuristic function. To understand the importance of constant parameter, see the package description : https://github.com/vlarmet/cppRouting/blob/master/README.md

All algorithms are **multithreaded.** Please use RcppParallel::setThreadOptions() to set the number of threads.

Value

A data.frame containing edges attributes, including flow.

Note

'from', 'to' and 'demand' must be the same length.

See Also

cpp_contract, assign_traffic

Examples

```
#Choose number of cores used by cppRouting
RcppParallel::setThreadOptions(numThreads = 1)

#Data describing edges of the graph
edges<-data.frame(from_vertex=c(0,0,1,1,2,2,3,4,4),
                  to_vertex=c(1,3,2,4,4,5,1,3,5),
                  cost=c(9,2,11,3,5,12,4,1,6))

# Origin-destination trips
trips <- data.frame(from = c(0,0,0,0,1,1,1,1,2,2,2,3,3,4,5,5,5,5,5),
                    to = c(1,2,5,3,2,5,2,4,2,5,2,3,5,2,0,0,3,5,1),
                    flow = c(10,30,15,5,5,2,3,6,4,15,20,2,3,6,2,1,4,5,3))

#Construct graph
graph<-makegraph(edges,directed=TRUE)


# Compute All-or-Nothing assignment
aon <- get_aon(Graph=graph, from=trips$from, to=trips$to, demand = trips$flow, algorithm = "d")
print(aon)
```

`get_detour` *Return the nodes that can be reached in a detour time set around the shortest path*

Description

Return the nodes that can be reached in a detour time set around the shortest path

Usage

```
get_detour(Graph, from, to, extra = NULL, keep = NULL, long = FALSE)
```

Arguments

`Graph`	An object generated by makegraph or cpp_simplify function.
`from`	A vector of one or more vertices from which shortest path are calculated (origin).
`to`	A vector of one or more vertices (destination).
`extra`	numeric. Additional cost
`keep`	numeric or character. Vertices of interest that will be returned.
`long`	logical. If `TRUE`, a long `data.frame` is returned instead of a list.

Details

Each returned nodes *n* meet the following condition :

SP(o,n) + SP(n,d) < SP(o,d) + t

with *SP* shortest distance/time, *o* the origin node, *d* the destination node and *t* the extra cost.

Modified bidirectional Dijkstra algorithm is ran for each path.

This algorithm is **multithreaded.** Please use `RcppParallel::setThreadOptions()` to set the number of threads.

Value

`list` or a `data.frame` of nodes that can be reached

Note

`from` and `to` must be the same length.

Examples

```
#Choose number of cores used by cppRouting
RcppParallel::setThreadOptions(numThreads = 1)

if(requireNamespace("igraph",quietly = TRUE)){

#Generate fully connected graph
gf<- igraph::make_full_graph(400)
igraph::V(gf)$names<-1:400

#Convert to data frame and add random weights
df<-igraph::as_long_data_frame(gf)
df$dist<-sample(1:100,nrow(df),replace = TRUE)

#Construct cppRouting graph
graph<-makegraph(df[,c(1,2,5)],directed = FALSE)

#Pick up random origin and destination node
origin<-sample(1:400,1)
destination<-sample(1:400,1)

#Compute distance from origin to all nodes
or_to_all<-get_distance_matrix(graph,from=origin,to=1:400)

#Compute distance from all nodes to destination
all_to_dest<-get_distance_matrix(graph,from=1:400,to=destination,)

#Get all shortest paths from origin to destination, passing by each node of the graph
total_paths<-rowSums(cbind(t(or_to_all),all_to_dest))

#Compute shortest path between origin and destination
distance<-get_distance_pair(graph,from=origin,to=destination)
```

```
#Compute detour with an additional cost of 3
det<-get_detour(graph,from=origin,to=destination,extra=3)

#Check result validity
length(unlist(det))
length(total_paths[total_paths < distance + 3])

}
```

`get_distance_matrix` *Compute all shortest distance between origin and destination nodes.*

Description

Compute all shortest distance between origin and destination nodes.

Usage

```
get_distance_matrix(
  Graph,
  from,
  to,
  algorithm = "phast",
  aggregate_aux = FALSE,
  allcores = FALSE
)
```

Arguments

`Graph`	An object generated by makegraph, cpp_simplify or cpp_contract function.
`from`	A vector of one or more vertices from which distances are calculated (origin).
`to`	A vector of one or more vertices (destination).
`algorithm`	Character. Only for contracted graph, `mch` for Many to many CH, `phast` for PHAST algorithm
`aggregate_aux`	Logical or character vector. If `TRUE`, the additional weight(s) declared in makegraph's `aux` are summed along shortest paths (all columns, if `aux` was a data.frame). A character vector of column names restricts this to specific columns.
`allcores`	Logical (deprecated). If `TRUE`, all cores are used.

Details

If graph is not contracted, `get_distance_matrix()` recursively perform Dijkstra algorithm for each `from` nodes. If graph is contracted, the user has the choice between :

- many to many contraction hierarchies (`mch`) : optimal for square matrix.
- PHAST (`phast`) : outperform mch on rectangular matrix

Shortest path is always computed according to the main edge weights, corresponding to the 3rd column of df argument in makegraph() function. If aggregate_aux argument is TRUE, the values returned are the sum of auxiliary weights along shortest paths.

When several shortest paths tie on the main edge weight, aggregate_aux is summed along whichever tied path the search happens to settle on. Normal, cpp_contract-contracted and CCH graphs can break such ties differently, so the aggregated value for a given from/to pair may occasionally differ between them, even though the main shortest distance always matches exactly. This is expected: it reflects a genuine choice between equally short paths, not an error.

All algorithms are **multithreaded.** allcores argument is deprecated, please use RcppParallel::setThreadOptions() to set the number of threads.

See details in package website : https://github.com/vlarmet/cppRouting/blob/master/README.md

Value

Matrix of shortest distances, or of aggregated aux values when aggregate_aux is used. Requesting more than one aux column returns a named list of matrices (one per requested value) instead of a single matrix.

See Also

get_distance_pair, get_multi_paths

Examples

```
#Choose number of cores used by cppRouting
RcppParallel::setThreadOptions(numThreads = 1)

#Data describing edges of the graph
edges <- data.frame(from_vertex = c(0,0,1,1,2,2,3,4,4),
                    to_vertex = c(1,3,2,4,4,5,1,3,5),
                    time = c(9,2,11,3,5,12,4,1,6),
                    dist = c(5,3,4,7,5,5,5,8,7))

#Construct directed graph with travel time as principal weight, and distance as secondary weight
graph <- makegraph(edges[,1:3], directed=TRUE, aux = edges$dist)

#Get all nodes IDs
nodes <- graph$dict$ref

# Get matrix of shortest times between all nodes : the result are in time unit
time_mat <- get_distance_matrix(graph, from = nodes, to = nodes)

# Get matrix of distance according shortest times : the result are in distance unit
dist_mat <- get_distance_matrix(graph, from = nodes, to = nodes, aggregate_aux = TRUE)

print(time_mat)
print(dist_mat)
```

`get_distance_pair` *Compute shortest distance between origin and destination nodes.*

Description

Compute shortest distance between origin and destination nodes.

Usage

```
get_distance_pair(
  Graph,
  from,
  to,
  aggregate_aux = FALSE,
  algorithm = "bi",
  constant = 1,
  allcores = FALSE
)
```

Arguments

`Graph`	An object generated by makegraph, cpp_simplify or cpp_contract function.
`from`	A vector of one or more vertices from which distances are calculated (origin).
`to`	A vector of one or more vertices (destination).
`aggregate_aux`	Logical or character vector. If `TRUE`, the additional weight(s) declared in makegraph's `aux` are summed along shortest paths (all columns, if `aux` was a data.frame). A character vector of column names restricts this to specific columns.
`algorithm`	character. `Dijkstra` for uni-directional Dijkstra, `bi` for bi-directional Dijkstra, `A*` for A star unidirectional search or `NBA` for New bi-directional A star .Default to `bi`
`constant`	numeric. Constant to maintain the heuristic function admissible in `A*` and `NBA` algorithms. Default to 1, when cost is expressed in the same unit than coordinates. See details
`allcores`	Logical (deprecated). If `TRUE`, all cores are used.

Details

If graph is not contracted, the user has the choice between :

- unidirectional Dijkstra (`Dijkstra`)
- A star (`A*`) : projected coordinates should be provided
- bidirectional Dijkstra (`bi`)
- New bi-directional A star (`NBA`) : projected coordinates should be provided

If the input graph has been contracted by cpp_contract function, the algorithm is a modified bidirectional search.

Shortest path is always computed according to the main edge weights, corresponding to the 3rd column of `df` argument in makegraph function. If `aggregate_aux` argument is `TRUE`, the values returned are the sum of auxiliary weights along shortest paths.

When several shortest paths tie on the main edge weight, `aggregate_aux` is summed along whichever tied path the search happens to settle on. Normal, cpp_contract-contracted and CCH graphs can break such ties differently, so the aggregated value for a given `from`/`to` pair may occasionally differ between them, even though the main shortest distance always matches exactly. This is expected: it reflects a genuine choice between equally short paths, not an error.

In A* and New Bidirectional A star algorithms, euclidean distance is used as heuristic function.

All algorithms are **multithreaded.** `allcores` argument is deprecated, please use `RcppParallel::setThreadOptions()` to set the number of threads.

To understand how A star algorithm work, see https://en.wikipedia.org/wiki/A*_search_algorithm. To understand the importance of constant parameter, see the package description : https://github.com/vlarmet/cppRouting/blob/master/README.md

Value

Vector of shortest distances, or of aggregated `aux` values when `aggregate_aux` is used. Requesting more than one `aux` column returns a data.frame with one column per requested value instead of a vector.

Note

`from` and `to` must be the same length.

See Also

get_distance_matrix, get_path_pair, cpp_contract

Examples

```
#Choose number of cores used by cppRouting
RcppParallel::setThreadOptions(numThreads = 1)

#Data describing edges of the graph
edges<-data.frame(from_vertex=c(0,0,1,1,2,2,3,4,4),
                  to_vertex=c(1,3,2,4,4,5,1,3,5),
                  cost=c(9,2,11,3,5,12,4,1,6),
                  dist = c(5,3,4,7,5,5,5,8,7))

#Construct directed  graph with travel time as principal weight, and distance as secondary weight
graph <- makegraph(edges[,1:3], directed=TRUE, aux = edges$dist)

#Get all nodes IDs
nodes <- graph$dict$ref

# Get shortest times between all nodes : the result are in time unit
```

```
time_mat <- get_distance_pair(graph, from = nodes, to = nodes)

# Get distance according shortest times : the result are in distance unit
dist_mat <- get_distance_pair(graph, from = nodes, to = nodes, aggregate_aux = TRUE)

print(time_mat)
print(dist_mat)
```

`get_isochrone` *Compute isochrones/isodistances from nodes.*

Description

Compute isochrones/isodistances from nodes.

Usage

```
get_isochrone(Graph, from, lim, setdif = FALSE, keep = NULL, long = FALSE)
```

Arguments

`Graph`	An object generated by makegraph or cpp_simplify function.
`from`	numeric or character. A vector of one or more vertices from which isochrones/isodistances are calculated.
`lim`	numeric. A vector of one or multiple breaks.
`setdif`	logical. If `TRUE` and `length(lim) > 1`, nodes that are reachable in a given break will not appear in a greater one.
`keep`	numeric or character. Vertices of interest that will be returned.
`long`	logical. If `TRUE`, a long `data.frame` is returned instead of a `list`.

Details

If `length(lim) > 1`, value is a `list` of `length(from)`, containing `lists` of `length(lim)`.

All algorithms are **multithreaded.** Please use `RcppParallel::setThreadOptions()` to set the number of threads.

For large graph, `keep` argument can be used for saving memory.

Value

`list` or a `data.frame` containing reachable nodes below cost limit(s).

Note

`get_isochrone()` recursively perform Dijkstra algorithm for each `from` nodes and stop when cost limit is reached.

Examples

```
#Choose number of cores used by cppRouting
RcppParallel::setThreadOptions(numThreads = 1)

#Data describing edges of the graph
edges<-data.frame(from_vertex=c(0,0,1,1,2,2,3,4,4),
                  to_vertex=c(1,3,2,4,4,5,1,3,5),
                  cost=c(9,2,11,3,5,12,4,1,6))

#Construct directed graph
directed_graph<-makegraph(edges,directed=TRUE)

#Get nodes reachable around node 4 with maximum distances of 1 and 2
iso<-get_isochrone(Graph=directed_graph,from = "4",lim=c(1,2))

#With setdif set to TRUE
iso2<-get_isochrone(Graph=directed_graph,from = "4",lim=c(1,2),setdif=TRUE)
print(iso)
print(iso2)
```

`get_multi_paths` *Compute all shortest paths between origin and destination nodes.*

Description

Compute all shortest paths between origin and destination nodes.

Usage

```
get_multi_paths(Graph, from, to, keep = NULL, long = FALSE)
```

Arguments

`Graph`	An object generated by makegraph or cpp_simplify function.
`from`	A vector of one or more vertices from which shortest paths are calculated (origin).
`to`	A vector of one or more vertices (destination).
`keep`	numeric or character. Vertices of interest that will be returned.
`long`	logical. If `TRUE`, a long data.frame is returned instead of a list.

Details

`get_multi_paths()` recursively perform Dijkstra algorithm for each 'from' nodes. It is the equivalent of get_distance_matrix, but it return the shortest path node sequence instead of the distance.

This algorithm is **multithreaded.** Please use `RcppParallel::setThreadOptions()` to set the number of threads.

Value

List or a data.frame containing shortest paths.

Note

Be aware that if 'from' and 'to' have consequent size, output will require much memory space.

See Also

get_path_pair, get_isochrone, get_detour

Examples

```
#Choose number of cores used by cppRouting
RcppParallel::setThreadOptions(numThreads = 1)

#Data describing edges of the graph
edges<-data.frame(from_vertex=c(0,0,1,1,2,2,3,4,4),
                  to_vertex=c(1,3,2,4,4,5,1,3,5),
                  cost=c(9,2,11,3,5,12,4,1,6))

#Get all nodes
nodes<-unique(c(edges$from_vertex,edges$to_vertex))

#Construct directed graph
directed_graph<-makegraph(edges,directed=TRUE)

#Get all shortest paths (node sequences) between all nodes
dir_paths<-get_multi_paths(Graph=directed_graph, from=nodes, to=nodes)
print(dir_paths)

#Get the same result in data.frame format
dir_paths_df<-get_multi_paths(Graph=directed_graph, from=nodes, to=nodes, long = TRUE)
print(dir_paths_df)
```

---

get_path_pair *Compute shortest path between origin and destination nodes.*

Description

Compute shortest path between origin and destination nodes.

Usage

```
get_path_pair(
  Graph,
  from,
  to,
  algorithm = "bi",
```

```
  constant = 1,
  keep = NULL,
  long = FALSE
)
```

Arguments

Graph	An object generated by makegraph, cpp_simplify or cpp_contract function.
from	A vector of one or more vertices from which shortest paths are calculated (origin).
to	A vector of one or more vertices (destination).
algorithm	character. Dijkstra for uni-directional Dijkstra, bi for bi-directional Dijkstra, A* for A star unidirectional search or NBA for New bi-directional A star .Default to bi
constant	numeric. Constant to maintain the heuristic function admissible in A* and NBA algorithms.
keep	numeric or character. Vertices of interest that will be returned.
long	logical. If TRUE, a long data.frame is returned instead of a list. Default to 1, when cost is expressed in the same unit than coordinates. See details

Details

If graph is not contracted, the user has the choice between :

- unidirectional Dijkstra (Dijkstra)
- A star (A*) : projected coordinates should be provided
- bidirectional Dijkstra (bi)
- New bi-directional A star (NBA) : projected coordinates should be provided

If the input graph has been contracted by cpp_contract function, the algorithm is a modified bidirectional search.

When several shortest paths tie on the main edge weight, the exact node sequence returned for a given from/to pair can differ between algorithms and graph representations (normal, contracted or CCH), since each may break the tie differently. All of these paths are equally optimal in terms of the main shortest distance.

In A* and NBA algorithms, euclidean distance is used as heuristic function.

All algorithms are **multithreaded.** Please use RcppParallel::setThreadOptions() to set the number of threads.

To understand the importance of constant parameter, see the package description : https://github.com/vlarmet/cppRouting/blob/master/README.md

Value

list or a data.frame containing shortest path nodes between from and to.

Note

`from` and `from` must be the same length.

See Also

get_multi_paths, get_isochrone, get_detour

Examples

```
#Choose number of cores used by cppRouting
RcppParallel::setThreadOptions(numThreads = 1)

#Data describing edges of the graph
edges<-data.frame(from_vertex=c(0,0,1,1,2,2,3,4,4),
                  to_vertex=c(1,3,2,4,4,5,1,3,5),
                  cost=c(9,2,11,3,5,12,4,1,6))

#Get all nodes
nodes<-unique(c(edges$from_vertex,edges$to_vertex))

#Construct directed and undirected graph
directed_graph<-makegraph(edges,directed=TRUE)
non_directed<-makegraph(edges,directed=FALSE)

#Sampling origin and destination nodes
origin<-sample(nodes,10,replace=TRUE)
destination<-sample(nodes,10,replace=TRUE)

#Get distance between origin and destination in the two graphs
dir_paths<-get_path_pair(Graph=directed_graph, from=origin, to=destination)
non_dir_paths<-get_path_pair(Graph=non_directed, from=origin, to=destination)
print(dir_paths)
print(non_dir_paths)
```

`makegraph` *Construct graph*

Description

Construct graph

Usage

```
makegraph(
  df,
  directed = TRUE,
  coords = NULL,
  aux = NULL,
  capacity = NULL,
```

```
  alpha = NULL,
  beta = NULL
)
```

Arguments

df	A data.frame or matrix containing 3 columns: from, to, cost. See details.
directed	logical. If FALSE, then all edges are duplicated by inverting 'from' and 'to' nodes.
coords	Optional. A data.frame or matrix containing all nodes coordinates. Columns order should be 'node_ID', 'X', 'Y'.
aux	Optional. A vector, a single value, or a data.frame with one or more columns, describing one or more additional edge weights.
capacity	Optional. A vector or a single value describing edge capacity. Used for traffic assignment.
alpha	Optional. A vector or a single value describing alpha parameter. Used for traffic assignment.
beta	Optional. A vector or a single value describing beta parameter. Used for traffic assignment.

Details

'from' and 'to' are character or numeric vector containing nodes IDs. 'cost' is a non-negative numeric vector describing the cost (e.g time, distance) between each 'from' and 'to' nodes. coords should not be angles (e.g latitude and longitude), but expressed in a projection system. aux is an additional weight describing each edge. Shortest paths are always computed using 'cost' but aux can be summed over shortest paths. capacity, alpha and beta are parameters used in the Volume Delay Function (VDF) to equilibrate traffic in the network. See assign_traffic. capacity, alpha, beta and aux must have a length equal to nrow(df). If a single value is provided, this value is replicated for each edge. alpha must be different from 0 and alpha must be greater or equal to 1. For more details and examples about traffic assignment, please see the package website : https://github.com/vlarmet/cppRouting/blob/master/README.md

Value

Named list with two useful attributes for the user :

nbnode : total number of vertices
dict$ref : vertices IDs

Examples

```
#Data describing edges of the graph
edges<-data.frame(from_vertex=c(0,0,1,1,2,2,3,4,4),
                  to_vertex=c(1,3,2,4,4,5,1,3,5),
                  cost=c(9,2,11,3,5,12,4,1,6))

#Construct directed and undirected graph
```

```
directed_graph<-makegraph(edges,directed=TRUE)
non_directed<-makegraph(edges,directed=FALSE)

#Visualizing directed and undirected graphs
if(requireNamespace("igraph",quietly = TRUE)){
  plot(igraph::graph_from_data_frame(edges))
  plot(igraph::graph_from_data_frame(edges,directed=FALSE))
}

#Coordinates of each nodes
coord<-data.frame(node=c(0,1,2,3,4,5),X=c(2,2,2,0,0,0),Y=c(0,2,2,0,2,4))

#Construct graph with coordinates
directed_graph2<-makegraph(edges, directed=TRUE, coords=coord)
```

`osm_bike_profile` *Default bicycle profile for OSM PBF import*

Description

Highway tag values kept for bicycle routing, with a class id and a flat assumed cycling pace (km/h, not a legal speed limit – `use_maxspeed` has no effect for this mode, see `graph_from_osmpbf`). Excludes `motorway`/`trunk` (+ `_link`) and pedestrian-only ways (`footway`/`pedestrian`/`steps`); includes `cycleway`/`path`/`track` in addition to ordinary roads. `clazz` ids reuse osm_car_profile's values for highway types shared with that profile.

Usage

```
osm_bike_profile()
```

Value

A data.frame with columns `highway`, `clazz` and `speed`, suitable as the `profile` element of cpp_extract's `osm` argument for a `.pbf` source.

`osm_car_profile` *Default car profile for OSM PBF import*

Description

Highway tag values kept for car routing, with a class id and a default speed (km/h) used unless overridden by a `maxspeed` tag. Mirrors a typical osm2po car profile (`wtr.finalMask = car`): motorway/trunk/primary/secondary (+ their `_link` variants), tertiary, residential, road and unclassified. `service`, `living_street`, `track`, `path`, cycleways and footways are intentionally excluded, as in that reference configuration.

Usage

```
osm_car_profile()
```

Value

A data.frame with columns `highway`, `clazz` and `speed`, suitable as the `profile` argument of cpp_extract for a `.pbf` source.

`osm_foot_profile` *Default pedestrian profile for OSM PBF import*

Description

Highway tag values kept for foot routing, with a class id and a flat assumed walking pace (km/h, not a legal speed limit – `use_maxspeed` has no effect for this mode, see `graph_from_osmpbf`). Excludes `motorway`/`trunk` (+ `_link`), which are not walkable; includes `footway`/`path`/`pedestrian`/`steps` in addition to ordinary roads, since pedestrians can generally use any street. `clazz` ids reuse osm_car_profile's values for highway types shared with that profile, so the same real-world road has the same class id regardless of mode.

Usage

```
osm_foot_profile()
```

Value

A data.frame with columns `highway`, `clazz` and `speed`, suitable as the `profile` element of cpp_extract's `osm` argument for a `.pbf` source.

`osm_hgv_profile` *Default heavy goods vehicle (truck) profile for OSM PBF import*

Description

Identical highway/clazz/speed table to osm_car_profile (trucks use the same road classes and, as a simplification, the same legal speed limits in this version) – the actual difference from `mode = "car"` is in access-tag resolution (`hgv=no`/`hgv=destination`/`hgv=yes`), not the set of roads considered. Weight/height/width restrictions (`maxweight`, `maxheight`, ...) are not resolved against a vehicle's actual dimensions in this version.

Usage

```
osm_hgv_profile()
```

Value

A data.frame with columns `highway`, `clazz` and `speed`, suitable as the `profile` element of cpp_extract's `osm` argument for a `.pbf` source.

to_df *Convert cppRouting graph to data.frame*

Description

Convert cppRouting graph to data.frame

Usage

```
to_df(Graph, aux = FALSE, capacity = FALSE, alpha = FALSE, beta = FALSE)
```

Arguments

Graph	An object generated by cppRouting::makegraph() or cpp_simplify() function.
aux	logical. If TRUE, append the graph's aux weight(s), when present (a plain vector becomes a column named aux; a named data.frame keeps its own column names). Silently ignored if the graph has no aux.
capacity	logical. If TRUE, append the graph's capacity, when present. Silently ignored if the graph has no capacity.
alpha	logical. If TRUE, append the graph's alpha, when present. Silently ignored if the graph has no alpha.
beta	logical. If TRUE, append the graph's beta, when present. Silently ignored if the graph has no beta.

Value

Data.frame with from, to and dist column, plus any of aux/capacity/alpha/beta requested.

Examples

```
#Simple directed graph

edges<-data.frame(from=c(1,2,3,4,5,6,7,8),
to=c(0,1,2,3,6,7,8,5),
dist=c(1,1,1,1,1,1,1,1))

#Construct cppRouting graph
graph<-makegraph(edges,directed=TRUE)

#Convert cppRouting graph to data.frame

df<-to_df(graph)
```

to_igraph *Convert cppRouting graph to an igraph object*

Description

Convert cppRouting graph to an igraph object

Usage

```
to_igraph(Graph, aux = FALSE, capacity = FALSE, alpha = FALSE, beta = FALSE)
```

Arguments

Graph	An object generated by cppRouting::makegraph() or cpp_simplify() function.
aux	logical. If TRUE, carry the graph's aux weight(s) over as edge attribute(s). See to_df.
capacity	logical. If TRUE, carry capacity over as an edge attribute.
alpha	logical. If TRUE, carry alpha over as an edge attribute.
beta	logical. If TRUE, carry beta over as an edge attribute.

Details

Vertices are named after the graph's own node refs (Graph$dict$ref); if the graph has coordinates, they are attached as vertex attributes x/y. Edge cost (dist) is exposed as the edge attribute weight, matching igraph's own convention for its shortest-path functions (igraph::distances(), igraph::shortest_paths(), etc. use weight by default) – so the result is directly usable with igraph's own algorithms (centrality, community detection, plotting...) without cppRouting needing to reimplement any of them.

Value

An igraph object. Requires the igraph package.

Examples

```
#Simple directed graph

edges<-data.frame(from=c(1,2,3,4,5,6,7,8),
to=c(0,1,2,3,6,7,8,5),
dist=c(1,1,1,1,1,1,1,1))

#Construct cppRouting graph
graph<-makegraph(edges,directed=TRUE)

#Convert cppRouting graph to an igraph object
if (requireNamespace("igraph", quietly = TRUE)) {
  ig <- to_igraph(graph)
}
```

Index

assign_traffic, 3, *7*, *16*, *27*

cpp_contract, 6, *8*, *15, 16*, *18–21*, *25*
cpp_customize, *7*, 8
cpp_extract, 9, *28, 29*
cpp_simplify, *7*, 13, *16*, *18*, *20*, *22, 23*, *25*

get_aon, *4*, *7, 8*, 14
get_detour, 16, *24*, *26*
get_distance_matrix, *7, 8*, 18, *21*, *23*
get_distance_pair, *7, 8*, *10*, *19*, 20
get_isochrone, 22, *24*, *26*
get_multi_paths, *19*, 23, *26*
get_path_pair, *7*, *21*, 24, *24*

makegraph, *3*, *5*, *7–9*, *12, 13*, *15, 16*, *18*, *20–23*, *25*, 26

osm_bike_profile, *11*, 28
osm_car_profile, *11*, 28, *28, 29*
osm_foot_profile, *11*, 29
osm_hgv_profile, *11*, 29

to_df, 30, *31*
to_igraph, 31