

Package: graphvec (via r-universe)

October 8, 2026

Version 0.1.0

Title Vectorised Graph Data Structures

Description Extends vectors to include graph relationships between their elements, and offers tools to compute useful summaries of the graph structure for use in summarising, filtering, and otherwise manipulating the graph. Node identity is positional rather than value-based, so isolated nodes and repeated values are represented without special handling. Three complementary data structures are provided, each an ordinary vector that stays a column in a data frame and slices consistently with it: 'node_vec' is vectorised along the nodes of a graph, 'edge_vec' is vectorised along its edges, and 'agg_vec' (with the tabular 'agg_df') represents the aggregation structure common in data analysis, such as a total row over a set of categories. This makes graph relationships a native part of tidy rectangular data analysis workflows, alongside tools such as those in 'dplyr'. Each of these can also be converted to 'igraph' objects for further analysis.

License MIT + file LICENSE

Imports cli, tibble, vctrs (>= 0.6.0)

Suggests callr, crayon, dplyr, ggplot2, igraph, pillar, purrr, readr, stringr, testthat (>= 3.0.0), tidyr, tsibble, waldo

Config/testthat/edition 3

Encoding UTF-8

Roxygen list(markdown = TRUE)

Config/roxygen2/version 8.0.0

URL <https://pkg.mitchelloharawild.com/graphvec/>,
<https://github.com/mitchelloharawild/graphvec>

BugReports <https://github.com/mitchelloharawild/graphvec/issues>

Config/rextendr/version 0.5.0

SystemRequirements Cargo (Rust's package manager), rustc >= 1.85.0, xz

Depends R (>= 4.2)
Config/pak/sysreqs xz-utils libclang-dev
Repository <https://mitchelloharawild.r-universe.dev>
Date/Publication 2026-10-08 00:41:04 UTC
RemoteUrl <https://github.com/mitchelloharawild/graphvec>
RemoteRef HEAD
RemoteSha 315b9b296e39774429e51fc3c29859531495376d

Contents

agg_df	2
agg_vec	3
as.igraph	5
edge_heads	6
edge_incident	7
edge_is_loop	8
edge_vec	9
graph_density	10
graph_is_directed	11
GraphBackend	12
is_aggregated	14
n_nodes	15
new_edge_vec	16
new_node_vec	17
node_degree	17
node_incident	18
node_is_isolated	19
node_neighbors	20
node_values	21
node_vec	22
nodes.agg_df	23
Index	26

agg_df	Create an aggregation table
--------	-----------------------------

Description

An `agg_df` is a table of `agg_vec()` columns, one row per level of a (possibly crossed) aggregation structure – e.g. Purpose and State columns where some rows total one dimension, some the other, some both. `nodes()/edges()` reorient it into a graph: a row is a child of another row whenever the parent aggregates exactly one more column and matches on every other column’s disaggregated value.

Usage

```
agg_df(...)
```

Arguments

... Named agg_vec columns, all the same length.

Details

An agg_df's rows form a lattice of aggregation levels, with one elementary chain per column (or nested group of columns, e.g. State/Region together), combined the way a symbolic formula like Purpose * (State / Region) would combine them. See [agg_vec\(\)](#) for the OLAP CUBE/ROLLUP correspondence this generalises, and for what the representation deliberately does not support: it is limited to nested and crossed dimension structures inferred by key matching, not a general hypergraph, and a partially-crossed ("pruned"/unbalanced) column combination gives a meet-semilattice missing some joins rather than an error.

Value

An agg_df object.

Examples

```
agg_df(
  Purpose = agg_vec(c(NA, "Business", "Holiday"), c(TRUE, FALSE, FALSE)),
  State = agg_vec(c("NSW", NA, NA), c(FALSE, TRUE, TRUE))
)
```

agg_vec	<i>Create an aggregation vector</i>
---------	-------------------------------------

Description

An aggregation vector is a special type of [node_vec\(\)](#) consisting of a single parent (the 'aggregated' value) and its children. Aggregated values are identified by a logical vector passed to the aggregated argument, and disaggregated values are provided in x. Aggregated values are displayed as <aggregated> by default.

Usage

```
agg_vec(x = character(), aggregated = logical(NROW(x)))
```

Arguments

x	The vector of values.
aggregated	A logical vector, the same length as x and with no missing values, to identify which values are <aggregated>.

Details

agg_vec represents one *dimension* of aggregation as a two-level chain (disaggregated values below their shared <aggregated> total). It relies on matching key values across columns, not a stored edge, to find each row's parent (see `agg_df()`). This is the same construction as SQL's CUBE/ROLLUP/GROUPING SETS (Gray, Bosworth, Layman & Pirahesh, 1996), where <aggregated> plays the role of their sentinel NULL marking "summed over this column"; several agg_vec columns combined in an `agg_df()` produce exactly the "lattice of cuboids" of Harinarayan, Rajaraman & Ullman (1996).

Because it infers parents rather than storing them, agg_vec/agg_df only support the nested (hierarchical) and crossed (grouped) dimension structures a symbolic formula like Purpose * (State / Region) can produce: always at least a meet-semilattice, and a full distributive lattice when every pair of combined columns is fully crossed. They cannot represent a hypergraph, that is, an aggregate whose value is jointly determined by one *specific, tagged* group of children (a directed hyperedge/AND-arc), as opposed to an ordinary total reachable by matching key values. A well-formed agg_vec has at most one row per distinct combination of aggregated/disaggregated values. A duplicate combination isn't an error: `nodes()` (and so `igraph::as.igraph()`) links every disaggregated row to the first <aggregated> row, so a duplicate disaggregated row gets its own edge and a later duplicate <aggregated> row is left without any. For arbitrary graphs, such as explicit hyperedges or aggregation structures with no shared bottom series across pathways (divergent groups), use `node_vec()/edge_vec()` directly instead.

Value

An agg_vec object.

Plotting with ggplot2

An agg_vec's ggplot2 `scale_type` is "agg", then "discrete", so an extension package can supply default `scale_*_agg()` scales. Without one, colour, fill, shape and the like get ggplot2's discrete scales, labelled by `format()` with <aggregated> as the last level; the x and y positions don't work, so plot `format(x)` there.

References

Gray, J., Bosworth, A., Layman, A., & Pirahesh, H. (1996). Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *ICDE*.

Harinarayan, V., Rajaraman, A., & Ullman, J. D. (1996). Implementing Data Cubes Efficiently. *SIGMOD*.

Examples

```
agg_vec(
  x = c(NA, "A", "B"),
  aggregated = c(TRUE, FALSE, FALSE)
)
```

as.igraph

*Convert a graph vector to an igraph object***Description**

Methods for converting node_vec, agg_vec, agg_df, and edge_vec objects to `igraph::igraph()` objects, keeping their node values and edge attributes.

Usage

```
## S3 method for class 'agg_vec'
as.igraph(x, ...)

## S3 method for class 'agg_df'
as.igraph(x, ...)

## S3 method for class 'node_vec'
as.igraph(x, ...)

## S3 method for class 'edge_vec'
as.igraph(x, ...)
```

Arguments

`x` A node_vec, agg_vec, agg_df, or edge_vec object.
`...` Additional arguments (currently unused).

Details

The vertices are the nodes, in order, and carry the node values as vertex attributes, following igraph's convention: a vector of node values becomes the name attribute, and each column of a data frame of node values becomes an attribute of the same name. The edges are in order too, with each edge attribute column as an igraph edge attribute. Values keep their type (a factor stays a factor, a date a date). A node_vec slice converts to the subgraph its nodes induce, and an edge_vec to every node of its graph, with the edges of the slice.

An agg_vec or agg_df is converted as the graph `nodes()` gives it, so `as.igraph(x)` is the same as `as.igraph(nodes(x))`: each row links to the rows that aggregate it (see `agg_df()`), and each column is a vertex attribute holding the column's agg_vec (named value for an agg_vec).

igraph has no concept of a hyperedge (an edge with more than one node in its from or to role). Converting a node_vec/edge_vec with a hyperedge column raises an error instead of silently dropping or flattening the extra incidence.

igraph has no missing edges either, so converting an edge_vec with a missing edge (e.g. from `vctrs::vec_init()`) raises an error: drop it first, e.g. with `x[!is.na(x)]`.

Value

An `igraph::igraph()` object.

See Also

[node_vec\(\)](#), [agg_vec\(\)](#), [agg_df\(\)](#), [edge_vec\(\)](#)

Examples

```
if (requireNamespace("igraph", quietly = TRUE)) {
  g <- node_vec(
    x = c("A", "B", "C"),
    from = c(1L, 2L),
    to = c(2L, 3L)
  )
  igraph::as.igraph(g)
}
```

edge_heads

Edge endpoints, edge-aligned

Description

The from/to node of every edge, aligned to the edge axis (length `n_edges(x)`) – `e$from/e$to` restated as functions so they're reachable from either orientation, not just an `edge_vec`. $O(M)$.

Usage

`edge_heads(x)`

`edge_tails(x)`

Arguments

`x` A `node_vec` or `edge_vec`. Either orientation is accepted and reoriented internally. Hyperedges are not yet supported.

Details

Values are node *identities* (the node data itself), not positions – use [node_incident\(\)](#) for positions.

Value

A vector of node identities, one per edge.

Examples

```
g <- node_vec(x = c("A", "B", "C"), from = c(1L, 2L), to = c(2L, 3L))
edge_tails(g)
edge_heads(g)
```

edge_incident	<i>Edges touching a node</i>
---------------	------------------------------

Description

The edges incident to node *i*, in increasing edge position, with the same mode and self-loop handling as [node_neighbors\(\)](#) – so `length(edge_incident(x, i))` is `node_degree(x)[i]` too.

Usage

```
edge_incident(x, i, mode = c("out", "in", "all"))
```

Arguments

<i>x</i>	A <code>node_vec</code> or <code>edge_vec</code> . Either orientation is accepted and reoriented internally. Hyperedges are not yet supported.
<i>i</i>	One or more node positions to query.
<i>mode</i>	For a directed graph: "out" (successors), "in" (predecessors), or "all" (both – possibly with duplicates if <i>i</i> and another node are joined in both directions). Ignored for an undirected graph, where every incident edge is symmetric; an undirected self-loop appears once (see node_degree() on self-loops).

Details

Answered in $O(d)$ per query node, like [node_neighbors\(\)](#), from an index of each node's incident edges that the graph builds the first time it is needed.

Value

An integer vector of edge positions when *i* has length 1; a list of integer vectors (one per element of *i*) otherwise.

Examples

```
g <- node_vec(x = c("A", "B", "C"), from = c(1L, 2L), to = c(2L, 3L))
edge_incident(g, 2, mode = "all")
```

edge_is_loop

Edge position predicates

Description

edge_is_loop() flags a self-loop (from == to), free. edge_multiplicity() counts, for each edge, how many edges share its endpoints (the same ordered pair for a directed graph, the same unordered pair otherwise); edge_is_multi() flags an edge with multiplicity greater than one, i.e. one with at least one parallel duplicate. Both are O(M).

Usage

```
edge_is_loop(x)
```

```
edge_multiplicity(x)
```

```
edge_is_multi(x)
```

Arguments

x A node_vec or edge_vec. Either orientation is accepted and reoriented internally. Hyperedges are not yet supported.

Details

Multiplicity is counted among the edges x *currently* holds: an edge_vec sliced down to one of a pair of parallel edges has multiplicity 1, matching n_edges(x) and length(x). A missing edge (e.g. from vctrs::vec_init()) has no endpoints, so its multiplicity is NA and it is not parallel to any edge, not even another missing one.

Value

A logical vector of length n_edges(x), except edge_multiplicity(), which returns an integer vector of the same length.

Examples

```
g <- node_vec(x = c("A", "B", "C"), from = c(1L, 1L, 3L), to = c(2L, 2L, 3L))
edge_is_loop(g)
edge_multiplicity(g)
edge_is_multi(g)
```

edge_vec

*Graph vector along edges***Description**

An edge_vec is a vector of graph edges with associated node data stored as attributes.

Usage

```
edge_vec(
  from = integer(),
  to = integer(),
  ...,
  nodes = data.frame(),
  directed = TRUE
)
```

Arguments

from	Integer vector of 'from' node indices, one per edge. A list of integer vectors instead opts into hyperedges: each element gives the (zero or more) 'from' node positions for that edge, an ordinary edge being the length-1 case.
to	Integer vector of 'to' node indices, or a list of integer vectors for hyperedges, the same way as from.
...	Named edge attribute vectors (e.g. weight = c(1, 2, 5)), recycled to the number of edges. from and to are reserved and cannot be used as attribute names. Attribute columns are stored on the edge table itself, so they slice, replicate, and reorient with the edges they belong to (see nodes() / edges()).
nodes	Vector of node data (any vector, including a data frame of node attributes). Every from/to position must be one of its elements (rows). The default, a zero-column data frame, means no node data: the nodes are then every position up to the largest one in from and to.
directed	A single logical value: is incidence ordered (from -> to) or symmetric?

Value

An edge_vec object.

Slicing, combining and equality

Every edge_vec belongs to a graph, which slicing keeps: `e[i]` selects edges of the same graph, between the same nodes.

Combining edge_vecs, with `c()`, `[<-`, `vctrs::vec_c()`, `dplyr::bind_rows()`, `dplyr::rows_patch()` and the like, shares the graph between inputs of the same graph: their edges keep pointing at the

same nodes, and the nodes aren't copied, so `c(e, e)` has the same nodes as `e`. `Edge_vecs` of different graphs are combined as a disjoint union of their graphs (each graph's nodes once, in order of first appearance); the edges always stay in the order they're combined in.

Two edges are equal when they are edges of the same graph between the same node positions, with the same edge attributes, whether or not there is node data. This is what `duplicated()`, `unique()`, `vctrs::vec_in()`, `joins`, `dplyr::distinct()` and `dplyr::count()` use, so `e[2]` matches the second edge of `e` (or of `c(e, e)`), but edges of two separately built `edge_vecs` never match, even with identical labels: to match those by label, compare `format(e)` explicitly, e.g. `dplyr::mutate(df, key = format(e))` before joining by `"key"`. Base `match()` and `%in%` only work on an `edge_vec` with edge attributes (on R ≥ 4.3); `vctrs::vec_match()` and `vctrs::vec_in()` work on any, and are faster on long vectors, as they hash the edges directly rather than first building a string key for every element. `Edge_vecs` sort by the node values at each end (then the edge attributes, then node positions to break ties). Hyperedges have no graph identity: they compare by node values and always combine as a disjoint union.

Plotting with ggplot2

An `edge_vec`'s ggplot2 `scale type` is `"edge"`, then `"discrete"`, so an extension package can supply default `scale_*_edge()` scales. Without one, colour, fill, shape and the like get ggplot2's discrete scales, labelled by `format()`; the x and y positions don't work, so plot `format(e)` there.

Examples

```
g <- edge_vec(
  from = c(1L, 2L, 1L, 3L),
  to = c(2L, 3L, 3L, 1L),
  weight = c(1, 2, 5, 3),
  nodes = data.frame(
    id = 1:3,
    label = c("A", "B", "C")
  )
)

# Access node data via `$$`
g$from$label
g$to$label

# Access an edge attribute via `$$`
g$weight
```

graph_density

Graph density

Description

The fraction of possible edges that are actually present: M over the number of ordered pairs (directed) or unordered pairs (undirected) of distinct nodes. Self-loops and parallel edges still count towards M , so a multigraph can exceed a density of 1.

Usage

```
graph_density(x)
```

Arguments

x A node_vec or edge_vec. Either orientation is accepted and reoriented internally. Hyperedges are not yet supported.

Value

A single double, or NaN for a graph with fewer than 2 nodes (no possible edges to take a fraction of).

Examples

```
g <- node_vec(x = c("A", "B", "C"), from = c(1L, 2L), to = c(2L, 3L))
graph_density(g)
```

graph_is_directed	<i>Whole-graph predicates</i>
-------------------	-------------------------------

Description

graph_is_directed() reads the directed flag x was built with – is incidence ordered (from -> to) or symmetric? graph_has_loops() is any(edge_is_loop(x)): does any edge join a node to itself?

Usage

```
graph_is_directed(x)
```

```
graph_has_loops(x)
```

Arguments

x A node_vec or edge_vec. Either orientation is accepted and reoriented internally. Hyperedges are not yet supported.

Value

A single logical.

Examples

```
g <- node_vec(x = c("A", "B", "C"), from = c(1L, 2L), to = c(2L, 3L))
graph_is_directed(g)
graph_has_loops(g)

looped <- node_vec(x = c("A", "B"), from = 1L, to = 1L)
graph_has_loops(looped)
```

GraphBackend

The shared topology backing a node_vec/edge_vec pair (non-hyperedge case only – see _dev/RUST_BACKEND.md). Wraps one of several physical Repr variants, auto-selected at construction from graph shape (_dev/petgraph_data_types.md S3/S5); directed is metadata every variant’s query methods interpret, not itself part of the shape decision. This keeps the object immutable and shareable: a node_vec, its edges() reorientation, and any edge_vec sliced from it can all hold the same pointer.

Description

The shared topology backing a node_vec/edge_vec pair (non-hyperedge case only – see _dev/RUST_BACKEND.md). Wraps one of several physical Repr variants, auto-selected at construction from graph shape (_dev/petgraph_data_types.md S3/S5); directed is metadata every variant’s query methods interpret, not itself part of the shape decision. This keeps the object immutable and shareable: a node_vec, its edges() reorientation, and any edge_vec sliced from it can all hold the same pointer.

Usage

GraphBackend

Methods

Method new: Build a graph on n nodes from 1-based from/to positions. Automatically picks the cheapest Repr the graph’s shape qualifies for (_dev/petgraph_data_types.md S3/S5 – this decision belongs here, not in R, so there is exactly one place shape detection can drift out of sync with the representation it feeds). Edges are added in input order and never removed afterwards, so edge ids (0-based internally, 1-based at the R boundary) stay stable and match the row order of the R-side edge attribute table, for every representation.

Errors (an R error, not a crash) unless every from/to position is in $1..=n$ and the two have the same length: an NA arrives as `i32::MIN`, and a position outside the graph would otherwise index out of bounds or, at 0 or below, wrap to a huge usize allocation. The R constructors check this first, with friendlier messages.

Method uid: A number unique to this graph among every graph built in the R session, never reused (unlike a memory address, which can be once a graph is garbage collected), and starting from a random point per session so it doesn’t collide with a graph saved from another one. This is

a graph's identity: two R objects are of the same graph exactly when their GraphBackends have the same uid(), including after a saveRDS()/readRDS() round trip, which rebuilds the graph behind a new address but keeps its uid (graphvec_backend_revive()). A double, so it fits an R numeric exactly (it's kept to 53 bits).

Method repr_name: Which physical representation this backend picked, as a stable name ("general", "dense", "csr") – the one diagnostic entry point for "which Repr is this", replacing what used to be a separate is_tree()/is_dense()/is_csr() boolean per variant. That pattern grew one new #[extendr] method – permanent, exported R API the moment it's added, per this file's own "the method set is the contract" principle – for every future Repr addition; this single method's match arm count grows with Repr instead, so the R-visible surface stays fixed no matter how many representations GraphBackend eventually holds.

Method degrees: Every node's degree, in node order – degree() (below) for each node in one call, so node_degree() and the node predicates cross the R/Rust boundary once rather than once per node. Same mode and self-loop semantics as degree(), which it is defined by.

Method neighbors_many: The neighbours of every node in nodes (1-based, repeats allowed), each in increasing order, as one CSR pair list(ptr, idx): node nodes[k]'s neighbours are idx[(ptr[k] + 1):ptr[k + 1]], with ptr 0-based offsets of length length(nodes) + 1. One call for any number of query nodes (node_neighbors() splits it into its list result), with the same mode, self-loop and one-entry-per-edge semantics as neighbors(). Sorted here because neighbors()'s own order depends on which Repr the graph picked.

Method incident_many: The ids of the edges incident to every node in nodes (1-based, repeats allowed), each in increasing order, as the same CSR pair list(ptr, idx) neighbors_many() returns, with 1-based edge ids in idx. Same mode and self-loop semantics as neighbors(): an undirected self-loop is listed once, a directed one twice under "all", so node nodes[k] has degree() incident edges. Answered from the edge-id incidence index (built on first use), O(degree) per node rather than a scan of every edge.

Method has_edge: Adjacency test. For an undirected graph, checks both orientations.

Method edge_endpoints: Edges as 1-based (from, to) pairs: all of them in construction/edge-id order when ids is NULL, otherwise those of the 1-based edge ids ids, in ids' order (repeats allowed). An NA id is a missing edge, with NA endpoints. Backs edge_vec's format()/\$from/\$to and as.igraph() – no R-side edge table is needed for topology once this exists – and lets a slice of a big graph read only its own edges' endpoints, O(length(ids)).

Method induced_subgraph: Induced-subgraph remap for a node set sliced/replicated according to idx: idx[j] is the 1-based *old* node position that *new* node j came from, 0 meaning "no source" (a new node with nothing mapping to it – R's NA_integer_ maps to this sentinel before the call, since it doesn't survive the R -> Rust integer conversion). Duplicate entries mean a replicated node. An edge is dropped if either endpoint has no surviving new position; an edge whose endpoint(s) were replicated is cloned once per combination.

Returns list(from, to, source_edge), all 1-based and in a stable order (new to slowest, new from fastest, matching expand.grid(from, to)'s order): source_edge[k] is the original edge id that surviving new edge k was cloned from, so the R side can carry edge attribute columns across replication with edges[source_edge,].

Local to the selection: one of at most `n_nodes()` / 4 positions walks only the out-edges of its own nodes, through the edge-id incidence index, and sorts the edges it keeps, so slicing a few nodes off a big graph costs their degree, not the graph's size. A larger one scans every edge in order instead, which is cheaper than that walk and sort once the selection is a sizeable share of the graph, and needs no index. Either way the result is the same, in edge-id order.

This is representation-independent: it reads edges through `edge_at()` and the incidence index alone (S4's edge-identity contract, upheld there), never `self.repr` directly, so it needs no per-variant duplicate. The result is a plain from/to/source_edge list either way – it does not construct a new `GraphBackend` itself (R reconstructs one from these lists via `new()`, confirmed by grepping `R/node_vec.R`'s `[.node_vec]`), so the *new* backend's shape (which needn't match the old one – both terms of the density ratio move here, `N` becoming `idx.len()` and `M` however many edges survived or were cloned, so e.g. dropping the centre of a `Repr::Dense` star strips every edge at once and leaves something far below `DENSE_THRESHOLD`, while dropping the isolated nodes that were holding a `Repr::Csr` graph under that threshold pushes what's left above it) is re-decided by `new()`'s own detection from scratch, same as it would be for any other from/to/directed input.

is_aggregated

Is the element an aggregation of smaller data

Description

Is the element an aggregation of smaller data

Usage

```
is_aggregated(x)
```

Arguments

`x` An object.

Value

A logical vector indicating which elements are aggregated.

See Also

[agg_vec\(\)](#)

Examples

```
v <- agg_vec(c(NA, "A", "B"), c(TRUE, FALSE, FALSE))
is_aggregated(v)
```

n_nodes	<i>Node and edge counts</i>
---------	-----------------------------

Description

The number of nodes/edges in `x`, kept bare rather than `graph_`-prefixed – the "ask either orientation" counts tied to `length()` (`_dev/OPERATIONS.md` §1 rule 3). Both are $O(1)$.

Usage

```
n_nodes(x)
```

```
n_edges(x)
```

Arguments

<code>x</code>	A <code>node_vec</code> or <code>edge_vec</code> . Either orientation is accepted and reoriented internally. Hyperedges are not yet supported.
----------------	--

Details

`n_edges()` counts the edges `x` *currently* holds, so it always agrees with `length(edges(x))`: an `edge_vec` sliced away from its graph keeps the shared graph pointer but only a selection of its edges, and it is the selection that counts.

A missing edge (e.g. from `vctrs::vec_init()`, `lag()` or a join) is still an element of an `edge_vec`, so `n_edges()` counts it, matching `length(x)`. It joins no nodes, though, so `node_degree()` and the other node measures ignore it, and `nodes()` leaves it out of the graph.

Value

A single integer.

Examples

```
g <- node_vec(x = c("A", "B", "C"), from = c(1L, 2L), to = c(2L, 3L))
n_nodes(g)
n_edges(g)
```

new_edge_vec	<i>Constructor function for edge_vec</i>
--------------	--

Description

Constructor function for edge_vec

Usage

```
new_edge_vec(
  from = integer(),
  to = integer(),
  ...,
  nodes = data.frame(),
  directed = TRUE
)
```

Arguments

from	Integer vector of 'from' node indices, or a list of integer vectors for hyperedges.
to	Integer vector of 'to' node indices, or a list of integer vectors for hyperedges.
...	Named edge attribute fields, already recycled to the number of edges.
nodes	Vector of node data (any vector, including a data frame of node attributes).
directed	A single logical value: is incidence ordered (from -> to) or symmetric?

Value

An edge_vec object.

Examples

```
new_edge_vec(
  from = c(1L, 2L),
  to = c(2L, 3L),
  nodes = data.frame(label = c("A", "B", "C"))
)
```

new_node_vec	<i>Constructor function for node_vec</i>
--------------	--

Description

Constructor function for node_vec

Usage

```
new_node_vec(  
  x = list(),  
  edges = data.frame(from = integer(), to = integer()),  
  directed = TRUE  
)
```

Arguments

x	A vector representing the nodes in the graph.
edges	A data frame with columns from and to representing the edges
directed	A single logical value: is incidence ordered (from -> to) or symmetric?

Value

A node_vec object.

Examples

```
new_node_vec(  
  x = c("A", "B", "C"),  
  edges = data.frame(from = c(1L, 2L), to = c(2L, 3L))  
)
```

node_degree	<i>Node degree</i>
-------------	--------------------

Description

The number of edges incident to each node, for every node at once.

Usage

```
node_degree(x, mode = c("all", "out", "in"))
```

Arguments

x	A node_vec or edge_vec. Either orientation is accepted and reoriented internally. Hyperedges are not yet supported.
mode	For a directed graph: "all" (in- plus out-degree), "out", or "in". Ignored for an undirected graph, where in- and out-degree coincide.

Details

Answered directly by the shared GraphBackend (_dev/RUST_BACKEND.md): one $O(1)$ lookup per node for "out"/"in", and for "all" on a directed graph, from the degree counts cached when the graph was built – $O(N)$ overall, with no adjacency index to rebuild per call.

Value

An integer vector of length `n_nodes(x)`.

Self-loops

In an **undirected** graph a self-loop counts **once**, matching the backend's own convention (and 'petgraph's `neighbors_undirected()`) – one entry per incident edge, so `node_degree()` always equals `length(node_neighbors())`. Note that this differs from 'igraph', which follows the text-book "a self-loop adds 2 to the undirected degree" convention. In a **directed** graph a self-loop contributes 1 to the out-degree and 1 to the in-degree, so 2 under `mode = "all"`.

Examples

```
g <- node_vec(x = c("A", "B", "C"), from = c(1L, 2L), to = c(2L, 3L))
node_degree(g)
node_degree(g, mode = "out")
```

node_incident	<i>Endpoint nodes of an edge</i>
---------------	----------------------------------

Description

The nodes touching edge `i` – its `from` and `to` positions, in that order (a self-loop returns the same position twice). $O(1)$ per edge: an edge's own endpoints are stored directly, so no adjacency lookup is involved.

Usage

```
node_incident(x, i)
```

Arguments

x	A node_vec or edge_vec. Either orientation is accepted and reoriented internally. Hyperedges are not yet supported.
i	One or more edge positions to query.

Value

An integer vector of length 2 (`c(from, to)`) when `i` has length 1; a list of such vectors (one per element of `i`) otherwise.

Examples

```
g <- node_vec(x = c("A", "B", "C"), from = c(1L, 2L), to = c(2L, 3L))
node_incident(g, 1)
node_incident(g, 1:2)
```

node_is_isolated	<i>Node position predicates</i>
------------------	---------------------------------

Description

Free $O(1)$ -per-node predicates on `node_degree()`: a node with no incident edges at all, no incoming edge, or no outgoing edge, respectively. For an undirected graph, `node_is_root()/node_is_leaf()` coincide with `node_is_isolated()` – in- and out-degree are the same thing there, so "no incoming edge" and "no incident edge" mean the same thing.

Usage

```
node_is_isolated(x)

node_is_root(x)

node_is_leaf(x)
```

Arguments

<code>x</code>	A <code>node_vec</code> or <code>edge_vec</code> . Either orientation is accepted and reoriented internally. Hyperedges are not yet supported.
----------------	--

Details

A self-loop makes its node non-isolated (and neither a root nor a leaf in the directed case), since it is an incident edge like any other.

Value

A logical vector of length `n_nodes(x)`.

Examples

```
g <- node_vec(x = c("A", "B", "C"), from = c(1L, 2L), to = c(2L, 3L))
node_is_isolated(g)
node_is_root(g)
node_is_leaf(g)
```

node_neighbors	<i>Neighboring nodes</i>
----------------	--------------------------

Description

The nodes adjacent to node *i*, answered in $O(d)$ by the shared GraphBackend's adjacency (*d* being the degree of *i*) – no per-call index build, unlike the $O(M)$ scan a bare edge list would need.

Usage

```
node_neighbors(x, i, mode = c("out", "in", "all"))
node_parents(x, i)
node_children(x, i)
```

Arguments

<i>x</i>	A <code>node_vec</code> or <code>edge_vec</code> . Either orientation is accepted and reoriented internally. Hyperedges are not yet supported.
<i>i</i>	One or more node positions to query.
<i>mode</i>	For a directed graph: "out" (successors), "in" (predecessors), or "all" (both – possibly with duplicates if <i>i</i> and another node are joined in both directions). Ignored for an undirected graph, where every incident edge is symmetric; an undirected self-loop appears once (see node_degree() on self-loops).

Details

`node_parents()/node_children()` are the `mode = "in"/mode = "out"` aliases, named for the directed reading.

One entry is returned **per incident edge**, not per distinct neighbour, so parallel edges produce repeated positions and `length(node_neighbors(x, i))` is `node_degree(x)[i]`. Positions come back in increasing order: the backend's own iteration order depends on which physical representation the graph picked (`_dev/petgraph_data_types.md`), so it is sorted here to keep the result reproducible. It is therefore *not* positionally paired with [edge_incident\(\)](#), which is ordered by edge position.

Value

An integer vector of node positions when *i* has length 1; a list of integer vectors (one per element of *i*) otherwise.

Examples

```
g <- node_vec(x = c("A", "B", "C"), from = c(1L, 2L), to = c(2L, 3L))
node_neighbors(g, 2)
node_neighbors(g, 1:3)
node_children(g, 1)
```

node_values	<i>The plain values of a node_vec</i>
-------------	---------------------------------------

Description

node_values() gives the values a node_vec was built from (an atomic vector or a data frame of node attributes), without the graph or the nodes' identity. Nodes compare by identity (see [node_vec\(\)](#)): ==, match(), %in% and joins only match nodes of the same graph, never a plain value, so comparing nodes by their labels is done explicitly on node_values() (or on format()).

Usage

```
node_values(x, ...)
```

Arguments

x	A node_vec. Plain vectors and data frames, which are already values, are returned as they are.
...	Passed on to methods.

Value

The node values, one per node: a vector of the type the node_vec was built from, or a data frame.

Examples

```
n <- node_vec(c("A", "B", "C", "A"), from = 1:3, to = 2:4)
node_values(n)

# Match nodes by value, rather than by identity:
node_values(n) %in% c("A", "B")
node_values(n) == "A"
```

node_vec	<i>Graph vector along nodes</i>
----------	---------------------------------

Description

A node_vec is a vector of graph nodes with associated edges stored as attributes.

Usage

```
node_vec(x = list(), from = integer(), to = integer(), ..., directed = TRUE)
```

Arguments

x	A vector representing the nodes in the graph.
from	Integer vector of 'from' node positions into x, one per edge. A list of integer vectors instead opts into hyperedges: each element gives the (zero or more) 'from' positions for that edge, an ordinary edge being the length-1 case.
to	Integer vector of 'to' node positions into x, or a list of integer vectors for hyperedges, the same way as from.
...	Named edge attribute vectors (e.g. weight = c(1, 2, 5)), recycled to the number of edges. from and to are reserved and cannot be used as attribute names. Attribute columns are stored on the edge table itself, so they slice, replicate, and reorient with the edges they belong to (see nodes()/edges()).
directed	A single logical value: is incidence ordered (from -> to) or symmetric?

Value

A node_vec object.

Slicing and combining

Every node_vec belongs to a graph. Slicing (x[i], dplyr::filter(), dplyr::arrange(), ...) keeps that graph and selects nodes in it: the slice's edges are the induced subgraph on its nodes, so an edge is dropped once either of its ends is, and a node repeated within a slice (x[c(1, 1)]) is a separate copy of the graph, as for [c\(\)](#) below.

Combining node_vecs, with [c\(\)](#), [\[<-](#), [\[\[<-](#), [vctrs::vec_c\(\)](#), [dplyr::bind_rows\(\)](#), [dplyr::if_else\(\)](#), [dplyr::rows_patch\(\)](#) and the like, groups the nodes by the graph they belong to:

- Nodes of the same graph go back into that graph, keeping every edge between them, including edges between nodes that came from different inputs: `c(x[1:2], x[3:4])` is x, and `if_else(cond, x, x)` or `x[2] <- x[2]` keep all of x's edges.
- The same node coming from two different inputs makes them separate copies of the graph, a disjoint union: `c(x, x)` has two copies of every node and edge, and `c(x[1:2], x[2:3])` two copies of x[2], each with only its own input's edges. Repeats within one slice make copies the same way: `x[c(1:n, 1:n)]`, `rep(x, 2)` and `vctrs::vec_rep(x, 2)` are `c(x, x)`, and the k-th repeat of each node (`x[c(1, 1, 2)]`, `rep(x, each = 2)`) belongs to the k-th copy.

- Nodes of different graphs are a disjoint union of those graphs. The nodes always stay in the order they're combined in.

Assigning plain values with `[<-` relabels nodes and keeps the graph. Two nodes are equal when they are (copies of) the same node of the same graph, with the same value. This is what `==`, `duplicated()`, `unique()`, `vctrs::vec_equal()` and joins use, so `x[1]` equals the first node of `x` (or of `c(x, x)`), but not another node with the same label, nor a node of a separately built `node_vec`. A node is never equal to a plain value: `x == "A"` is an error, and `match()` and `%in%` (on R >= 4.3) find no match. Compare the values explicitly instead, with `node_values(x) == "A"` or `node_values(x) %in% "A"` (see `node_values()`). On long vectors, prefer `vctrs::vec_match()` and `vctrs::vec_in()`, which hash the nodes directly, to base `match()` and `%in%`, which first build a string key for every element of both vectors. Nodes have no order or arithmetic of their own either, so `<`, `+` and the other operators error, as for an `edge_vec`. Hyperedge `node_vecs` have no graph identity and compare by value.

Plotting with ggplot2

A `node_vec`'s ggplot2 `scale type` is "node", then the scale types of its values with "discrete" in place of "continuous" ("discrete" for data-frame values), so an extension package can supply default `scale_*_node()` scales. Without one, ggplot2 uses its own scales: character, factor and logical nodes get discrete scales, labelled by their values (a factor's in level order), and dates and date-times get date scales. Nodes have no arithmetic, so numeric nodes (and data-frame ones) get discrete scales instead of continuous ones: these work for colour, fill, shape and the like, labelled by `format()`, but not for the x and y positions. Plot `node_values(x)` there for a continuous scale, or `format(x)` for a discrete one. Points are grouped by node, so two nodes with the same label are separate groups.

Examples

```
g <- node_vec(
  x = factor(c("A", "B", "C")),
  from = c(1L, 2L, 1L),
  to = c(2L, 3L, 1L),
  weight = c(1, 2, 5)
)
g

if (requireNamespace("igraph", quietly = TRUE)) {
  igraph::as.igraph(g)
}
```

Description

nodes() and edges() return the same underlying graph, enumerated along the node or edge axis respectively, regardless of which orientation *x* started in. Both directions are lossless and involutive: node attributes, isolated nodes, and directed all survive, because both orientations carry the same logical graph — reorientation only changes which table the result is indexed by.

Usage

```
## S3 method for class 'agg_df'
nodes(x, ...)

## S3 method for class 'agg_df'
edges(x, ...)

## S3 method for class 'agg_vec'
nodes(x, ...)

## S3 method for class 'agg_vec'
edges(x, ...)

## S3 method for class 'edge_vec'
edges(x, ...)

## S3 method for class 'edge_vec'
nodes(x, ...)

## S3 method for class 'node_vec'
nodes(x, ...)

## S3 method for class 'node_vec'
edges(x, ...)

nodes(x, ...)

edges(x, ...)
```

Arguments

<i>x</i>	A node_vec or edge_vec.
...	Passed on to methods.

Value

nodes() returns a node_vec. edges() returns an edge_vec.

Examples

```
g <- node_vec(
  x = c("A", "B", "C"),
```

nodes.agg_df

25

```
    from = c(1L, 2L),  
    to = c(2L, 3L)  
  )  
edges(g)  
nodes(edges(g))
```

Index

`$.GraphBackend (GraphBackend)`, [12](#)

`agg_df`, [2](#)
`agg_df()`, [4–6](#)
`agg_vec`, [3](#)
`agg_vec()`, [2, 3, 6, 14](#)
`as.igraph`, [5](#)

`c()`, [9, 22](#)

`duplicated()`, [10, 23](#)

`edge_heads`, [6](#)
`edge_incident`, [7](#)
`edge_incident()`, [20](#)
`edge_is_loop`, [8](#)
`edge_is_multi (edge_is_loop)`, [8](#)
`edge_multiplicity (edge_is_loop)`, [8](#)
`edge_tails (edge_heads)`, [6](#)
`edge_vec`, [9](#)
`edge_vec()`, [4, 6](#)
`edges (nodes.agg_df)`, [23](#)
`edges()`, [2, 9, 22](#)

`format()`, [4, 10, 23](#)

`graph_density`, [10](#)
`graph_has_loops (graph_is_directed)`, [11](#)
`graph_is_directed`, [11](#)
`GraphBackend`, [12](#)

`igraph::as.igraph()`, [4](#)
`igraph::igraph()`, [5](#)
`is_aggregated`, [14](#)

`match()`, [10](#)

`n_edges (n_nodes)`, [15](#)
`n_nodes`, [15](#)
`new_edge_vec`, [16](#)
`new_node_vec`, [17](#)

`node_children (node_neighbors)`, [20](#)
`node_degree`, [17](#)
`node_degree()`, [7, 15, 19, 20](#)
`node_incident`, [18](#)
`node_incident()`, [6](#)
`node_is_isolated`, [19](#)
`node_is_leaf (node_is_isolated)`, [19](#)
`node_is_root (node_is_isolated)`, [19](#)
`node_neighbors`, [20](#)
`node_neighbors()`, [7](#)
`node_parents (node_neighbors)`, [20](#)
`node_values`, [21](#)
`node_values()`, [23](#)
`node_vec`, [22](#)
`node_vec()`, [3, 4, 6, 21](#)
`nodes (nodes.agg_df)`, [23](#)
`nodes()`, [2, 4, 5, 9, 15, 22](#)
`nodes.agg_df`, [23](#)

`scale type`, [4, 10, 23](#)

`unique()`, [10, 23](#)