

# Package: cppRouting (via r-universe)

August 17, 2024

**Type** Package

**Title** Algorithms for Routing and Solving the Traffic Assignment Problem

**Version** 3.1

**Date** 2022-11-28

**Author** Vincent Larmet

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

**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](https://doi.org/10.1007/BF01386390)>), bidirectional Dijkstra (Goldberg, Andrew & Fonseca F. Werneck, Renato (2005) <<https://archive.siam.org/meetings/alenix05/papers/03agoldberg.pdf>>), A\* search (P. E. Hart, N. J. Nilsson et B. Raphael (1968) <[doi:10.1109/TSSC.1968.300136](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/10.1287/trsc.1120.0409)>), Algorithm-B (R. B. Dial (2006) <[doi:10.1016/j.trb.2006.02.008](https://doi.org/10.1016/j.trb.2006.02.008)>).

**License** GPL (>= 2)

**Encoding** UTF-8

**LazyData** true

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

**LinkingTo** Rcpp, RcppParallel, RcppProgress

**SystemRequirements** GNU make, C++11

**RoxygenNote** 7.2.1  
**URL** <https://github.com/vlarmet/cppRouting>  
**Suggests** knitr, rmarkdown, igraph  
**VignetteBuilder** knitr  
**NeedsCompilation** yes  
**Repository** <https://fastverse.r-universe.dev>  
**RemoteUrl** <https://github.com/vlarmet/cppRouting>  
**RemoteRef** HEAD  
**RemoteSha** b1715a8544f4485c87c1b41dfb08fd65af997b62

Contents

|                               |           |
|-------------------------------|-----------|
| assign_traffic . . . . .      | 2         |
| cpp_contract . . . . .        | 6         |
| cpp_simplify . . . . .        | 7         |
| get_aon . . . . .             | 8         |
| get_detour . . . . .          | 10        |
| get_distance_matrix . . . . . | 12        |
| get_distance_pair . . . . .   | 14        |
| get_isochrone . . . . .       | 16        |
| get_multi_paths . . . . .     | 17        |
| get_path_pair . . . . .       | 18        |
| makegraph . . . . .           | 20        |
| to_df . . . . .               | 22        |
| <b>Index</b>                  | <b>23</b> |

---

|                |   |
|----------------|---|
| assign_traffic | <i>Algorithms for solving the Traffic Assignment Problem (TAP).</i> |
|----------------|---|

---

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",
constant = 1,
dial_params = NULL,
verbose = TRUE
)

```

## Arguments

|             |  |
|-------------|--|
| Graph       | An object generated by <a href="#">makegraph</a> 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 or cbi. 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 |
| 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 :

$t = t_0 * (1 + a * (V/C)^b)$  with  $t$  = actual travel time (minutes),  $t_0$  = 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 :  $\text{abs}(\text{TSTT}/\text{SPTT} - 1)$  with TSTT (Total System Travel Time) =  $\text{sum}(\text{flow} * \text{cost})$ , SPTT (Shortest Path Travel Time) =  $\text{sum}(\text{aon} * \text{cost})$

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

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 and cphast. Optimal for dense matrix. One-to-many algorithm is called N times, with N being the number of unique from (or to) nodes

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 :

- `inner_iter` : 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 subtracting 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 = "bfgs")

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)
```

**Arguments**

|        |  |
|--------|--|
| Graph  | An object generated by <a href="#">makegraph</a> or <a href="#">cpp_simplify</a> function. |
| silent | Logical. If TRUE, progress is not displayed.   |

**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. 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.

**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)
```

---

|              |   |
|--------------|---|
| cpp_simplify | <i>Reduce the number of edges by removing non-intersection nodes, duplicated edges and isolated loops in the graph.</i> |
|--------------|---|

---

## 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 <a href="#">makegraph</a> 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

Additional edge attributes like aux, alpha, beta and capacity will be 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 <a href="#">makegraph</a> , or <a href="#">cpp_contract</a> 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 | <i>Return the nodes that can be reached in a detour time set around the shortest path</i> |
|------------|---|

---

## 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 <a href="#">makegraph</a> or <a href="#">cpp_simplify</a> 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 <a href="#">makegraph</a> , <a href="#">cpp_simplify</a> or <a href="#">cpp_contract</a> 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. If TRUE, the additional weight is summed along shortest paths.   |
| 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.

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.

## Note

It is not possible to aggregate auxiliary weights on a Graph object coming from [cpp\\_simplify](#) function.

## 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 <a href="#">makegraph</a> , <a href="#">cpp_simplify</a> or <a href="#">cpp_contract</a> 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. If TRUE, the additional weight is summed along shortest paths.   |
| 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.

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](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.

### Note

`from` and `to` must be the same length. It is not possible to aggregate auxiliary weights on a `Graph` object coming from `cpp_simplify` function.

### 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 | <i>Compute isochrones/isodistances from nodes.</i> |
|---------------|--|

---

## Description

Compute isochrones/isodistances from nodes.

## Usage

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

## Arguments

|        |  |
|--------|--|
| Graph  | An object generated by <a href="#">makegraph</a> or <a href="#">cpp_simplify</a> 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 <code>length(lim) &gt; 1</code> , 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 | <i>Compute all shortest paths between origin and destination nodes.</i> |
|-----------------|---|

---

## 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 <a href="#">makegraph</a> or <a href="#">cpp_simplify</a> 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 <a href="#">makegraph</a> , <a href="#">cpp_simplify</a> or <a href="#">cpp_contract</a> 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.

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 or a single value describing an additional edge weight.                                      |
| 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)
```

---

to\_df

---

*Convert cppRouting graph to data.frame*


---

### Description

Convert cppRouting graph to data.frame

### Usage

```
to_df(Graph)
```

### Arguments

Graph                      An object generated by cppRouting::makegraph() or cpp\_simplify() function.

### Value

Data.frame with from, to and dist column

### 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)
```

# Index

`assign_traffic`, [2](#), [9](#), [21](#)

`cpp_contract`, [6](#), [9](#), [12](#), [14](#), [15](#), [19](#)

`cpp_simplify`, [6](#), [7](#), [10](#), [12–17](#), [19](#)

`get_aon`, [3](#), [6](#), [8](#)

`get_detour`, [10](#), [18](#), [19](#)

`get_distance_matrix`, [6](#), [12](#), [15](#), [17](#)

`get_distance_pair`, [6](#), [13](#), [14](#)

`get_isochrone`, [16](#), [18](#), [19](#)

`get_multi_paths`, [13](#), [17](#), [19](#)

`get_path_pair`, [6](#), [15](#), [18](#), [18](#)

`makegraph`, [3](#), [5–7](#), [9](#), [10](#), [12](#), [14–17](#), [19](#), [20](#)

`to_df`, [22](#)