Package: igraph (via r-universe)

October 8, 2026

Title Network Analysis and Visualization

Version 2.3.4

Description Routines for simple graphs and network analysis. It can handle large graphs very well and provides functions for generating random and regular graphs, graph visualization, centrality methods and much more.

License GPL (>= 2)

URL https://r.igraph.org/, https://igraph.org/, https://igraph.discourse.group/

BugReports https://github.com/igraph/rigraph/issues

Depends methods, R (>= 3.5.0)

Imports cli, graphics, grDevices, lifecycle, magrittr, Matrix, pkgconfig (>= 2.0.0), rlang (>= 1.1.0), stats, utils, vctrs

Suggests ape (>= 5.7-0.1), callr, decor, digest, igraphdata, knitr, rgl (>= 1.3.14), rmarkdown, scales, stats4, tcltk, testthat, vdiffr, withr

Enhances graph

LinkingTo cpp11 (>= 0.5.0)

VignetteBuilder knitr

Config/build/compilation-database false

Config/build/never-clean true

Config/comment/compilation-database Generate manually with pkgload:::generate_db() for faster pkgload::load_all()

Config/gha/extra-packages graph=?ignore-before-r=4.4.0

Config/comment/gha/extra-packages graph is an optional Enhances dependency published only on Bioconductor. The Bioconductor releases matching R 4.2 and R 4.3 (graph 3.16 / 3.18) are archived and return HTTP 404, which aborts pak dependency resolution on those jobs. The ?ignore-before-r pak parameter makes pak ignore graph on R older than 4.4.0, where a published release still exists.

Config/Needs/build devtools, irlba, pkgconfig

Config/Needs/coverage covr

Config/Needs/roxygen2 r-lib/roxygen2, igraph/igraph.r2cdocs, moodymudskipper/devtag

Config/Needs/website here, readr, tibble, xmlparsedata, xml2

Config/testthat/edition 3

Config/testthat/parallel true

Config/testthat/start-first aaa-auto, vs-es, scan, vs-operators, weakref, watts.strogatz.game

Encoding UTF-8

SystemRequirements libxml2 (optional), glpk (>= 4.57, optional)

Config/roxygen2/version 8.1.0.9000

NeedsCompilation yes

Author Gábor Csárdi [aut] (ORCID: <https://orcid.org/0000-0001-7098-9676>), Tamás Nepusz [aut] (ORCID: <https://orcid.org/0000-0002-1451-338X>), Vincent Traag [aut] (ORCID: <https://orcid.org/0000-0003-3170-3879>), Szabolcs Horvát [aut] (ORCID: <https://orcid.org/0000-0002-3100-523X>), Fabio Zanini [aut] (ORCID: <https://orcid.org/0000-0001-7097-8539>), Daniel Noom [aut], Kirill Müller [aut, cre] (ORCID: <https://orcid.org/0000-0002-1416-3412>), Michael Antonov [ctb], Chan Zuckerberg Initiative [fnd] (ROR: <https://ror.org/02qenvm24>), David Schoch [aut] (ORCID: <https://orcid.org/0000-0003-2952-4812>), Maëlle Salmon [aut] (ORCID: <https://orcid.org/0000-0002-2815-0399>), R Consortium [fnd] (ROR: <https://ror.org/01z833950>)

Maintainer Kirill Müller <kirill@cynkra.com>

Config/pak/sysreqs libglpk-dev libxml2-dev

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

Date/Publication 2026-09-30 15:40:02 UTC

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

RemoteRef HEAD

RemoteSha 0e650308e406f733d8fac1ed1a91099facd48027

Contents

[.igraph 10
[[.igraph 13
%>% 15
+.igraph 15
add_edges 18

add_layout_ . . . 19
add_vertices . . . 20
adjacent_vertices . . . 21
align_layout . . . 22
all_simple_paths . . . 22
alpha_centrality . . . 24
are_adjacent . . . 26
arpack_defaults . . . 27
articulation_points . . . 31
as.igraph . . . 32
as.matrix.igraph . . . 33
as_adj_list . . . 34
as_adjacency_matrix . . . 36
as_biadjacency_matrix . . . 37
as_data_frame . . . 39
as_directed . . . 41
as_edgelist . . . 43
as_graphnel . . . 44
as_ids . . . 45
as_long_data_frame . . . 46
as_membership . . . 47
assortativity . . . 48
authority_score . . . 50
automorphism_group . . . 51
betweenness . . . 53
bfs . . . 55
biconnected_components . . . 58
bipartite_gnm . . . 59
bipartite_mapping . . . 61
bipartite_projection . . . 62
c.igraph.es . . . 64
c.igraph.vs . . . 65
canonical_permutation . . . 66
categorical_pal . . . 68
centr_betw . . . 69
centr_betw_tmax . . . 70
centr_clo . . . 71
centr_clo_tmax . . . 72
centr_degree . . . 73
centr_degree_tmax . . . 74
centr_eigen . . . 75
centr_eigen_tmax . . . 76
centralize . . . 77
cliques . . . 79
closeness . . . 82
cluster_edge_betweenness . . . 84
cluster_fast_greedy . . . 86
cluster_fluid_communities . . . 88

cluster_infomap . . . 89
cluster_label_prop . . . 91
cluster_leading_eigen . . . 93
cluster_leiden . . . 95
cluster_louvain . . . 98
cluster_optimal . . . 100
cluster_spinglass . . . 101
cluster_walktrap . . . 104
cocitation . . . 106
cohesive_blocks . . . 107
compare . . . 112
complementer . . . 114
component_distribution . . . 115
component_wise . . . 117
compose . . . 118
connect . . . 119
consensus_tree . . . 122
console . . . 123
constraint . . . 124
contract . . . 125
convex_hull . . . 127
coreness . . . 128
count_automorphisms . . . 129
count_isomorphisms . . . 131
count_motifs . . . 132
count_reachable . . . 133
count_subgraph_isomorphisms . . . 134
curve_multiple . . . 136
decompose . . . 137
degree . . . 138
delete_edge_attr . . . 140
delete_edges . . . 140
delete_graph_attr . . . 141
delete_vertex_attr . . . 142
delete_vertices . . . 143
dfs . . . 144
diameter . . . 147
difference . . . 148
difference.igraph . . . 149
difference.igraph.es . . . 150
difference.igraph.vs . . . 151
dim_select . . . 152
disjoint_union . . . 154
distance_table . . . 155
diverging_pal . . . 160
diversity . . . 161
dominator_tree . . . 163
dot-data . . . 164

dyad_census . . . 165
E . . . 166
each_edge . . . 167
eccentricity . . . 168
edge . . . 170
edge_attr . . . 171
edge_attr_names . . . 172
edge_attr<- . . . 172
edge_connectivity . . . 173
edge_density . . . 175
eigen_centrality . . . 177
embed_adjacency_matrix . . . 179
embed_laplacian_matrix . . . 181
ends . . . 183
feedback_arc_set . . . 184
feedback_vertex_set . . . 185
find_cycle . . . 186
fit_hrg . . . 187
fit_power_law . . . 188
get_edge_ids . . . 191
girth . . . 192
global_efficiency . . . 194
gorder . . . 196
graph_ . . . 197
graph_attr . . . 197
graph_attr_names . . . 198
graph_attr<- . . . 199
graph_center . . . 200
graph_from_adj_list . . . 201
graph_from_adjacency_matrix . . . 202
graph_from_atlas . . . 207
graph_from_biadjacency_matrix . . . 208
graph_from_edgelist . . . 209
graph_from_graphdb . . . 210
graph_from_graphnel . . . 212
graph_from_isomorphism_class . . . 213
graph_from_lcf . . . 214
graph_from_literal . . . 215
graph_id . . . 217
graph_version . . . 218
graphlet_basis . . . 219
greedy_vertex_coloring . . . 221
groups . . . 222
gsize . . . 223
harmonic_centrality . . . 224
has_eulerian_path . . . 225
head_of . . . 226
head_print . . . 227

hits_scores . . . 228
hrg . . . 229
hrg-methods . . . 230
hrg_tree . . . 231
identical_graphs . . . 231
igraph-attribute-combination . . . 232
igraph-dollar . . . 234
igraph-es-attributes . . . 235
igraph-es-indexing . . . 236
igraph-es-indexing2 . . . 238
igraph-minus . . . 240
igraph-vs-attributes . . . 241
igraph-vs-indexing . . . 243
igraph-vs-indexing2 . . . 245
igraph_options . . . 246
incident . . . 248
incident_edges . . . 249
indent_print . . . 250
intersection . . . 251
intersection.igraph . . . 251
intersection.igraph.es . . . 253
intersection.igraph.vs . . . 254
invalidate_cache . . . 255
is_acyclic . . . 256
is_biconnected . . . 257
is_bipartite . . . 258
is_chordal . . . 258
is_complete . . . 260
is_dag . . . 261
is_degseq . . . 262
is_directed . . . 263
is_forest . . . 264
is_graphical . . . 265
is_igraph . . . 266
is_matching . . . 267
is_min_separator . . . 269
is_named . . . 270
is_printer_callback . . . 271
is_separator . . . 272
is_tree . . . 273
is_weighted . . . 274
isomorphic . . . 275
isomorphism_class . . . 277
isomorphisms . . . 278
ivs . . . 279
k_shortest_paths . . . 281
keeping_degseq . . . 282
knn . . . 283

laplacian_matrix . . . 285
layout_ . . . 287
layout_as_bipartite . . . 289
layout_as_star . . . 290
layout_as_tree . . . 292
layout_in_circle . . . 294
layout_modifier . . . 295
layout_nicely . . . 296
layout_on_grid . . . 297
layout_on_sphere . . . 299
layout_randomly . . . 300
layout_with_dh . . . 301
layout_with_drl . . . 303
layout_with_fr . . . 306
layout_with_gem . . . 308
layout_with_graphopt . . . 310
layout_with_kk . . . 311
layout_with_lgl . . . 314
layout_with_mds . . . 315
layout_with_sugiyama . . . 317
local_scan . . . 321
make_ . . . 323
make_bipartite_graph . . . 325
make_chordal_ring . . . 326
make_circulant . . . 327
make_clusters . . . 328
make_de_bruijn_graph . . . 329
make_empty_graph . . . 330
make_from_prufer . . . 331
make_full_bipartite_graph . . . 332
make_full_citation_graph . . . 333
make_full_graph . . . 334
make_full_multipartite . . . 335
make_graph . . . 336
make_kautz_graph . . . 339
make_lattice . . . 340
make_line_graph . . . 342
make_ring . . . 343
make_star . . . 344
make_tree . . . 345
make_turan . . . 346
make_wheel . . . 347
match_vertices . . . 348
max_cardinality . . . 349
max_flow . . . 351
membership . . . 352
merge_coords . . . 356
min_cut . . . 358

min_separators . . . 360
min_st_separators . . . 361
modularity.igraph . . . 363
motifs . . . 365
mst . . . 366
neighbors . . . 368
norm_coords . . . 369
normalize . . . 370
page_rank . . . 371
path . . . 373
permute . . . 374
plot.common . . . 375
plot.igraph . . . 382
plot.sir . . . 384
plot_dendrogram . . . 386
plot_dendrogram.igraphHRG . . . 388
power_centrality . . . 390
predict_edges . . . 392
print.igraph . . . 394
print.igraph.es . . . 396
print.igraph.vs . . . 397
print.igraphHRG . . . 399
print.igraphHRGConsensus . . . 400
printer_callback . . . 401
r_pal . . . 402
radius . . . 402
random_walk . . . 403
read_graph . . . 405
realize_bipartite_degseq . . . 408
realize_degseq . . . 409
reciprocity . . . 411
rep.igraph . . . 412
rev.igraph.es . . . 413
rev.igraph.vs . . . 414
reverse_edges . . . 415
rewire . . . 416
rglplot . . . 416
running_mean . . . 417
sample_ . . . 418
sample_bipartite . . . 419
sample_chung_lu . . . 421
sample_correlated_gnp . . . 424
sample_correlated_gnp_pair . . . 426
sample_degseq . . . 427
sample_dirichlet . . . 431
sample_dot_product . . . 432
sample_fitness . . . 433
sample_fitness_pl . . . 435

sample_forestfire . . . 437
sample_gnm . . . 439
sample_gnp . . . 440
sample_grg . . . 441
sample_growing . . . 443
sample_hierarchical_sbm . . . 444
sample_hrg . . . 445
sample_islands . . . 446
sample_k_regular . . . 447
sample_last_cit . . . 448
sample_motifs . . . 450
sample_pa . . . 451
sample_pa_age . . . 454
sample_pref . . . 457
sample_sbm . . . 459
sample_seq . . . 461
sample_smallworld . . . 462
sample_spanning_tree . . . 463
sample_sphere_surface . . . 464
sample_sphere_volume . . . 465
sample_traits_callaway . . . 466
sample_tree . . . 468
scan_stat . . . 469
sequential_pal . . . 471
set_edge_attr . . . 472
set_graph_attr . . . 473
set_vertex_attr . . . 473
set_vertex_attrs . . . 474
shapes . . . 475
similarity . . . 478
simple_cycles . . . 480
simplified . . . 481
simplify . . . 482
spectrum . . . 483
split_join_distance . . . 485
st_cuts . . . 486
st_min_cuts . . . 488
stochastic_matrix . . . 489
strength . . . 490
subcomponent . . . 492
subgraph . . . 493
subgraph_centrality . . . 494
subgraph_isomorphic . . . 496
subgraph_isomorphisms . . . 498
tail_of . . . 500
time_bins . . . 500
tkplot . . . 503
to_prufer . . . 506

topo_sort . . . 507
transitive_closure . . . 508
transitivity . . . 509
triad_census . . . 511
triangles . . . 513
unfold_tree . . . 514
union . . . 515
union.igraph . . . 516
union.igraph.es . . . 517
union.igraph.vs . . . 518
unique.igraph.es . . . 519
unique.igraph.vs . . . 520
upgrade_graph . . . 520
V . . . 521
vertex . . . 523
vertex.shape.pie . . . 524
vertex_attr . . . 525
vertex_attr_names . . . 526
vertex_attr<- . . . 526
vertex_connectivity . . . 527
voronoi_cells . . . 529
weighted_cliques . . . 531
which_multiple . . . 532
which_mutual . . . 534
with_edge_ . . . 535
with_graph_ . . . 536
with_igraph_opt . . . 537
with_vertex_ . . . 537
without_attr . . . 538
without_loops . . . 539
without_multiples . . . 539
write_graph . . . 540

Index **543**

`[.igraph` *Query and manipulate a graph as it were an adjacency matrix*

Description

Query and manipulate a graph as it were an adjacency matrix

Usage

```
## S3 method for class 'igraph'

  x[
  i,
  j,
  ...,
  from,
  to,
  sparse = igraph_opt("sparsematrices"),
  edges = FALSE,
  drop = TRUE,
  attr = if (is_weighted(x)) "weight"
]
```

Arguments

x	The graph.
i	Index. Vertex ids or names or logical vectors. See details below.
j	Index. Vertex ids or names or logical vectors. See details below.
...	Currently ignored.
from	A numeric or character vector giving vertex ids or names. Together with the to argument, it can be used to query/set a sequence of edges. See details below. This argument cannot be present together with any of the i and j arguments and if it is present, then the to argument must be present as well.
to	A numeric or character vector giving vertex ids or names. Together with the from argument, it can be used to query/set a sequence of edges. See details below. This argument cannot be present together with any of the i and j arguments and if it is present, then the from argument must be present as well.
sparse	Logical scalar, whether to return sparse matrices.
edges	Logical scalar, whether to return edge ids.
drop	Ignored.
attr	If not NULL, then it should be the name of an edge attribute. This attribute is queried and returned.

Details

The single bracket indexes the (possibly weighted) adjacency matrix of the graph. Here is what you can do with it:

1. Check whether there is an edge between two vertices (v and w) in the graph:

   ```
   graph[v, w]
   ```

   A numeric scalar is returned, one if the edge exists, zero otherwise.

2. Extract the (sparse) adjacency matrix of the graph, or part of it:

```
  graph[]
graph[1:3,5:6]
graph[c(1,3,5),]
```

The first variants returns the full adjacency matrix, the other two return part of it.

3. The `from` and `to` arguments can be used to check the existence of many edges. In this case, both `from` and `to` must be present and they must have the same length. They must contain vertex ids or names. A numeric vector is returned, of the same length as `from` and `to`, it contains ones for existing edges edges and zeros for non-existing ones. Example:

```
  graph[from=1:3, to=c(2,3,5)]
```

.

4. For weighted graphs, the `[` operator returns the edge weights. For non-esistent edges zero weights are returned. Other edge attributes can be queried as well, by giving the `attr` argument.

5. Querying edge ids instead of the existance of edges or edge attributes. E.g.

```
  graph[1, 2, edges=TRUE]
```

returns the id of the edge between vertices 1 and 2, or zero if there is no such edge.

6. Adding one or more edges to a graph. For this the element(s) of the imaginary adjacency matrix must be set to a non-zero numeric value (or `TRUE`):

```
  graph[1, 2] <- 1
graph[1:3,1] <- 1
graph[from=1:3, to=c(2,3,5)] <- TRUE
```

This does not affect edges that are already present in the graph, i.e. no multiple edges are created.

7. Adding weighted edges to a graph. The `attr` argument contains the name of the edge attribute to set, so it does not have to be 'weight':

```
  graph[1, 2, attr="weight"]<- 5
graph[from=1:3, to=c(2,3,5)] <- c(1,-1,4)
```

If an edge is already present in the network, then only its weights or other attribute are updated. If the graph is already weighted, then the `attr="weight"` setting is implicit, and one does not need to give it explicitly.

8. Deleting edges. The replacement syntax allow the deletion of edges, by specifying `FALSE` or `NULL` as the replacement value:

```
  graph[v, w] <- FALSE
```

removes the edge from vertex v to vertex w. As this can be used to delete edges between two sets of vertices, either pairwise:

```
  graph[from=v, to=w] <- FALSE
```

or not:

```
  graph[v, w] <- FALSE
```

if v and w are vectors of edge ids or names.

'[' allows logical indices and negative indices as well, with the usual R semantics. E.g.

```
graph[degree(graph)==0, 1] <- 1
```

adds an edge from every isolate vertex to vertex one, and

```
G <- make_empty_graph(10)
G[-1,1] <- TRUE
```

creates a star graph.

Of course, the indexing operators support vertex names, so instead of a numeric vertex id a vertex can also be given to '[' and '[['.

Value

A scalar or matrix. See details below.

See Also

Other structural queries: [[.igraph()](), [adjacent_vertices()](), [are_adjacent()](), [ends()](), [get_edge_ids()](), [gorder()](), [gsize()](), [head_of()](), [incident()](), [incident_edges()](), [is_directed()](), [neighbors()](), [tail_of()]()

[[.igraph *Query and manipulate a graph as it were an adjacency list*

Description

Query and manipulate a graph as it were an adjacency list

Usage

```
## S3 method for class 'igraph'
x[[i, j, from, to, ..., directed = TRUE, edges = FALSE, exact = TRUE]]
```

Arguments

x	The graph.
i	Index, integer, character or logical, see details below.
j	Index, integer, character or logical, see details below.
from	A numeric or character vector giving vertex ids or names. Together with the to argument, it can be used to query/set a sequence of edges. See details below. This argument cannot be present together with any of the i and j arguments and if it is present, then the to argument must be present as well.

to	A numeric or character vector giving vertex ids or names. Together with the from argument, it can be used to query/set a sequence of edges. See details below. This argument cannot be present together with any of the i and j arguments and if it is present, then the from argument must be present as well.
...	Additional arguments are not used currently.
directed	Logical scalar, whether to consider edge directions in directed graphs. It is ignored for undirected graphs.
edges	Logical scalar, whether to return edge ids.
exact	Ignored.

Details

The double bracket operator indexes the (imaginary) adjacency list of the graph. This can used for the following operations:

1. Querying the adjacent vertices for one or more vertices:

```
  graph[[1:3,]]
graph[[,1:3]]
```

   The first form gives the successors, the second the predecessors or the 1:3 vertices. (For undirected graphs they are equivalent.)

2. Querying the incident edges for one or more vertices, if the edges argument is set to TRUE:

```
  graph[[1:3, , edges=TRUE]]
graph[[, 1:3, edges=TRUE]]
```

3. Querying the edge ids between two sets or vertices, if both indices are used. E.g.

```
  graph[[v, w, edges=TRUE]]
```

   gives the edge ids of all the edges that exist from vertices v to vertices w.

The alternative argument names from and to can be used instead of the usual i and j, to make the code more readable:

```
 graph[[from = 1:3]]
graph[[from = v, to = w, edges = TRUE]]
```

'[[' operators allows logical indices and negative indices as well, with the usual R semantics.

Vertex names are also supported, so instead of a numeric vertex id a vertex can also be given to '[' and '[['.

See Also

Other structural queries: [.igraph(), adjacent_vertices(), are_adjacent(), ends(), get_edge_ids(), gorder(), gsize(), head_of(), incident(), incident_edges(), is_directed(), neighbors(), tail_of()

%>% *Magrittr's pipes*

Description

igraph re-exports the %>% operator of magrittr, because we find it very useful. Please see the documentation in the magrittr package.

Usage

```
lhs %>% rhs
```

Arguments

lhs	Left hand side of the pipe.
rhs	Right hand side of the pipe.

Value

Result of applying the right hand side to the result of the left hand side.

Examples

```
make_ring(10) %>%
  add_edges(c(1, 6)) %>%
  plot()
```

+.igraph *Add vertices, edges or another graph to a graph*

Description

Add vertices, edges or another graph to a graph

Usage

```
## S3 method for class 'igraph'
e1 + e2
```

Arguments

e1	First argument, probably an igraph graph, but see details below.
e2	Second argument, see details below.

Details

The plus operator can be used to add vertices or edges to graph. The actual operation that is performed depends on the type of the right hand side argument.

- If is is another igraph graph object and they are both named graphs, then the union of the two graphs are calculated, see `union()`.
- If it is another igraph graph object, but either of the two are not named, then the disjoint union of the two graphs is calculated, see `disjoint_union()`.
- If it is a numeric scalar, then the specified number of vertices are added to the graph.
- If it is a character scalar or vector, then it is interpreted as the names of the vertices to add to the graph.
- If it is an object created with the `vertex()` or `vertices()` function, then new vertices are added to the graph. This form is appropriate when one wants to add some vertex attributes as well. The operands of the `vertices()` function specifies the number of vertices to add and their attributes as well.

  The unnamed arguments of `vertices()` are concatenated and used as the 'name' vertex attribute (i.e. vertex names), the named arguments will be added as additional vertex attributes. Examples:

  ```
  g <- g +
       vertex(shape="circle", color= "red")
  g <- g + vertex("foo", color="blue")
  g <- g + vertex("bar", "foobar")
  g <- g + vertices("bar2", "foobar2", color=1:2, shape="rectangle")
  ```

  `vertex()` is just an alias to `vertices()`, and it is provided for readability. The user should use it if a single vertex is added to the graph.
- If it is an object created with the `edge()` or `edges()` function, then new edges will be added to the graph. The new edges and possibly their attributes can be specified as the arguments of the `edges()` function.

  The unnamed arguments of `edges()` are concatenated and used as vertex ids of the end points of the new edges. The named arguments will be added as edge attributes.

  Examples:

  ```
  g <- make_empty_graph() +
         vertices(letters[1:10]) +
         vertices("foo", "bar", "bar2", "foobar2")
  g <- g + edge("a", "b")
  g <- g + edges("foo", "bar", "bar2", "foobar2")
  g <- g + edges(c("bar", "foo", "foobar2", "bar2"), color="red", weight=1:2)
  ```

  See more examples below.

  `edge()` is just an alias to `edges()` and it is provided for readability. The user should use it if a single edge is added to the graph.
- If it is an object created with the `path()` function, then new edges that form a path are added. The edges and possibly their attributes are specified as the arguments to the `path()` function. The non-named arguments are concatenated and interpreted as the vertex ids along the path. The remaining arguments are added as edge attributes.

  Examples:

```
g <- make_empty_graph() + vertices(letters[1:10])
g <- g + path("a", "b", "c", "d")
g <- g + path("e", "f", "g", weight=1:2, color="red")
g <- g + path(c("f", "c", "j", "d"), width=1:3, color="green")
```

It is important to note that, although the plus operator is commutative, i.e. is possible to write

```
graph <- "foo" + make_empty_graph()
```

it is not associative, e.g.

```
graph <- "foo" + "bar" + make_empty_graph()
```

results a syntax error, unless parentheses are used:

```
graph <- "foo" + ( "bar" + make_empty_graph() )
```

For clarity, we suggest to always put the graph object on the left hand side of the operator:

```
graph <- make_empty_graph() + "foo" + "bar"
```

See Also

Other functions for manipulating graph structure: add_edges(), add_vertices(), complementer(), compose(), connect(), contract(), delete_edges(), delete_vertices(), difference(), difference.igraph(), disjoint_union(), edge(), igraph-minus, intersection(), intersection.igraph(), path(), permute(), rep.igraph(), reverse_edges(), simplify(), transitive_closure(), union(), union.igraph(), vertex()

Examples

```
# 10 vertices named a,b,c,... and no edges
g <- make_empty_graph() + vertices(letters[1:10])

# Add edges to make it a ring
g <- g + path(letters[1:10], letters[1], color = "grey")

# Add some extra random edges
g <- g + edges(sample(V(g), 10, replace = TRUE), color = "red")
g$layout <- layout_in_circle
plot(g)
```

add_edges *Add edges to a graph*

Description

The new edges are given as a vertex sequence, e.g. internal numeric vertex ids, or vertex names. The first edge points from `edges[1]` to `edges[2]`, the second from `edges[3]` to `edges[4]`, etc.

Usage

```
add_edges(graph, edges, ..., attr = list())
```

Arguments

`graph`	The input graph
`edges`	The edges to add, a vertex sequence with even number of vertices.
`...`	Additional arguments, they must be named, and they will be added as edge attributes, for the newly added edges. See also details below.
`attr`	A named list, its elements will be added as edge attributes, for the newly added edges. See also details below.

Details

If attributes are supplied, and they are not present in the graph, their values for the original edges of the graph are set to `NA`.

Value

The graph, with the edges (and attributes) added.

Related documentation in the C library

`edges()`, `vcount()`, `get_eids()`, `ecount()`

See Also

Other functions for manipulating graph structure: `+.igraph()`, `add_vertices()`, `complementer()`, `compose()`, `connect()`, `contract()`, `delete_edges()`, `delete_vertices()`, `difference()`, `difference.igraph()`, `disjoint_union()`, `edge()`, `igraph-minus`, `intersection()`, `intersection.igraph()`, `path()`, `permute()`, `rep.igraph()`, `reverse_edges()`, `simplify()`, `transitive_closure()`, `union()`, `union.igraph()`, `vertex()`

Examples

```
g <- make_empty_graph(n = 5) %>%
  add_edges(c(
    1, 2,
    2, 3,
    3, 4,
    4, 5
  )) %>%
  set_edge_attr("color", value = "red") %>%
  add_edges(c(5, 1), color = "green")
E(g)[[]]
plot(g)
```

---

add_layout_ *Add layout to graph*

Description

Add layout to graph

Usage

```
add_layout_(graph, ..., overwrite = TRUE)
```

Arguments

graph	The input graph.
...	Additional arguments are passed to layout_().
overwrite	Whether to overwrite the layout of the graph, if it already has one.

Value

The input graph, with the layout added.

See Also

layout_() for a description of the layout API.

Other graph layouts: component_wise(), layout_(), layout_as_bipartite(), layout_as_star(), layout_as_tree(), layout_in_circle(), layout_nicely(), layout_on_grid(), layout_on_sphere(), layout_randomly(), layout_with_dh(), layout_with_fr(), layout_with_gem(), layout_with_graphopt(), layout_with_kk(), layout_with_lgl(), layout_with_mds(), layout_with_sugiyama(), merge_coords(), norm_coords(), normalize()

Examples

```
(make_star(11) + make_star(11)) %>%
  add_layout_(as_star(), component_wise()) %>%
  plot()
```

add_vertices *Add vertices to a graph*

Description

If attributes are supplied, and they are not present in the graph, their values for the original vertices of the graph are set to NA.

Usage

```
add_vertices(graph, nv, ..., attr = list())
```

Arguments

graph	The input graph.
nv	The number of vertices to add.
...	Additional arguments, they must be named, and they will be added as vertex attributes, for the newly added vertices. See also details below.
attr	A named list, its elements will be added as vertex attributes, for the newly added vertices. See also details below.

Value

The graph, with the vertices (and attributes) added.

Related documentation in the C library

add_vertices(), vcount()

See Also

Other functions for manipulating graph structure: +.igraph(), add_edges(), complementer(), compose(), connect(), contract(), delete_edges(), delete_vertices(), difference(), difference.igraph(), disjoint_union(), edge(), igraph-minus, intersection(), intersection.igraph(), path(), permute(), rep.igraph(), reverse_edges(), simplify(), transitive_closure(), union(), union.igraph(), vertex()

Examples

```
g <- make_empty_graph() %>%
  add_vertices(3, color = "red") %>%
  add_vertices(2, color = "green") %>%
  add_edges(c(
    1, 2,
    2, 3,
    3, 4,
    4, 5
  ))
```

```
g
V(g)[[]]
plot(g)
```

`adjacent_vertices` *Adjacent vertices of multiple vertices in a graph*

Description

This function is similar to `neighbors()`, but it queries the adjacent vertices for multiple vertices at once.

Usage

```
adjacent_vertices(graph, v, mode = c("out", "in", "all", "total"))
```

Arguments

`graph`	Input graph.
`v`	The vertices to query.
`mode`	Whether to query outgoing ('out'), incoming ('in') edges, or both types ('all'). This is ignored for undirected graphs.

Value

A list of vertex sequences.

Related documentation in the C library

`vcount()`

See Also

Other structural queries: `[.igraph()`, `[[.igraph()`, `are_adjacent()`, `ends()`, `get_edge_ids()`, `gorder()`, `gsize()`, `head_of()`, `incident()`, `incident_edges()`, `is_directed()`, `neighbors()`, `tail_of()`

Examples

```
g <- make_graph("Zachary")
adjacent_vertices(g, c(1, 34))
```

`align_layout` *Align a vertex layout This function centers a vertex layout on the coordinate system origin and rotates the layout to achieve a visually pleasing alignment with the coordinate axes. Doing this is particularly useful with force-directed layouts such as* `layout_with_fr()`.

Description

Align a vertex layout This function centers a vertex layout on the coordinate system origin and rotates the layout to achieve a visually pleasing alignment with the coordinate axes. Doing this is particularly useful with force-directed layouts such as `layout_with_fr()`.

Usage

```
align_layout(graph, layout)
```

Arguments

`graph`	The graph whose layout is to be aligned.
`layout`	A matrix whose rows are the coordinates of vertices.

Value

modified layout matrix

Related documentation in the C library

`layout_align()`

Examples

```
g <- make_lattice(c(3, 3))
l1 <- layout_with_fr(g)
l2 <- align_layout(g,l1)
plot(g, layout = l1)
plot(g, layout = l2)
```

`all_simple_paths` *List all simple paths from one source*

Description

This function lists all simple paths from one source vertex to another vertex or vertices. A path is simple if contains no repeated vertices.

Usage

```
all_simple_paths(
  graph,
  from,
  to = V(graph),
  mode = c("out", "in", "all", "total"),
  cutoff = -1
)
```

Arguments

graph	The input graph.
from	The source vertex.
to	The target vertex of vertices. Defaults to all vertices.
mode	Character constant, gives whether the shortest paths to or from the given vertices should be calculated for directed graphs. If out then the shortest paths *from* the vertex, if in then *to* it will be considered. If all, the default, then the corresponding undirected graph will be used, i.e. not directed paths are searched. This argument is ignored for undirected graphs.
cutoff	Maximum length of the paths that are considered. If negative, no cutoff is used.

Details

Note that potentially there are exponentially many paths between two vertices of a graph, and you may run out of memory when using this function, if your graph is lattice-like.

This function ignores multiple and loop edges.

Value

A list of integer vectors, each integer vector is a path from the source vertex to one of the target vertices. A path is given by its vertex ids.

Related documentation in the C library

get_all_simple_paths(), vcount()

See Also

Other paths: diameter(), distance_table(), eccentricity(), graph_center(), radius()

Examples

```
g <- make_ring(10)
all_simple_paths(g, 1, 5)
all_simple_paths(g, 1, c(3, 5))
```

`alpha_centrality` *Find Bonacich alpha centrality scores of network positions*

Description

`alpha_centrality()` calculates the alpha centrality of some (or all) vertices in a graph.

Usage

```
alpha_centrality(
  graph,
  nodes = V(graph),
  alpha = 1,
  loops = FALSE,
  exo = 1,
  weights = NULL,
  tol = 1e-07,
  sparse = TRUE
)
```

Arguments

`graph`	The input graph, can be directed or undirected. In undirected graphs, edges are treated as if they were reciprocal directed ones.
`nodes`	Vertex sequence, the vertices for which the alpha centrality values are returned. (For technical reasons they will be calculated for all vertices, anyway.)
`alpha`	Parameter specifying the relative importance of endogenous versus exogenous factors in the determination of centrality. See details below.
`loops`	Whether to eliminate loop edges from the graph before the calculation.
`exo`	The exogenous factors, in most cases this is either a constant – the same factor for every node, or a vector giving the factor for every vertex. Note that too long vectors will be truncated and too short vectors will be replicated to match the number of vertices.
`weights`	A character scalar that gives the name of the edge attribute to use in the adjacency matrix. If it is `NULL`, then the 'weight' edge attribute of the graph is used, if there is one. Otherwise, or if it is `NA`, then the calculation uses the standard adjacency matrix.
`tol`	Tolerance for near-singularities during matrix inversion, see `solve()`.
`sparse`	Logical scalar, whether to use sparse matrices for the calculation. The 'Matrix' package is required for sparse matrix support

Details

The alpha centrality measure can be considered as a generalization of eigenvector centrality to directed graphs. It was proposed by Bonacich in 2001 (see reference below).

The alpha centrality of the vertices in a graph is defined as the solution of the following matrix equation:

$$x = \alpha A^T x + e,$$

where A is the (not necessarily symmetric) adjacency matrix of the graph, e is the vector of exogenous sources of status of the vertices and α is the relative importance of the endogenous versus exogenous factors.

Value

A numeric vector contaning the centrality scores for the selected vertices.

Warning

Singular adjacency matrices cause problems for this algorithm, the routine may fail is certain cases.

Related documentation in the C library

vcount(), simplify(), get_adjacency(), get_adjacency_sparse(), edges(), get_eids(), ecount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Bonacich, P. and Lloyd, P. (2001). "Eigenvector-like measures of centrality for asymmetric relations" *Social Networks*, 23, 191-201.

See Also

eigen_centrality() and power_centrality()

Centrality measures: authority_score(), betweenness(), closeness(), diversity(), eigen_centrality(), harmonic_centrality(), hits_scores(), page_rank(), power_centrality(), spectrum(), strength(), subgraph_centrality()

Examples

```
# The examples from Bonacich's paper
g.1 <- make_graph(c(1, 3, 2, 3, 3, 4, 4, 5))
g.2 <- make_graph(c(2, 1, 3, 1, 4, 1, 5, 1))
g.3 <- make_graph(c(1, 2, 2, 3, 3, 4, 4, 1, 5, 1))
alpha_centrality(g.1)
alpha_centrality(g.2)
alpha_centrality(g.3, alpha = 0.5)
```

`are_adjacent` *Are two vertices adjacent?*

Description

The order of the vertices only matters in directed graphs, where the existence of a directed `(v1, v2)` edge is queried.

Usage

```
are_adjacent(graph, v1, v2)
```

Arguments

`graph`	The graph.
`v1`	The first vertex, tail in directed graphs.
`v2`	The second vertex, head in directed graphs.

Value

A logical scalar, `TRUE` if edge `(v1, v2)` exists in the graph.

Related documentation in the C library

`are_adjacent()`, `vcount()`

See Also

Other structural queries: `[.igraph()`, `[[.igraph()`, `adjacent_vertices()`, `ends()`, `get_edge_ids()`, `gorder()`, `gsize()`, `head_of()`, `incident()`, `incident_edges()`, `is_directed()`, `neighbors()`, `tail_of()`

Examples

```
ug <- make_ring(10)
ug
are_adjacent(ug, 1, 2)
are_adjacent(ug, 2, 1)

dg <- make_ring(10, directed = TRUE)
dg
are_adjacent(ug, 1, 2)
are_adjacent(ug, 2, 1)
```

arpack_defaults *ARPACK eigenvector calculation*

Description

Interface to the ARPACK library for calculating eigenvectors of sparse matrices

Usage

```
arpack_defaults()

arpack(
  func,
  extra = NULL,
  sym = FALSE,
  options = arpack_defaults(),
  env = parent.frame(),
  complex = !sym
)
```

Arguments

func	The function to perform the matrix-vector multiplication. ARPACK requires to perform these by the user. The function gets the vector x as the first argument, and it should return Ax, where A is the "input matrix". (The input matrix is never given explicitly.) The second argument is extra.
extra	Extra argument to supply to func.
sym	Logical scalar, whether the input matrix is symmetric. Always supply TRUE here if it is, since it can speed up the computation.
options	Options to ARPACK, a named list to overwrite some of the default option values. See details below.
env	The environment in which func will be evaluated.
complex	Whether to convert the eigenvectors returned by ARPACK into R complex vectors. By default this is not done for symmetric problems (these only have real eigenvectors/values), but only non-symmetric ones. If you have a non-symmetric problem, but you're sure that the results will be real, then supply FALSE here.

Details

ARPACK is a library for solving large scale eigenvalue problems. The package is designed to compute a few eigenvalues and corresponding eigenvectors of a general n by n matrix A. It is most appropriate for large sparse or structured matrices A where structured means that a matrix-vector product w <- Av requires order n rather than the usual order n^2 floating point operations.

This function is an interface to ARPACK. igraph does not contain all ARPACK routines, only the ones dealing with symmetric and non-symmetric eigenvalue problems using double precision real numbers.

The eigenvalue calculation in ARPACK (in the simplest case) involves the calculation of the Av product where A is the matrix we work with and v is an arbitrary vector. The function supplied in the `fun` argument is expected to perform this product. If the product can be done efficiently, e.g. if the matrix is sparse, then `arpack()` is usually able to calculate the eigenvalues very quickly.

The `options` argument specifies what kind of calculation to perform. It is a list with the following members, they correspond directly to ARPACK parameters. On input it has the following fields:

bmat Character constant, possible values: '`I`', standard eigenvalue problem, $Ax = \lambda x$; and '`G`', generalized eigenvalue problem, $Ax = \lambda Bx$. Currently only '`I`' is supported.

n Numeric scalar. The dimension of the eigenproblem. You only need to set this if you call `arpack()` directly. (I.e. not needed for `eigen_centrality()`, `page_rank()`, etc.)

which Specify which eigenvalues/vectors to compute, character constant with exactly two characters. Possible values for symmetric input matrices:

"LA" Compute `nev` largest (algebraic) eigenvalues.

"SA" Compute `nev` smallest (algebraic) eigenvalues.

"LM" Compute `nev` largest (in magnitude) eigenvalues.

"SM" Compute `nev` smallest (in magnitude) eigenvalues.

"BE" Compute `nev` eigenvalues, half from each end of the spectrum. When `nev` is odd, compute one more from the high end than from the low end.

Possible values for non-symmetric input matrices:

"LM" Compute `nev` eigenvalues of largest magnitude.

"SM" Compute `nev` eigenvalues of smallest magnitude.

"LR" Compute `nev` eigenvalues of largest real part.

"SR" Compute `nev` eigenvalues of smallest real part.

"LI" Compute `nev` eigenvalues of largest imaginary part.

"SI" Compute `nev` eigenvalues of smallest imaginary part.

This parameter is sometimes overwritten by the various functions, e.g. `page_rank()` always sets '`LM`'.

nev Numeric scalar. The number of eigenvalues to be computed.

tol Numeric scalar. Stopping criterion: the relative accuracy of the Ritz value is considered acceptable if its error is less than `tol` times its estimated value. If this is set to zero then machine precision is used.

ncv Number of Lanczos vectors to be generated.

ldv Numberic scalar. It should be set to zero in the current implementation.

ishift Either zero or one. If zero then the shifts are provided by the user via reverse communication. If one then exact shifts with respect to the reduced tridiagonal matrix T. Please always set this to one.

maxiter Maximum number of Arnoldi update iterations allowed.

nb Blocksize to be used in the recurrence. Please always leave this on the default value, one.

mode The type of the eigenproblem to be solved. Possible values if the input matrix is symmetric:

1 $Ax = \lambda x$, A is symmetric.
2 $Ax = \lambda Mx$, A is symmetric, M is symmetric positive definite.
3 $Kx = \lambda Mx$, K is symmetric, M is symmetric positive semi-definite.
4 $Kx = \lambda KGx$, K is symmetric positive semi-definite, KG is symmetric indefinite.
5 $Ax = \lambda Mx$, A is symmetric, M is symmetric positive semi-definite. (Cayley transformed mode.)

Please note that only `mode==1` was tested and other values might not work properly. Possible values if the input matrix is not symmetric:

1 $Ax = \lambda x$.
2 $Ax = \lambda Mx$, M is symmetric positive definite.
3 $Ax = \lambda Mx$, M is symmetric semi-definite.
4 $Ax = \lambda Mx$, M is symmetric semi-definite.

Please note that only `mode==1` was tested and other values might not work properly.

start Not used currently. Later it be used to set a starting vector.

sigma Not used currently.

sigmai Not use currently.

: On output the following additional fields are added:

info Error flag of ARPACK. Possible values:

0 Normal exit.
1 Maximum number of iterations taken.
3 No shifts could be applied during a cycle of the implicitly restarted Arnoldi iteration. One possibility is to increase the size of `ncv` relative to `nev`.

ARPACK can return more error conditions than these, but they are converted to regular igraph errors.

iter Number of Arnoldi iterations taken.

nconv Number of "converged" Ritz values. This represents the number of Ritz values that satisfy the convergence critetion.

numop Total number of matrix-vector multiplications.

numopb Not used currently.

numreo Total number of steps of re-orthogonalization.

Please see the ARPACK documentation for additional details.

Value

A named list with the following members:

values Numeric vector, the desired eigenvalues.

vectors Numeric matrix, the desired eigenvectors as columns. If `complex=TRUE` (the default for non-symmetric problems), then the matrix is complex.

options A named list with the supplied `options` and some information about the performed calculation, including an ARPACK exit code. See the details above.

Author(s)

Rich Lehoucq, Kristi Maschhoff, Danny Sorensen, Chao Yang for ARPACK, Gabor Csardi <csardi.gabor@gmail.com> for the R interface.

References

D.C. Sorensen, Implicit Application of Polynomial Filters in a k-Step Arnoldi Method. *SIAM J. Matr. Anal. Apps.*, 13 (1992), pp 357-385.

R.B. Lehoucq, Analysis and Implementation of an Implicitly Restarted Arnoldi Iteration. *Rice University Technical Report* TR95-13, Department of Computational and Applied Mathematics.

B.N. Parlett & Y. Saad, Complex Shift and Invert Strategies for Real Matrices. *Linear Algebra and its Applications*, vol 88/89, pp 575-595, (1987).

See Also

eigen_centrality(), page_rank(), hub_score(), cluster_leading_eigen() are some of the functions in igraph that use ARPACK.

Examples

```
# Identity matrix
f <- function(x, extra = NULL) x
arpack(f, options = list(n = 10, nev = 2, ncv = 4), sym = TRUE)

# Graph laplacian of a star graph (undirected), n>=2
# Note that this is a linear operation
f <- function(x, extra = NULL) {
  y <- x
  y[1] <- (length(x) - 1) * x[1] - sum(x[-1])
  for (i in 2:length(x)) {
    y[i] <- x[i] - x[1]
  }
  y
}

arpack(f, options = list(n = 10, nev = 1, ncv = 3), sym = TRUE)

# double check
eigen(laplacian_matrix(make_star(10, mode = "undirected")))

## First three eigenvalues of the adjacency matrix of a graph
## We need the 'Matrix' package for this

library("Matrix")
set.seed(42)
g <- sample_gnp(1000, 5 / 1000)
M <- as_adjacency_matrix(g, sparse = TRUE)
f2 <- function(x, extra = NULL) {
  cat(".")
  as.vector(M %*% x)
```

```
}
baev <- arpack(
  f2,
  sym = TRUE,
  options = list(
    n = vcount(g),
    nev = 3,
    ncv = 8,
    which = "LM",
    maxiter = 2000
  )
)
```

`articulation_points` *Articulation points and bridges of a graph*

Description

`articulation_points()` finds the articulation points (or cut vertices)

Usage

```
articulation_points(graph)

bridges(graph)
```

Arguments

`graph`	The input graph. It is treated as an undirected graph, even if it is directed.

Details

Articulation points or cut vertices are vertices whose removal increases the number of connected components in a graph. Similarly, bridges or cut-edges are edges whose removal increases the number of connected components in a graph. If the original graph was connected, then the removal of a single articulation point or a single bridge makes it disconnected. If a graph contains no articulation points, then its vertex connectivity is at least two.

Value

For `articulation_points()`, a numeric vector giving the vertex IDs of the articulation points of the input graph. For `bridges()`, a numeric vector giving the edge IDs of the bridges of the input graph.

Related documentation in the C library

`articulation_points()`, `vcount()`, `bridges()`, `ecount()`, `edges()`, `get_eids()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

biconnected_components(), components(), is_connected(), vertex_connectivity(), edge_connectivity()

Connected components: biconnected_components(), component_distribution(), count_reachable(), decompose(), is_biconnected()

Examples

```
g <- disjoint_union(make_full_graph(5), make_full_graph(5))
clu <- components(g)$membership
g <- add_edges(g, c(match(1, clu), match(2, clu)))
articulation_points(g)

g <- make_graph("krackhardt_kite")
bridges(g)
```

---

as.igraph *Conversion to igraph*

Description

These functions convert various objects to igraph graphs.

Usage

```
as.igraph(x, ...)
```

Arguments

x	The object to convert.
...	Additional arguments. None currently.

Details

You can use as.igraph() to convert various objects to igraph graphs. Right now the following objects are supported:

- codeigraphHRG These objects are created by the fit_hrg() and consensus_tree() functions.

Value

All these functions return an igraph graph.

Related documentation in the C library

create(), vcount(), famous(), empty(), simplify()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>.

Examples

```
g <- make_full_graph(5) + make_full_graph(5)
hrg <- fit_hrg(g)
as.igraph(hrg)
```

---

as.matrix.igraph *Convert igraph objects to adjacency or edge list matrices*

Description

Get adjacency or edgelist representation of the network stored as an igraph object.

Usage

```
## S3 method for class 'igraph'
as.matrix(x, matrix.type = c("adjacency", "edgelist"), ...)
```

Arguments

x	object of class igraph, the network
matrix.type	character, type of matrix to return, currently "adjacency" or "edgelist" are supported
...	other arguments to/from other methods

Details

If matrix.type is "edgelist", then a two-column numeric edge list matrix is returned. The value of attrname is ignored.

If matrix.type is "adjacency", then a square adjacency matrix is returned. For adjacency matrices, you can use the attr keyword argument to use the values of an edge attribute in the matrix cells. See the documentation of as_adjacency_matrix for more details.

Other arguments passed through ... are passed to either as_adjacency_matrix() or as_edgelist() depending on the value of matrix.type.

Value

Depending on the value of matrix.type either a square adjacency matrix or a two-column numeric matrix representing the edgelist.

Related documentation in the C library

get_edgelist(), get_adjacency(), get_adjacency_sparse(), vcount(), edges(), get_eids(), ecount()

Author(s)

Michal Bojanowski, originally from the intergraph package

See Also

Other conversion: as_adj_list(), as_adjacency_matrix(), as_biadjacency_matrix(), as_data_frame(), as_directed(), as_edgelist(), as_graphnel(), as_long_data_frame(), graph_from_adj_list(), graph_from_graphnel()

Examples

```
g <- make_graph("zachary")
as.matrix(g, "adjacency")
as.matrix(g, "edgelist")
# use edge attribute "weight"
E(g)$weight <- rep(1:10, length.out = ecount(g))
as.matrix(g, "adjacency", sparse = FALSE, attr = "weight")
```

---

as_adj_list *Adjacency lists*

Description

Create adjacency lists from a graph, either for adjacent edges or for neighboring vertices

Usage

```
as_adj_list(
  graph,
  mode = c("all", "out", "in", "total"),
  loops = c("twice", "once", "ignore"),
  multiple = TRUE
)

as_adj_edge_list(
  graph,
  mode = c("all", "out", "in", "total"),
  loops = c("twice", "once", "ignore")
)
```

Arguments

graph	The input graph.
mode	Character scalar, it gives what kind of adjacent edges/vertices to include in the lists. 'out' is for outgoing edges/vertices, 'in' is for incoming edges/vertices, 'all' is for both. This argument is ignored for undirected graphs.
loops	Character scalar, one of "ignore" (to omit loops), "twice" (to include loop edges twice) and "once" (to include them once). "twice" is not allowed for directed graphs and will be replaced with "once".
multiple	Logical scalar, set to FALSE to use only one representative of each set of parallel edges.

Details

as_adj_list() returns a list of numeric vectors, which include the ids of neighbor vertices (according to the mode argument) of all vertices.

as_adj_edge_list() returns a list of numeric vectors, which include the ids of adjacent edges (according to the mode argument) of all vertices.

If igraph_opt("return.vs.es") is true (default), the numeric vectors of the adjacency lists are coerced to igraph.vs, this can be a very expensive operation on large graphs.

Value

A list of igraph.vs or a list of numeric vectors depending on the value of igraph_opt("return.vs.es"), see details for performance characteristics.

Related documentation in the C library

is_directed(), vcount(), edges(), get_eids(), ecount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

as_edgelist(), as_adjacency_matrix()

Other conversion: as.matrix.igraph(), as_adjacency_matrix(), as_biadjacency_matrix(), as_data_frame(), as_directed(), as_edgelist(), as_graphnel(), as_long_data_frame(), graph_from_adj_list(), graph_from_graphnel()

Examples

```
g <- make_ring(10)
as_adj_list(g)
as_adj_edge_list(g)
```

`as_adjacency_matrix` *Convert a graph to an adjacency matrix*

Description

Sometimes it is useful to work with a standard representation of a graph, like an adjacency matrix.

Usage

```
as_adjacency_matrix(
  graph,
  type = c("both", "upper", "lower"),
  attr = NULL,
  edges = deprecated(),
  names = TRUE,
  sparse = igraph_opt("sparsematrices")
)
```

Arguments

`graph`	The graph to convert.
`type`	Gives how to create the adjacency matrix for undirected graphs. It is ignored for directed graphs. Possible values: `upper`: the upper right triangle of the matrix is used, `lower`: the lower left triangle of the matrix is used. `both`: the whole matrix is used, a symmetric matrix is returned.
`attr`	Either `NULL` or a character string giving an edge attribute name. If `NULL` a traditional adjacency matrix is returned. If not `NULL` then the values of the given edge attribute are included in the adjacency matrix. If the graph has multiple edges, the edge attribute of an arbitrarily chosen edge (for the multiple edges) is included. This argument is ignored if `edges` is `TRUE`. Note that this works only for certain attribute types. If the `sparse` argumen is `TRUE`, then the attribute must be either logical or numeric. If the `sparse` argument is `FALSE`, then character is also allowed. The reason for the difference is that the `Matrix` package does not support character sparse matrices yet.
`edges`	**[Deprecated]** Logical scalar, whether to return the edge ids in the matrix. For non-existant edges zero is returned.
`names`	Logical constant, whether to assign row and column names to the matrix. These are only assigned if the `name` vertex attribute is present in the graph.
`sparse`	Logical scalar, whether to create a sparse matrix. The '`Matrix`' package must be installed for creating sparse matrices.

Details

`as_adjacency_matrix()` returns the adjacency matrix of a graph, a regular matrix if `sparse` is `FALSE`, or a sparse matrix, as defined in the '`Matrix`' package, if `sparse` if `TRUE`.

Value

A vcount(graph) by vcount(graph) (usually) numeric matrix.

Related documentation in the C library

get_adjacency(), get_adjacency_sparse(), vcount(), edges(), get_eids(), ecount()

See Also

graph_from_adjacency_matrix(), read_graph()

Other conversion: as.matrix.igraph(), as_adj_list(), as_biadjacency_matrix(), as_data_frame(), as_directed(), as_edgelist(), as_graphnel(), as_long_data_frame(), graph_from_adj_list(), graph_from_graphnel()

Examples

```
g <- sample_gnp(10, 2 / 10)
as_adjacency_matrix(g)
V(g)$name <- letters[1:vcount(g)]
as_adjacency_matrix(g)
E(g)$weight <- runif(ecount(g))
as_adjacency_matrix(g, attr = "weight")
```

---

as_biadjacency_matrix *Bipartite adjacency matrix of a bipartite graph*

Description

This function can return a sparse or dense bipartite adjacency matrix of a bipartite network. The bipartite adjacency matrix is an n times m matrix, n and m are the number of vertices of the two kinds.

Usage

```
as_biadjacency_matrix(
  graph,
  types = NULL,
  attr = NULL,
  names = TRUE,
  sparse = FALSE
)
```

Arguments

graph	The input graph. The direction of the edges is ignored in directed graphs.
types	An optional vertex type vector to use instead of the type vertex attribute. You must supply this argument if the graph has no type vertex attribute.
attr	Either NULL or a character string giving an edge attribute name. If NULL, then a traditional bipartite adjacency matrix is returned. If not NULL then the values of the given edge attribute are included in the bipartite adjacency matrix. If the graph has multiple edges, the edge attribute of an arbitrarily chosen edge (for the multiple edges) is included.
names	Logical scalar, if TRUE and the vertices in the graph are named (i.e. the graph has a vertex attribute called name), then vertex names will be added to the result as row and column names. Otherwise the ids of the vertices are used as row and column names.
sparse	Logical scalar, if it is TRUE then a sparse matrix is created, you will need the Matrix package for this.

Details

Bipartite graphs have a type vertex attribute in igraph, this is boolean and FALSE for the vertices of the first kind and TRUE for vertices of the second kind.

Some authors refer to the bipartite adjacency matrix as the "bipartite incidence matrix". igraph 1.6.0 and later does not use this naming to avoid confusion with the edge-vertex incidence matrix.

Value

A sparse or dense matrix.

Related documentation in the C library

get_biadjacency(), get_edgelist(), vcount(), edges(), get_eids(), ecount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

graph_from_biadjacency_matrix() for the opposite operation.

Other conversion: as.matrix.igraph(), as_adj_list(), as_adjacency_matrix(), as_data_frame(), as_directed(), as_edgelist(), as_graphnel(), as_long_data_frame(), graph_from_adj_list(), graph_from_graphnel()

Examples

```
g <- make_bipartite_graph(c(0, 1, 0, 1, 0, 0), c(1, 2, 2, 3, 3, 4))
as_biadjacency_matrix(g)
```

as_data_frame *Creating igraph graphs from data frames or vice-versa*

Description

This function creates an igraph graph from one or two data frames containing the (symbolic) edge list and edge/vertex attributes.

Usage

```
as_data_frame(x, what = c("edges", "vertices", "both"))

graph_from_data_frame(d, directed = TRUE, vertices = NULL)

from_data_frame(...)
```

Arguments

x	An igraph object.
what	Character constant, whether to return info about vertices, edges, or both. The default is 'edges'.
d	A data frame containing a symbolic edge list in the first two columns. Additional columns are considered as edge attributes. Since version 0.7 this argument is coerced to a data frame with `as.data.frame`.
directed	Logical scalar, whether or not to create a directed graph.
vertices	A data frame with vertex metadata, or `NULL`. See details below. Since version 0.7 this argument is coerced to a data frame with `as.data.frame`, if not `NULL`.
...	Passed to `graph_from_data_frame()`.

Details

`graph_from_data_frame()` creates igraph graphs from one or two data frames. It has two modes of operation, depending whether the `vertices` argument is `NULL` or not.

If `vertices` is `NULL`, then the first two columns of `d` are used as a symbolic edge list and additional columns as edge attributes. The names of the attributes are taken from the names of the columns.

If `vertices` is not `NULL`, then it must be a data frame giving vertex metadata. The first column of `vertices` is assumed to contain symbolic vertex names, this will be added to the graphs as the 'name' vertex attribute. Other columns will be added as additional vertex attributes. If `vertices` is not `NULL` then the symbolic edge list given in `d` is checked to contain only vertex names listed in `vertices`.

Typically, the data frames are exported from some spreadsheet software like Excel and are imported into R via `read.table()`, `read.delim()` or `read.csv()`.

All edges in the data frame are included in the graph, which may include multiple parallel edges and loops.

as_data_frame() converts the igraph graph into one or more data frames, depending on the what argument.

If the what argument is edges (the default), then the edges of the graph and also the edge attributes are returned. The edges will be in the first two columns, named from and to. (This also denotes edge direction for directed graphs.) For named graphs, the vertex names will be included in these columns, for other graphs, the numeric vertex ids. The edge attributes will be in the other columns. It is not a good idea to have an edge attribute named from or to, because then the column named in the data frame will not be unique. The edges are listed in the order of their numeric ids.

If the what argument is vertices, then vertex attributes are returned. Vertices are listed in the order of their numeric vertex ids.

If the what argument is both, then both vertex and edge data is returned, in a list with named entries vertices and edges.

Value

An igraph graph object for graph_from_data_frame(), and either a data frame or a list of two data frames named edges and vertices for as.data.frame.

Related documentation in the C library

get_edgelist(), vcount(), add_vertices(), empty(), edges(), get_eids(), ecount()

Note

For graph_from_data_frame() NA elements in the first two columns 'd' are replaced by the string "NA" before creating the graph. This means that all NAs will correspond to a single vertex.

NA elements in the first column of 'vertices' are also replaced by the string "NA", but the rest of 'vertices' is not touched. In other words, vertex names (=the first column) cannot be NA, but other vertex attributes can.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

graph_from_literal() for another way to create graphs, read.table() to read in tables from files.

Other conversion: as.matrix.igraph(), as_adj_list(), as_adjacency_matrix(), as_biadjacency_matrix(), as_directed(), as_edgelist(), as_graphnel(), as_long_data_frame(), graph_from_adj_list(), graph_from_graphnel()

Other biadjacency: graph_from_biadjacency_matrix()

Examples

```
## A simple example with a couple of actors
## The typical case is that these tables are read in from files....
actors <- data.frame(
```

```
  name = c(
    "Alice", "Bob", "Cecil", "David",
    "Esmeralda"
  ),
  age = c(48, 33, 45, 34, 21),
  gender = c("F", "M", "F", "M", "F")
)
relations <- data.frame(
  from = c(
    "Bob", "Cecil", "Cecil", "David",
    "David", "Esmeralda"
  ),
  to = c("Alice", "Bob", "Alice", "Alice", "Bob", "Alice"),
  same.dept = c(FALSE, FALSE, TRUE, FALSE, FALSE, TRUE),
  friendship = c(4, 5, 5, 2, 1, 1), advice = c(4, 5, 5, 4, 2, 3)
)
g <- graph_from_data_frame(relations, directed = TRUE, vertices = actors)
print(g, e = TRUE, v = TRUE)

## The opposite operation
as_data_frame(g, what = "vertices")
as_data_frame(g, what = "edges")
```

`as_directed` *Convert between directed and undirected graphs*

Description

`as_directed()` converts an undirected graph to directed, `as_undirected()` does the opposite, it converts a directed graph to undirected.

Usage

```
as_directed(graph, mode = c("mutual", "arbitrary", "random", "acyclic"))

as_undirected(
  graph,
  mode = c("collapse", "each", "mutual"),
  edge.attr.comb = igraph_opt("edge.attr.comb")
)
```

Arguments

`graph`	The graph to convert.
`mode`	Character constant, defines the conversion algorithm. For `as_directed()` it can be `mutual` or `arbitrary`. For `as_undirected()` it can be `each`, `collapse` or `mutual`. See details below.

`edge.attr.comb` Specifies what to do with edge attributes, if `mode="collapse"` or `mode="mutual"`. In these cases many edges might be mapped to a single one in the new graph, and their attributes are combined. Please see `attribute.combination()` for details on this.

Details

Conversion algorithms for `as_directed()`:

"arbitrary" The number of edges in the graph stays the same, an arbitrarily directed edge is created for each undirected edge, but the direction of the edge is deterministic (i.e. it always points the same way if you call the function multiple times).

"mutual" Two directed edges are created for each undirected edge, one in each direction.

"random" The number of edges in the graph stays the same, and a randomly directed edge is created for each undirected edge. You will get different results if you call the function multiple times with the same graph.

"acyclic" The number of edges in the graph stays the same, and a directed edge is created for each undirected edge such that the resulting graph is guaranteed to be acyclic. This is achieved by ensuring that edges always point from a lower index vertex to a higher index. Note that the graph may include cycles of length 1 if the original graph contained loop edges.

Conversion algorithms for `as_undirected()`:

"each" The number of edges remains constant, an undirected edge is created for each directed one, this version might create graphs with multiple edges.

"collapse" One undirected edge will be created for each pair of vertices which are connected with at least one directed edge, no multiple edges will be created.

"mutual" One undirected edge will be created for each pair of mutual edges. Non-mutual edges are ignored. This mode might create multiple edges if there are more than one mutual edge pairs between the same pair of vertices.

Value

A new graph object.

Related documentation in the C library

`to_directed()`, `to_undirected()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

`simplify()` for removing multiple and/or loop edges from a graph.

Other conversion: `as.matrix.igraph()`, `as_adj_list()`, `as_adjacency_matrix()`, `as_biadjacency_matrix()`, `as_data_frame()`, `as_edgelist()`, `as_graphnel()`, `as_long_data_frame()`, `graph_from_adj_list()`, `graph_from_graphnel()`

Examples

```
g <- make_ring(10)
as_directed(g, "mutual")
g2 <- make_star(10)
as_undirected(g)

# Combining edge attributes
g3 <- make_ring(10, directed = TRUE, mutual = TRUE)
E(g3)$weight <- seq_len(ecount(g3))
ug3 <- as_undirected(g3)
print(ug3, e = TRUE)

x11(width = 10, height = 5)
layout(rbind(1:2))
plot(g3, layout = layout_in_circle, edge.label = E(g3)$weight)
plot(ug3, layout = layout_in_circle, edge.label = E(ug3)$weight)


g4 <- make_graph(c(
  1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 5, 4,
  6, 7, 7, 6, 7, 8, 7, 8, 8, 7, 8, 9, 8, 9,
  9, 8, 9, 8, 9, 9, 10, 10, 10, 10
))
E(g4)$weight <- seq_len(ecount(g4))
ug4 <- as_undirected(g4,
  mode = "mutual",
  edge.attr.comb = list(weight = length)
)
print(ug4, e = TRUE)
```

---

as_edgelist *Convert a graph to an edge list*

Description

Sometimes it is useful to work with a standard representation of a graph, like an edge list.

Usage

```
as_edgelist(graph, names = TRUE)
```

Arguments

graph	The graph to convert.
names	Whether to return a character matrix containing vertex names (i.e. the name vertex attribute) if they exist or numeric vertex ids.

Details

as_edgelist() returns the list of edges in a graph.

Value

A ecount(graph) by 2 numeric matrix.

Related documentation in the C library

get_edgelist(), vcount()

See Also

graph_from_adjacency_matrix(), read_graph()

Other conversion: as.matrix.igraph(), as_adj_list(), as_adjacency_matrix(), as_biadjacency_matrix(), as_data_frame(), as_directed(), as_graphnel(), as_long_data_frame(), graph_from_adj_list(), graph_from_graphnel()

Examples

```
g <- sample_gnp(10, 2 / 10)
as_edgelist(g)

V(g)$name <- LETTERS[seq_len(gorder(g))]
as_edgelist(g)
```

---

as_graphnel *Convert igraph graphs to graphNEL objects from the graph package*

Description

The graphNEL class is defined in the graph package, it is another way to represent graphs. These functions are provided to convert between the igraph and the graphNEL objects.

Usage

```
as_graphnel(graph)
```

Arguments

graph	An igraph graph object.

Details

as_graphnel() converts an igraph graph to a graphNEL graph. It converts all graph/vertex/edge attributes. If the igraph graph has a vertex attribute 'name', then it will be used to assign vertex names in the graphNEL graph. Otherwise numeric igraph vertex ids will be used for this purpose.

Value

`as_graphnel()` returns a graphNEL graph object.

Related documentation in the C library

`get_edgelist()`, `is_directed()`, `edges()`, `vcount()`, `has_multiple()`, `get_eids()`, `ecount()`

See Also

`graph_from_graphnel()` for the other direction, `as_adjacency_matrix()`, `graph_from_adjacency_matrix()`, `as_adj_list()` and `graph_from_adj_list()` for other graph representations.

Other conversion: `as.matrix.igraph()`, `as_adj_list()`, `as_adjacency_matrix()`, `as_biadjacency_matrix()`, `as_data_frame()`, `as_directed()`, `as_edgelist()`, `as_long_data_frame()`, `graph_from_adj_list()`, `graph_from_graphnel()`

Examples

```
## Undirected
g <- make_ring(10)
V(g)$name <- letters[1:10]
GNEL <- as_graphnel(g)
g2 <- graph_from_graphnel(GNEL)
g2

## Directed
g3 <- make_star(10, mode = "in")
V(g3)$name <- letters[1:10]
GNEL2 <- as_graphnel(g3)
g4 <- graph_from_graphnel(GNEL2)
g4
```

---

as_ids *Convert a vertex or edge sequence to an ordinary vector*

Description

Convert a vertex or edge sequence to an ordinary vector

Usage

```
as_ids(seq)

## S3 method for class 'igraph.vs'
as_ids(seq)

## S3 method for class 'igraph.es'
as_ids(seq)
```

Arguments

seq	The vertex or edge sequence.

Details

For graphs without names, a numeric vector is returned, containing the internal numeric vertex or edge ids.

For graphs with names, and vertex sequences, the vertex names are returned in a character vector.

For graphs with names and edge sequences, a character vector is returned, with the 'bar' notation: a|b means an edge from vertex a to vertex b.

Value

A character or numeric vector, see details below.

See Also

Other vertex and edge sequences: E(), V(), igraph-es-attributes, igraph-es-indexing, igraph-es-indexing2, igraph-vs-attributes, igraph-vs-indexing, igraph-vs-indexing2, print.igraph.es(), print.igraph.vs()

Examples

```
g <- make_ring(10)
as_ids(V(g))
as_ids(E(g))

V(g)$name <- letters[1:10]
as_ids(V(g))
as_ids(E(g))
```

---

as_long_data_frame *Convert a graph to a long data frame*

Description

A long data frame contains all metadata about both the vertices and edges of the graph. It contains one row for each edge, and all metadata about that edge and its incident vertices are included in that row. The names of the columns that contain the metadata of the incident vertices are prefixed with from_ and to_. The first two columns are always named from and to and they contain the numeric ids of the incident vertices. The rows are listed in the order of numeric vertex ids.

Usage

```
as_long_data_frame(graph)
```

Arguments

graph	Input graph

Value

A long data frame.

Related documentation in the C library

get_edgelist(), vcount()

See Also

Other conversion: as.matrix.igraph(), as_adj_list(), as_adjacency_matrix(), as_biadjacency_matrix(), as_data_frame(), as_directed(), as_edgelist(), as_graphnel(), graph_from_adj_list(), graph_from_graphnel()

Examples

```
g <- make_(
  ring(10),
  with_vertex_(name = letters[1:10], color = "red"),
  with_edge_(weight = 1:10, color = "green")
)
as_long_data_frame(g)
```

---

as_membership *Declare a numeric vector as a membership vector*

Description

This is useful if you want to use functions defined on membership vectors, but your membership vector does not come from an igraph clustering method.

Usage

```
as_membership(x)
```

Arguments

x	The input vector.

Value

The input vector, with the `membership` class added.

See Also

Community detection: cluster_edge_betweenness(), cluster_fast_greedy(), cluster_fluid_communities(), cluster_infomap(), cluster_label_prop(), cluster_leading_eigen(), cluster_leiden(), cluster_louvain(), cluster_optimal(), cluster_spinglass(), cluster_walktrap(), compare(), groups(), make_clusters(), membership(), modularity.igraph(), plot_dendrogram(), split_join_distance(), voronoi_cells()

Examples

```
## Compare to the correct clustering
g <- (make_full_graph(10) + make_full_graph(10)) %>%
  rewire(each_edge(p = 0.2))
correct <- rep(1:2, each = 10) %>% as_membership()
fc <- cluster_fast_greedy(g)
compare(correct, fc)
compare(correct, membership(fc))
```

---

assortativity *Assortativity coefficient*

Description

The assortativity coefficient is positive if similar vertices (based on some external property) tend to connect to each, and negative otherwise.

Usage

```
assortativity(
  graph,
  values,
  ...,
  values.in = NULL,
  directed = TRUE,
  normalized = TRUE,
  types1 = NULL,
  types2 = NULL
)

assortativity_nominal(graph, types, directed = TRUE, normalized = TRUE)

assortativity_degree(graph, directed = TRUE)
```

Arguments

graph	The input graph, it can be directed or undirected.
values	The vertex values, these can be arbitrary numeric values.
...	These dots are for future extensions and must be empty.
values.in	A second value vector to use for the incoming edges when calculating assortativity for a directed graph. Supply NULL here if you want to use the same values for outgoing and incoming edges. This argument is ignored (with a warning) if it is not NULL and undirected assortativity coefficient is being calculated.
directed	Logical scalar, whether to consider edge directions for directed graphs. This argument is ignored for undirected graphs. Supply TRUE here to do the natural thing, i.e. use directed version of the measure for directed graphs and the undirected version for undirected graphs.

`normalized` Boolean, whether to compute the normalized assortativity. The non-normalized nominal assortativity is identical to modularity. The non-normalized value-based assortativity is simply the covariance of the values at the two ends of edges.

`types1`, `types2` **[Deprecated]** Deprecated aliases for `values` and `values.in`, respectively.

`types` Vector giving the vertex types. They as assumed to be integer numbers, starting with one. Non-integer values are converted to integers with `as.integer()`. Character vectors are converted to integers using `as.factor()`.

Details

The assortativity coefficient measures the level of homophyly of the graph, based on some vertex labeling or values assigned to vertices. If the coefficient is high, that means that connected vertices tend to have the same labels or similar assigned values.

M.E.J. Newman defined two kinds of assortativity coefficients, the first one is for categorical labels of vertices. `assortativity_nominal()` calculates this measure. It is defined as

$$r = \frac{\sum_i e_{ii} - \sum_i a_i b_i}{1 - \sum_i a_i b_i}$$

where e_{ij} is the fraction of edges connecting vertices of type i and j, $a_i = \sum_j e_{ij}$ and $b_j = \sum_i e_{ij}$.

The second assortativity variant is based on values assigned to the vertices. `assortativity()` calculates this measure. It is defined as

$$r = \frac{1}{\sigma_q^2} \sum_{jk} jk(e_{jk} - q_j q_k)$$

for undirected graphs ($q_i = \sum_j e_{ij}$) and as

$$r = \frac{1}{\sigma_o \sigma_i} \sum_{jk} jk(e_{jk} - q_j^o q_k^i)$$

for directed ones. Here $q_i^o = \sum_j e_{ij}$, $q_i^i = \sum_j e_{ji}$, moreover, σ_q, σ_o and σ_i are the standard deviations of q, q^o and q^i, respectively.

The reason of the difference is that in directed networks the relationship is not symmetric, so it is possible to assign different values to the outgoing and the incoming end of the edges.

`assortativity_degree()` uses vertex degree as vertex values and calls `assortativity()`.

Undirected graphs are effectively treated as directed ones with all-reciprocal edges. Thus, self-loops are taken into account twice in undirected graphs.

Value

A single real number.

Related documentation in the C library

`assortativity()`, `assortativity_nominal()`, `assortativity_degree()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

M. E. J. Newman: Mixing patterns in networks, *Phys. Rev. E* 67, 026126 (2003) https://arxiv.org/abs/cond-mat/0209450

M. E. J. Newman: Assortative mixing in networks, *Phys. Rev. Lett.* 89, 208701 (2002) https://arxiv.org/abs/cond-mat/0205405

Examples

```
# random network, close to zero
assortativity_degree(sample_gnp(10000, 3 / 10000))

# BA model, tends to be dissortative
assortativity_degree(sample_pa(10000, m = 4))
```

---

authority_score *Kleinberg's authority centrality scores.*

Description

Kleinberg's authority centrality scores.

Kleinberg's hub centrality scores.

Usage

```
authority_score(
  graph,
  scale = TRUE,
  weights = NULL,
  options = arpack_defaults()
)

hub_score(graph, scale = TRUE, weights = NULL, options = arpack_defaults())
```

Arguments

graph	The input graph.
scale	Logical scalar, whether to scale the result to have a maximum score of one. If no scaling is used then the result vector has unit length in the Euclidean norm.
weights	Optional positive weight vector for calculating weighted scores. If the graph has a `weight` edge attribute, then this is used by default. This function interprets edge weights as connection strengths. In the random surfer model, an edge with a larger weight is more likely to be selected by the surfer.
options	A named list, to override some ARPACK options. See `arpack()` for details.

Related documentation in the C library

hub_and_authority_scores(), vcount(), edges(), get_eids(), ecount()

See Also

Centrality measures: alpha_centrality(), betweenness(), closeness(), diversity(), eigen_centrality(), harmonic_centrality(), hits_scores(), page_rank(), power_centrality(), spectrum(), strength(), subgraph_centrality()

---

automorphism_group *Generating set of the automorphism group of a graph*

Description

Compute the generating set of the automorphism group of a graph.

Usage

```
automorphism_group(
  graph,
  colors = NULL,
  sh = c("fm", "f", "fs", "fl", "flm", "fsm"),
  details = FALSE
)
```

Arguments

graph	The input graph, it is treated as undirected.
colors	The colors of the individual vertices of the graph; only vertices having the same color are allowed to match each other in an automorphism. When omitted, igraph uses the color attribute of the vertices, or, if there is no such vertex attribute, it simply assumes that all vertices have the same color. Pass NULL explicitly if the graph has a color vertex attribute but you do not want to use it.
sh	The splitting heuristics for the BLISS algorithm. Possible values are: 'f': first non-singleton cell, 'fl': first largest non-singleton cell, 'fs': first smallest non-singleton cell, 'fm': first maximally non-trivially connected non-singleton cell, 'flm': first largest maximally non-trivially connected non-singleton cell, 'fsm': first smallest maximally non-trivially connected non-singleton cell.
details	Specifies whether to provide additional details about the BLISS internals in the result.

Details

An automorphism of a graph is a permutation of its vertices which brings the graph into itself. The automorphisms of a graph form a group and there exists a subset of this group (i.e. a set of permutations) such that every other permutation can be expressed as a combination of these permutations. These permutations are called the generating set of the automorphism group.

This function calculates a possible generating set of the automorphism of a graph using the BLISS algorithm. See also the BLISS homepage at http://www.tcs.hut.fi/Software/bliss/index.html. The calculated generating set is not necessarily minimal, and it may depend on the splitting heuristics used by BLISS.

Value

When `details` is `FALSE`, a list of vertex permutations that form a generating set of the automorphism group of the input graph. When `details` is `TRUE`, a named list with two members:

generators Returns the generators themselves

info Additional information about the BLISS internals. See `count_automorphisms()` for more details.

Related documentation in the C library

`automorphism_group()`, `vcount()`

Author(s)

Tommi Junttila (https://users.ics.aalto.fi/tjunttil/) for BLISS, Gabor Csardi <csardi.gabor@gmail.com> for the igraph glue code and Tamas Nepusz <ntamas@gmail.com> for this manual page.

References

Tommi Junttila and Petteri Kaski: Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs, *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics.* 2007.

See Also

`canonical_permutation()`, `permute()`, `count_automorphisms()`

Other graph automorphism: `count_automorphisms()`

Examples

```
## A ring has n*2 automorphisms, and a possible generating set is one that
## "turns" the ring by one vertex to the left or right
g <- make_ring(10)
automorphism_group(g)
```

`betweenness` *Vertex and edge betweenness centrality*

Description

The vertex and edge betweenness are (roughly) defined by the number of geodesics (shortest paths) going through a vertex or an edge.

Usage

```
betweenness(
  graph,
  v = V(graph),
  directed = TRUE,
  weights = NULL,
  normalized = FALSE,
  cutoff = -1
)

edge_betweenness(
  graph,
  e = E(graph),
  directed = TRUE,
  weights = NULL,
  cutoff = -1
)
```

Arguments

`graph`	The graph to analyze.
`v`	The vertices for which the vertex betweenness will be calculated.
`directed`	Logical, whether directed paths should be considered while determining the shortest paths.
`weights`	Optional positive weight vector for calculating weighted betweenness. If the graph has a `weight` edge attribute, then this is used by default. Weights are used to calculate weighted shortest paths, so they are interpreted as distances.
`normalized`	Logical scalar, whether to normalize the betweenness scores. If `TRUE`, then the results are normalized by the number of ordered or unordered vertex pairs in directed and undirected graphs, respectively. In an undirected graph, $$B^n = \frac{2B}{(n-1)(n-2)},$$ where B^n is the normalized, B the raw betweenness, and n is the number of vertices in the graph. Note that the same normalization factor is used even when setting a `cutoff` on the considered shortest path lengths, even though the number of vertex pairs reachable from each other may be less than $(n-1)(n-2)/2$.

cutoff	The maximum shortest path length to consider when calculating betweenness. If negative, then there is no such limit.
e	The edges for which the edge betweenness will be calculated.

Details

The vertex betweenness of vertex v is defined by

$$\sum_{i \neq j, i \neq v, j \neq v} g_{ivj}/g_{ij}$$

The edge betweenness of edge e is defined by

$$\sum_{i \neq j} g_{iej}/g_{ij}.$$

betweenness() calculates vertex betweenness, edge_betweenness() calculates edge betweenness.

Here g_{ij} is the total number of shortest paths between vertices i and j while g_{ivj} is the number of those shortest paths which pass though vertex v.

Both functions allow you to consider only paths of length cutoff or smaller; this can be run for larger graphs, as the running time is not quadratic (if cutoff is small). If cutoff is negative (the default), then the function calculates the exact betweenness scores. Since igraph 1.6.0, a cutoff value of zero is treated literally, i.e. paths of length larger than zero are ignored.

For calculating the betweenness a similar algorithm to the one proposed by Brandes (see References) is used.

Value

A numeric vector with the betweenness score for each vertex in v for betweenness().

A numeric vector with the edge betweenness score for each edge in e for edge_betweenness().

Related documentation in the C library

betweenness_cutoff(), is_directed(), vcount(), edges(), get_eids(), ecount(), edge_betweenness_cutoff()

Note

edge_betweenness() might give false values for graphs with multiple edges.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Freeman, L.C. (1979). Centrality in Social Networks I: Conceptual Clarification. *Social Networks*, 1, 215-239. doi:10.1016/03788733(78)900217

Ulrik Brandes, A Faster Algorithm for Betweenness Centrality. *Journal of Mathematical Sociology* 25(2):163-177, 2001. doi:10.1080/0022250X.2001.9990249

See Also

`closeness()`, `degree()`, `harmonic_centrality()`

Centrality measures: `alpha_centrality()`, `authority_score()`, `closeness()`, `diversity()`, `eigen_centrality()`, `harmonic_centrality()`, `hits_scores()`, `page_rank()`, `power_centrality()`, `spectrum()`, `strength()`, `subgraph_centrality()`

Examples

```
g <- sample_gnp(10, 3 / 10)
betweenness(g)
edge_betweenness(g)
```

---

bfs *Breadth-first search*

Description

Breadth-first search is an algorithm to traverse a graph. We start from a root vertex and spread along every edge "simultaneously".

Usage

```
bfs(
  graph,
  root,
  mode = c("out", "in", "all", "total"),
  ...,
  unreachable = TRUE,
  restricted = NULL,
  order = TRUE,
  rank = FALSE,
  parent = FALSE,
  pred = FALSE,
  succ = FALSE,
  dist = FALSE,
  callback = NULL,
  extra = NULL,
  rho = parent.frame(),
```

```
  neimode = deprecated(),
  father = deprecated()
)
```

Arguments

graph	The input graph.
root	Numeric vector, usually of length one. The root vertex, or root vertices to start the search from.
mode	For directed graphs specifies the type of edges to follow. 'out' follows outgoing, 'in' incoming edges. 'all' ignores edge directions completely. 'total' is a synonym for 'all'. This argument is ignored for undirected graphs.
...	These dots are for future extensions and must be empty.
unreachable	Logical scalar, whether the search should visit the vertices that are unreachable from the given root vertex (or vertices). If TRUE, then additional searches are performed until all vertices are visited.
restricted	NULL (=no restriction), or a vector of vertices (ids or symbolic names). In the latter case, the search is restricted to the given vertices.
order	Logical scalar, whether to return the ordering of the vertices.
rank	Logical scalar, whether to return the rank of the vertices.
parent	Logical scalar, whether to return the parent of the vertices.
pred	Logical scalar, whether to return the predecessors of the vertices.
succ	Logical scalar, whether to return the successors of the vertices.
dist	Logical scalar, whether to return the distance from the root of the search tree.
callback	If not NULL, then it must be callback function. This is called whenever a vertex is visited. The callback function should return FALSE to continue the search or TRUE to stop it. See details below.
extra	Additional argument to supply to the callback function.
rho	The environment in which the callback function is evaluated.
neimode	**[Deprecated]** This argument is deprecated from igraph 1.3.0; use mode instead.
father	**[Deprecated]** Use parent instead.

Details

The callback function must have the following arguments:

graph The input graph is passed to the callback function here.

data A named numeric vector, with the following entries: 'vid', the vertex that was just visited, 'pred', its predecessor (zero if this is the first vertex), 'succ', its successor (zero if this is the last vertex), 'rank', the rank of the current vertex, 'dist', its distance from the root of the search tree.

extra The extra argument.

The callback must return FALSE to continue the search or TRUE to terminate it. See examples below on how to use the callback function.

Value

A named list with the following entries:

root Numeric scalar. The root vertex that was used as the starting point of the search.

neimode Character scalar. The mode argument of the function call. Note that for undirected graphs this is always 'all', irrespectively of the supplied value.

order Numeric vector. The vertex ids, in the order in which they were visited by the search.

rank Numeric vector. The rank for each vertex, zero for unreachable vertices.

parent Numeric vector. The parent of each vertex, i.e. the vertex it was discovered from.

father Like parent, kept for compatibility for now.

pred Numeric vector. The previously visited vertex for each vertex, or 0 if there was no such vertex.

succ Numeric vector. The next vertex that was visited after the current one, or 0 if there was no such vertex.

dist Numeric vector, for each vertex its distance from the root of the search tree. Unreachable vertices have a negative distance as of igraph 1.6.0, this used to be NaN.

Note that order, rank, parent, pred, succ and dist might be NULL if their corresponding argument is FALSE, i.e. if their calculation is not requested.

Related documentation in the C library

vcount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

dfs() for depth-first search.

Other structural.properties: component_distribution(), connect(), constraint(), coreness(), degree(), dfs(), distance_table(), edge_density(), feedback_arc_set(), feedback_vertex_set(), girth(), is_acyclic(), is_dag(), is_matching(), k_shortest_paths(), knn(), reciprocity(), subcomponent(), subgraph(), topo_sort(), transitivity(), unfold_tree(), which_multiple(), which_mutual()

Examples

```
## Two rings
bfs(make_ring(10) %du% make_ring(10),
  root = 1, "out",
  order = TRUE, rank = TRUE, parent = TRUE, pred = TRUE,
  succ = TRUE, dist = TRUE
)

## How to use a callback
```

```
f <- function(graph, data, extra) {
  print(data)
  FALSE
}
tmp <- bfs(make_ring(10) %du% make_ring(10),
  root = 1, "out",
  callback = f
)

## How to use a callback to stop the search
## We stop after visiting all vertices in the initial component
f <- function(graph, data, extra) {
  data["succ"] == -1
}
bfs(make_ring(10) %du% make_ring(10), root = 1, callback = f)
```

`biconnected_components` *Biconnected components*

Description

Finding the biconnected components of a graph

Usage

```
biconnected_components(graph)
```

Arguments

`graph`	The input graph. It is treated as an undirected graph, even if it is directed.

Details

A graph is biconnected if the removal of any single vertex (and its adjacent edges) does not disconnect it.

A biconnected component of a graph is a maximal biconnected subgraph of it. The biconnected components of a graph can be given by the partition of its edges: every edge is a member of exactly one biconnected component. Note that this is not true for vertices: the same vertex can be part of many biconnected components.

Value

A named list with three components:

no Numeric scalar, an integer giving the number of biconnected components in the graph.

tree_edges The components themselves, a list of numeric vectors. Each vector is a set of edge ids giving the edges in a biconnected component. These edges define a spanning tree of the component.

component_edges A list of numeric vectors. It gives all edges in the components.

components A list of numeric vectors, the vertices of the components.

articulation_points The articulation points of the graph. See articulation_points().

Related documentation in the C library

biconnected_components(), vcount(), edges(), get_eids(), ecount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

articulation_points(), components(), is_connected(), vertex_connectivity()

Connected components: articulation_points(), component_distribution(), count_reachable(), decompose(), is_biconnected()

Examples

```
g <- disjoint_union(make_full_graph(5), make_full_graph(5))
clu <- components(g)$membership
g <- add_edges(g, c(which(clu == 1), which(clu == 2)))
bc <- biconnected_components(g)
```

---

bipartite_gnm *Bipartite random graphs*

Description

Generate bipartite graphs using the Erdős-Rényi model

Usage

```
bipartite_gnm(n1, n2, m, ..., directed = FALSE, mode = c("out", "in", "all"))

bipartite_gnp(n1, n2, p, ..., directed = FALSE, mode = c("out", "in", "all"))

sample_bipartite_gnm(
  n1,
  n2,
  m,
  ...,
  directed = FALSE,
```

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

sample_bipartite_gnp(
  n1,
  n2,
  p,
  ...,
  directed = FALSE,
  mode = c("out", "in", "all")
)
```

Arguments

n1	Integer scalar, the number of bottom vertices.
n2	Integer scalar, the number of top vertices.
m	Integer scalar, the number of edges for $G(n, m)$ graphs.
...	These dots are for future extensions and must be empty.
directed	Logical scalar, whether to create a directed graph. See also the mode argument.
mode	Character scalar, specifies how to direct the edges in directed graphs. If it is 'out', then directed edges point from bottom vertices to top vertices. If it is 'in', edges point from top vertices to bottom vertices. 'out' and 'in' do not generate mutual edges. If this argument is 'all', then each edge direction is considered independently and mutual edges might be generated. This argument is ignored for undirected graphs.
p	Real scalar, connection probability for $G(n, p)$ graphs.

Details

Similarly to unipartite (one-mode) networks, we can define the $G(n, p)$, and $G(n, m)$ graph classes for bipartite graphs, via their generating process. In $G(n, p)$ every possible edge between top and bottom vertices is realized with probability p, independently of the rest of the edges. In $G(n, m)$, we uniformly choose m edges to realize.

Related documentation in the C library

bipartite_game_gnm(), vcount(), bipartite_game_gnp()

See Also

Random graph models (games): erdos.renyi.game(), sample_(), sample_bipartite(), sample_chung_lu(), sample_correlated_gnp(), sample_correlated_gnp_pair(), sample_degseq(), sample_dot_product(), sample_fitness(), sample_fitness_pl(), sample_forestfire(), sample_gnm(), sample_gnp(), sample_grg(), sample_growing(), sample_hierarchical_sbm(), sample_islands(), sample_k_regular(), sample_last_cit(), sample_pa(), sample_pa_age(), sample_pref(), sample_sbm(), sample_smallworld(), sample_traits_callaway(), sample_tree()

Examples

```
## empty graph
sample_bipartite_gnp(10, 5, p = 0)

## full graph
sample_bipartite_gnp(10, 5, p = 1)

## random bipartite graph
sample_bipartite_gnp(10, 5, p = .1)

## directed bipartite graph, G(n,m)
sample_bipartite_gnm(10, 5, m = 20, directed = TRUE, mode = "all")
```

---

bipartite_mapping *Decide whether a graph is bipartite*

Description

This function decides whether the vertices of a network can be mapped to two vertex types in a way that no vertices of the same type are connected.

Usage

```
bipartite_mapping(graph)
```

Arguments

graph	The input graph.

Details

A bipartite graph in igraph has a 'type' vertex attribute giving the two vertex types.

This function simply checks whether a graph *could* be bipartite. It tries to find a mapping that gives a possible division of the vertices into two classes, such that no two vertices of the same class are connected by an edge.

The existence of such a mapping is equivalent of having no circuits of odd length in the graph. A graph with loop edges cannot bipartite.

Note that the mapping is not necessarily unique, e.g. if the graph has at least two components, then the vertices in the separate components can be mapped independently.

Value

A named list with two elements:

res A logical scalar, TRUE if the can be bipartite, FALSE otherwise.

type A possible vertex type mapping, a logical vector. If no such mapping exists, then an empty vector.

Related documentation in the C library

is_bipartite(), vcount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Bipartite graphs: bipartite_projection(), is_bipartite(), make_bipartite_graph()

Examples

```
## Rings with an even number of vertices are bipartite
g <- make_ring(10)
bipartite_mapping(g)

## All star graphs are bipartite
g2 <- make_star(10)
bipartite_mapping(g2)

## A graph containing a triangle is not bipartite
g3 <- make_ring(10)
g3 <- add_edges(g3, c(1, 3))
bipartite_mapping(g3)
```

---

bipartite_projection *Project a bipartite graph*

Description

A bipartite graph is projected into two one-mode networks

Usage

```
bipartite_projection(
  graph,
  types = NULL,
  multiplicity = TRUE,
  probe1 = NULL,
  which = c("both", "true", "false"),
  remove.type = TRUE
)

bipartite_projection_size(graph, types = NULL)
```

Arguments

graph	The input graph. It can be directed, but edge directions are ignored during the computation.
types	An optional vertex type vector to use instead of the 'type' vertex attribute. You must supply this argument if the graph has no 'type' vertex attribute.
multiplicity	If TRUE, then igraph keeps the multiplicity of the edges as an edge attribute called 'weight'. E.g. if there is an A-C-B and also an A-D-B triple in the bipartite graph (but no more X, such that A-X-B is also in the graph), then the multiplicity of the A-B edge in the projection will be 2.
probe1	This argument can be used to specify the order of the projections in the resulting list. If given, then it is considered as a vertex id (or a symbolic vertex name); the projection containing this vertex will be the first one in the result list. This argument is ignored if only one projection is requested in argument which.
which	A character scalar to specify which projection(s) to calculate. The default is to calculate both.
remove.type	Logical scalar, whether to remove the type vertex attribute from the projections. This makes sense because these graphs are not bipartite any more. However if you want to combine them with each other (or other bipartite graphs), then it is worth keeping this attribute. By default it will be removed.

Details

Bipartite graphs have a type vertex attribute in igraph, this is boolean and FALSE for the vertices of the first kind and TRUE for vertices of the second kind.

bipartite_projection_size() calculates the number of vertices and edges in the two projections of the bipartite graphs, without calculating the projections themselves. This is useful to check how much memory the projections would need if you have a large bipartite graph.

bipartite_projection() calculates the actual projections. You can use the probe1 argument to specify the order of the projections in the result. By default vertex type FALSE is the first and TRUE is the second.

bipartite_projection() keeps vertex attributes.

Value

A list of two undirected graphs. See details above.

Related documentation in the C library

edges(), vcount(), get_eids(), ecount(), bipartite_projection_size()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Bipartite graphs: bipartite_mapping(), is_bipartite(), make_bipartite_graph()

Examples

```
## Projection of a full bipartite graph is a full graph
g <- make_full_bipartite_graph(10, 5)
proj <- bipartite_projection(g)
isomorphic(proj[[1]], make_full_graph(10))
isomorphic(proj[[2]], make_full_graph(5))

## The projection keeps the vertex attributes
M <- matrix(0, nrow = 5, ncol = 3)
rownames(M) <- c("Alice", "Bob", "Cecil", "Dan", "Ethel")
colnames(M) <- c("Party", "Skiing", "Badminton")
M[] <- sample(0:1, length(M), replace = TRUE)
M
g2 <- graph_from_biadjacency_matrix(M)
g2$name <- "Event network"
proj2 <- bipartite_projection(g2)
print(proj2[[1]], g = TRUE, e = TRUE)
print(proj2[[2]], g = TRUE, e = TRUE)
```

---

c.igraph.es *Concatenate edge sequences*

Description

Concatenate edge sequences

Usage

```
## S3 method for class 'igraph.es'
c(..., recursive = FALSE)
```

Arguments

`...`	The edge sequences to concatenate. They must all refer to the same graph.
`recursive`	Ignored, included for S3 compatibility with the base `c` function.

Value

An edge sequence, the input sequences concatenated.

See Also

Other vertex and edge sequence operations: `c.igraph.vs()`, `difference.igraph.es()`, `difference.igraph.vs()`, `igraph-es-indexing`, `igraph-es-indexing2`, `igraph-vs-indexing`, `igraph-vs-indexing2`, `intersection.igraph.es`, `intersection.igraph.vs()`, `rev.igraph.es()`, `rev.igraph.vs()`, `union.igraph.es()`, `union.igraph.vs()`, `unique.igraph.es()`, `unique.igraph.vs()`

Examples

```
g <- make_(ring(10), with_vertex_(name = LETTERS[1:10]))
c(E(g)[1], E(g)["A|B"], E(g)[1:4])
```

`c.igraph.vs` *Concatenate vertex sequences*

Description

Concatenate vertex sequences

Usage

```
## S3 method for class 'igraph.vs'
c(..., recursive = FALSE)
```

Arguments

`...`	The vertex sequences to concatenate. They must refer to the same graph.
`recursive`	Ignored, included for S3 compatibility with the base `c` function.

Value

A vertex sequence, the input sequences concatenated.

See Also

Other vertex and edge sequence operations: `c.igraph.es()`, `difference.igraph.es()`, `difference.igraph.vs()`, `igraph-es-indexing`, `igraph-es-indexing2`, `igraph-vs-indexing`, `igraph-vs-indexing2`, `intersection.igraph.es`, `intersection.igraph.vs()`, `rev.igraph.es()`, `rev.igraph.vs()`, `union.igraph.es()`, `union.igraph.vs()`, `unique.igraph.es()`, `unique.igraph.vs()`

Examples

```
g <- make_(ring(10), with_vertex_(name = LETTERS[1:10]))
c(V(g)[1], V(g)["A"], V(g)[1:4])
```

`canonical_permutation` *Canonical permutation of a graph*

Description

The canonical permutation brings every isomorphic graphs into the same (labeled) graph.

Usage

```
canonical_permutation(
  graph,
  colors = NULL,
  sh = c("fm", "f", "fs", "fl", "flm", "fsm")
)
```

Arguments

`graph`	The input graph, treated as undirected.
`colors`	The colors of the individual vertices of the graph; only vertices having the same color are allowed to match each other in an automorphism. When omitted, igraph uses the `color` attribute of the vertices, or, if there is no such vertex attribute, it simply assumes that all vertices have the same color. Pass NULL explicitly if the graph has a `color` vertex attribute but you do not want to use it.
`sh`	Type of the heuristics to use for the BLISS algorithm. See details for possible values.

Details

`canonical_permutation()` computes a permutation which brings the graph into canonical form, as defined by the BLISS algorithm. All isomorphic graphs have the same canonical form.

See the paper below for the details about BLISS. This and more information is available at http://www.tcs.hut.fi/Software/bliss/index.html.

The possible values for the `sh` argument are:

"f" First non-singleton cell.

"fl" First largest non-singleton cell.

"fs" First smallest non-singleton cell.

"fm" First maximally non-trivially connectec non-singleton cell.

"flm" Largest maximally non-trivially connected non-singleton cell.

"fsm" Smallest maximally non-trivially connected non-singleton cell.

See the paper in references for details about these.

Value

A list with the following members:

labeling The canonical permutation which takes the input graph into canonical form. A numeric vector, the first element is the new label of vertex 0, the second element for vertex 1, etc.

info Some information about the BLISS computation. A named list with the following members:

"nof_nodes" The number of nodes in the search tree.

"nof_leaf_nodes" The number of leaf nodes in the search tree.

"nof_bad_nodes" Number of bad nodes.

"nof_canupdates" Number of canrep updates.

"max_level" Maximum level.

"group_size" The size of the automorphism group of the input graph, as a string. The string representation is necessary because the group size can easily exceed values that are exactly representable in floating point.

Related documentation in the C library

canonical_permutation(), vcount()

Author(s)

Tommi Junttila for BLISS, Gabor Csardi <csardi.gabor@gmail.com> for the igraph and R interfaces.

References

Tommi Junttila and Petteri Kaski: Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs, *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics.* 2007.

See Also

permute() to apply a permutation to a graph, isomorphic() for deciding graph isomorphism, possibly based on canonical labels.

Other graph isomorphism: count_isomorphisms(), count_subgraph_isomorphisms(), graph_from_isomorphism_clas
isomorphic(), isomorphism_class(), isomorphisms(), subgraph_isomorphic(), subgraph_isomorphisms()

Examples

```
## Calculate the canonical form of a random graph
g1 <- sample_gnm(10, 20)
cp1 <- canonical_permutation(g1)
cf1 <- permute(g1, cp1$labeling)

## Do the same with a random permutation of it
g2 <- permute(g1, sample(vcount(g1)))
cp2 <- canonical_permutation(g2)
cf2 <- permute(g2, cp2$labeling)
```

```
## Check that they are the same
el1 <- as_edgelist(cf1)
el2 <- as_edgelist(cf2)
el1 <- el1[order(el1[, 1], el1[, 2]), ]
el2 <- el2[order(el2[, 1], el2[, 2]), ]
all(el1 == el2)
```

`categorical_pal` *Palette for categories*

Description

This is a color blind friendly palette from https://jfly.uni-koeln.de/color/. It has 8 colors.

Usage

```
categorical_pal(n)
```

Arguments

`n`	The number of colors in the palette. We simply take the first `n` colors from the total 8.

Details

This is the suggested palette for visualizations where vertex colors mark categories, e.g. community membership.

Value

A character vector of RGB color codes.

Examples

```
library(igraphdata)
data(karate)
karate <- karate
  add_layout_(with_fr())
  set_vertex_attr("size", value = 10)

cl_k <- cluster_optimal(karate)

V(karate)$color <- membership(cl_k)
karate$palette <- categorical_pal(length(cl_k))
plot(karate)
```

See Also

Other palettes: `diverging_pal()`, `r_pal()`, `sequential_pal()`

centr_betw *Centralize a graph according to the betweenness of vertices*

Description

See centralize() for a summary of graph centralization.

Usage

```
centr_betw(graph, directed = TRUE, normalized = TRUE)
```

Arguments

graph	The input graph.
directed	logical scalar, whether to use directed shortest paths for calculating betweenness.
normalized	Logical scalar. Whether to normalize the graph level centrality score by dividing by the theoretical maximum.

Value

A named list with the following components:

res The node-level centrality scores.

centralization The graph level centrality index.

theoretical_max The maximum theoretical graph level centralization score for a graph with the given number of vertices, using the same parameters. If the normalized argument was TRUE, then the result was divided by this number.

Related documentation in the C library

centralization_betweenness()

See Also

Other centralization related: centr_betw_tmax(), centr_clo(), centr_clo_tmax(), centr_degree(), centr_degree_tmax(), centr_eigen(), centr_eigen_tmax(), centralize()

Examples

```
# A BA graph is quite centralized
g <- sample_pa(1000, m = 4)
centr_degree(g)$centralization
centr_clo(g, mode = "all")$centralization
centr_betw(g, directed = FALSE)$centralization
centr_eigen(g, directed = FALSE)$centralization
```

centr_betw_tmax *Theoretical maximum for betweenness centralization*

Description

See centralize() for a summary of graph centralization.

Usage

```
centr_betw_tmax(graph = NULL, nodes = 0, directed = TRUE)
```

Arguments

graph	The input graph. It can also be NULL if nodes and directed are both given.
nodes	The number of vertices. This is ignored if the graph is given.
directed	Logical scalar, whether to use directed shortest paths for calculating betweenness. Ignored if an undirected graph was given.

Value

Real scalar, the theoretical maximum (unnormalized) graph betweenness centrality score for graphs with given order and other parameters.

Related documentation in the C library

centralization_betweenness_tmax()

See Also

Other centralization related: centr_betw(), centr_clo(), centr_clo_tmax(), centr_degree(), centr_degree_tmax(), centr_eigen(), centr_eigen_tmax(), centralize()

Examples

```
# A BA graph is quite centralized
g <- sample_pa(1000, m = 4)
centr_betw(g, normalized = FALSE)$centralization %>%
  `/`(centr_betw_tmax(g))
centr_betw(g, normalized = TRUE)$centralization
```

centr_clo *Centralize a graph according to the closeness of vertices*

Description

See centralize() for a summary of graph centralization.

Usage

```
centr_clo(graph, mode = c("out", "in", "all", "total"), normalized = TRUE)
```

Arguments

graph	The input graph.
mode	This is the same as the mode argument of closeness().
normalized	Logical scalar. Whether to normalize the graph level centrality score by dividing by the theoretical maximum.

Value

A named list with the following components:

res The node-level centrality scores.

centralization The graph level centrality index.

theoretical_max The maximum theoretical graph level centralization score for a graph with the given number of vertices, using the same parameters. If the normalized argument was TRUE, then the result was divided by this number.

Related documentation in the C library

centralization_closeness()

See Also

Other centralization related: centr_betw(), centr_betw_tmax(), centr_clo_tmax(), centr_degree(), centr_degree_tmax(), centr_eigen(), centr_eigen_tmax(), centralize()

Examples

```
# A BA graph is quite centralized
g <- sample_pa(1000, m = 4)
centr_degree(g)$centralization
centr_clo(g, mode = "all")$centralization
centr_betw(g, directed = FALSE)$centralization
centr_eigen(g, directed = FALSE)$centralization
```

centr_clo_tmax *Theoretical maximum for closeness centralization*

Description

See centralize() for a summary of graph centralization.

Usage

```
centr_clo_tmax(graph = NULL, nodes = 0, mode = c("out", "in", "all", "total"))
```

Arguments

graph	The input graph. It can also be NULL if nodes is given.
nodes	The number of vertices. This is ignored if the graph is given.
mode	This is the same as the mode argument of closeness(). Ignored if an undirected graph is given.

Value

Real scalar, the theoretical maximum (unnormalized) graph closeness centrality score for graphs with given order and other parameters.

Related documentation in the C library

centralization_closeness_tmax()

See Also

Other centralization related: centr_betw(), centr_betw_tmax(), centr_clo(), centr_degree(), centr_degree_tmax(), centr_eigen(), centr_eigen_tmax(), centralize()

Examples

```
# A BA graph is quite centralized
g <- sample_pa(1000, m = 4)
centr_clo(g, normalized = FALSE)$centralization %>%
  `/`(centr_clo_tmax(g))
centr_clo(g, normalized = TRUE)$centralization
```

centr_degree *Centralize a graph according to the degrees of vertices*

Description

See centralize() for a summary of graph centralization.

Usage

```
centr_degree(
  graph,
  mode = c("all", "out", "in", "total"),
  loops = TRUE,
  normalized = TRUE
)
```

Arguments

graph	The input graph.
mode	This is the same as the mode argument of degree().
loops	Logical scalar, whether to consider loops edges when calculating the degree.
normalized	Logical scalar. Whether to normalize the graph level centrality score by dividing by the theoretical maximum.

Value

A named list with the following components:

res The node-level centrality scores.

centralization The graph level centrality index.

theoretical_max The maximum theoretical graph level centralization score for a graph with the given number of vertices, using the same parameters. If the normalized argument was TRUE, then the result was divided by this number.

Related documentation in the C library

centralization_degree()

See Also

Other centralization related: centr_betw(), centr_betw_tmax(), centr_clo(), centr_clo_tmax(), centr_degree_tmax(), centr_eigen(), centr_eigen_tmax(), centralize()

Examples

```
# A BA graph is quite centralized
g <- sample_pa(1000, m = 4)
centr_degree(g)$centralization
centr_clo(g, mode = "all")$centralization
centr_betw(g, directed = FALSE)$centralization
centr_eigen(g, directed = FALSE)$centralization
```

---

centr_degree_tmax *Theoretical maximum for degree centralization*

Description

See centralize() for a summary of graph centralization.

Usage

```
centr_degree_tmax(
  graph = NULL,
  nodes = 0,
  mode = c("all", "out", "in", "total"),
  loops
)
```

Arguments

graph	The input graph. It can also be NULL if nodes is given.
nodes	The number of vertices. This is ignored if the graph is given.
mode	This is the same as the mode argument of degree(). Ignored if graph is given and the graph is undirected.
loops	Logical scalar, whether to consider loops edges when calculating the degree.

Value

Real scalar, the theoretical maximum (unnormalized) graph degree centrality score for graphs with given order and other parameters.

Related documentation in the C library

centralization_degree_tmax()

See Also

Other centralization related: centr_betw(), centr_betw_tmax(), centr_clo(), centr_clo_tmax(), centr_degree(), centr_eigen(), centr_eigen_tmax(), centralize()

Examples

```
# A BA graph is quite centralized
g <- sample_pa(1000, m = 4)
centr_degree(g, normalized = FALSE)$centralization %>%
  `/`(centr_degree_tmax(g, loops = FALSE))
centr_degree(g, normalized = TRUE)$centralization
```

---

centr_eigen *Centralize a graph according to the eigenvector centrality of vertices*

Description

See `centralize()` for a summary of graph centralization.

Usage

```
centr_eigen(
  graph,
  directed = FALSE,
  scale = deprecated(),
  options = arpack_defaults(),
  normalized = TRUE
)
```

Arguments

`graph`	The input graph.
`directed`	logical scalar, whether to use directed shortest paths for calculating eigenvector centrality.
`scale`	**[Deprecated]** Ignored. Computing eigenvector centralization requires normalized eigenvector centrality scores.
`options`	This is passed to `eigen_centrality()`, the options for the ARPACK eigensolver.
`normalized`	Logical scalar. Whether to normalize the graph level centrality score by dividing by the theoretical maximum.

Value

A named list with the following components:

vector The node-level centrality scores.

value The corresponding eigenvalue.

options ARPACK options, see the return value of `eigen_centrality()` for details.

centralization The graph level centrality index.

theoretical_max The same as above, the theoretical maximum centralization score for a graph with the same number of vertices.

Related documentation in the C library

centralization_eigenvector_centrality()

See Also

Other centralization related: centr_betw(), centr_betw_tmax(), centr_clo(), centr_clo_tmax(), centr_degree(), centr_degree_tmax(), centr_eigen_tmax(), centralize()

Examples

```
# A BA graph is quite centralized
g <- sample_pa(1000, m = 4)
centr_degree(g)$centralization
centr_clo(g, mode = "all")$centralization
centr_betw(g, directed = FALSE)$centralization
centr_eigen(g, directed = FALSE)$centralization

# The most centralized graph according to eigenvector centrality
g0 <- make_graph(c(2, 1), n = 10, dir = FALSE)
g1 <- make_star(10, mode = "undirected")
centr_eigen(g0)$centralization
centr_eigen(g1)$centralization
```

---

centr_eigen_tmax *Theoretical maximum for eigenvector centralization*

Description

See centralize() for a summary of graph centralization.

Usage

```
centr_eigen_tmax(
  graph = NULL,
  nodes = 0,
  directed = FALSE,
  scale = deprecated()
)
```

Arguments

graph	The input graph. It can also be NULL, if nodes is given.
nodes	The number of vertices. This is ignored if the graph is given.
directed	logical scalar, whether to consider edge directions during the calculation. Ignored in undirected graphs.
scale	**[Deprecated]** Ignored. Computing eigenvector centralization requires normalized eigenvector centrality scores.

Value

Real scalar, the theoretical maximum (unnormalized) graph eigenvector centrality score for graphs with given vertex count and other parameters.

Related documentation in the C library

centralization_eigenvector_centrality_tmax()

See Also

Other centralization related: centr_betw(), centr_betw_tmax(), centr_clo(), centr_clo_tmax(), centr_degree(), centr_degree_tmax(), centr_eigen(), centralize()

Examples

```
# A BA graph is quite centralized
g <- sample_pa(1000, m = 4)
centr_eigen(g, normalized = FALSE)$centralization %>%
  `/`(centr_eigen_tmax(g))
centr_eigen(g, normalized = TRUE)$centralization
```

---

centralize *Centralization of a graph*

Description

Centralization is a method for creating a graph level centralization measure from the centrality scores of the vertices.

Usage

```
centralize(scores, theoretical.max = 0, normalized = TRUE)
```

Arguments

scores	The vertex level centrality scores.
theoretical.max	Real scalar. The graph-level centralization measure of the most centralized graph with the same number of vertices as the graph under study. This is only used if the normalized argument is set to TRUE.
normalized	Logical scalar. Whether to normalize the graph level centrality score by dividing by the supplied theoretical maximum.

Details

Centralization is a general method for calculating a graph-level centrality score based on node-level centrality measure. The formula for this is

$$C(G) = \sum_v (\max_w c_w - c_v),$$

where c_v is the centrality of vertex v.

The graph-level centralization measure can be normalized by dividing by the maximum theoretical score for a graph with the same number of vertices, using the same parameters, e.g. directedness, whether we consider loop edges, etc.

For degree, closeness and betweenness the most centralized structure is some version of the star graph, in-star, out-star or undirected star.

For eigenvector centrality the most centralized structure is the graph with a single edge (and potentially many isolates).

`centralize()` implements general centralization formula to calculate a graph-level score from vertex-level scores.

Value

A real scalar, the centralization of the graph from which `scores` were derived.

Related documentation in the C library

`centralization()`

References

Freeman, L.C. (1979). Centrality in Social Networks I: Conceptual Clarification. *Social Networks* 1, 215–239.

Wasserman, S., and Faust, K. (1994). *Social Network Analysis: Methods and Applications.* Cambridge University Press.

See Also

Other centralization related: `centr_betw()`, `centr_betw_tmax()`, `centr_clo()`, `centr_clo_tmax()`, `centr_degree()`, `centr_degree_tmax()`, `centr_eigen()`, `centr_eigen_tmax()`

Examples

```
# A BA graph is quite centralized
g <- sample_pa(1000, m = 4)
centr_degree(g)$centralization
centr_clo(g, mode = "all")$centralization
centr_eigen(g, directed = FALSE)$centralization

# Calculate centralization from pre-computed scores
deg <- degree(g)
tmax <- centr_degree_tmax(g, loops = FALSE)
```

```
centralize(deg, tmax)

# The most centralized graph according to eigenvector centrality
g0 <- make_graph(c(2, 1), n = 10, dir = FALSE)
g1 <- make_star(10, mode = "undirected")
centr_eigen(g0)$centralization
centr_eigen(g1)$centralization
```

`cliques` *Functions to find cliques, i.e. complete subgraphs in a graph*

Description

These functions find all, the largest or all the maximal cliques in an undirected graph. The size of the largest clique can also be calculated.

Tests if all pairs within a set of vertices are adjacent, i.e. whether they form a clique. An empty set and singleton set are considered to be a clique.

Usage

```
cliques(graph, min = NULL, max = NULL, ..., callback = NULL)

largest_cliques(graph)

max_cliques(
  graph,
  min = NULL,
  max = NULL,
  subset = NULL,
  file = NULL,
  ...,
  callback = NULL
)

count_max_cliques(graph, min = NULL, max = NULL, subset = NULL)

clique_num(graph)

largest_weighted_cliques(graph, vertex.weights = NULL)

weighted_clique_num(graph, vertex.weights = NULL)

clique_size_counts(graph, min = 0, max = 0, maximal = FALSE)

is_clique(graph, candidate, directed = FALSE)
```

Arguments

graph	The input graph.
min	Numeric constant, lower limit on the size of the cliques to find. NULL means no limit, i.e. it is the same as 0.
max	Numeric constant, upper limit on the size of the cliques to find. NULL means no limit.
...	These dots are for future extensions and must be empty.
callback	Optional function to call for each clique found. If provided, the function should accept one argument: clique (integer vector of vertex IDs in the clique, 1-based indexing). The function should return FALSE to continue the search or TRUE to stop it. If NULL (the default), all cliques are collected and returned as a list. **Important limitation:** Callback functions must NOT call any igraph functions (including simple queries like vcount() or ecount()). Doing so will cause R to crash due to reentrancy issues. Extract any needed graph information before calling the function with a callback, or use collector mode (the default) and process results afterward.
subset	If not NULL, then it must be a vector of vertex ids, numeric or symbolic if the graph is named. The algorithm is run from these vertices only, so only a subset of all maximal cliques is returned. See the Eppstein paper for details. This argument makes it possible to easily parallelize the finding of maximal cliques.
file	If not NULL, then it must be a file name, i.e. a character scalar. The output of the algorithm is written to this file. (If it exists, then it will be overwritten.) Each clique will be a separate line in the file, given with the numeric ids of its vertices, separated by whitespace.
vertex.weights	Vertex weight vector. If the graph has a weight vertex attribute, then this is used by default. If the graph does not have a weight vertex attribute and this argument is NULL, then every vertex is assumed to have a weight of 1. Note that the current implementation of the weighted clique finder supports positive integer weights only.
maximal	Specifies whether to look for all weighted cliques (FALSE) or only the maximal ones (TRUE).
candidate	The vertex set to test for being a clique.
directed	Whether to consider edge directions.

Details

cliques() find all complete subgraphs in the input graph, obeying the size limitations given in the min and max arguments.

largest_cliques() finds all largest cliques in the input graph. A clique is largest if there is no other clique including more vertices.

max_cliques() finds all maximal cliques in the input graph. A clique is maximal if it cannot be extended to a larger clique. The largest cliques are always maximal, but a maximal clique is not necessarily the largest.

count_max_cliques() counts the maximal cliques.

clique_num() calculates the size of the largest clique(s).

clique_size_counts() returns a numeric vector representing a histogram of clique sizes, between the given minimum and maximum clique size.

is_clique() tests whether all pairs within a vertex set are connected.

Value

cliques() returns a list containing numeric vectors of vertex ids if callback is NULL. Each list element is a clique, i.e. a vertex sequence of class igraph.vs. If callback is provided, returns NULL invisibly.

largest_cliques() and clique_num() return a list containing numeric vectors of vertex ids. Each list element is a clique, i.e. a vertex sequence of class igraph.vs.

max_cliques() returns NULL, invisibly, if its file argument is not NULL. The output is written to the specified file in this case.

clique_num() and count_max_cliques() return an integer scalar.

clique_size_counts() returns a numeric vector with the clique sizes such that the i-th item belongs to cliques of size i. Trailing zeros are currently truncated, but this might change in future versions.

is_clique() returns TRUE if the candidate vertex set forms a clique.

Related documentation in the C library

cliques(), cliques_callback(), vcount(), largest_cliques(), maximal_cliques_callback(), clique_number(), largest_weighted_cliques(), weighted_clique_number(), clique_size_hist(), maximal_cliques_hist(), is_clique()

Author(s)

Tamas Nepusz <ntamas@gmail.com> and Gabor Csardi <csardi.gabor@gmail.com>

References

For maximal cliques the following algorithm is implemented: David Eppstein, Maarten Loffler, Darren Strash: Listing All Maximal Cliques in Sparse Graphs in Near-optimal Time. https://arxiv.org/abs/1006.5440

See Also

Other cliques: is_complete(), ivs(), weighted_cliques()

Examples

```
# this usually contains cliques of size six
g <- sample_gnp(100, 0.3)
clique_num(g)
cliques(g, min = 6)
largest_cliques(g)
```

```
# To have a bit less maximal cliques, about 100-200 usually
g <- sample_gnp(100, 0.03)
max_cliques(g)

# Check that all returned vertex sets are indeed cliques
all(sapply(max_cliques(g), function (c) is_clique(g, c)))
```

`closeness` *Closeness centrality of vertices*

Description

Closeness centrality measures how many steps are required to access every other vertex from a given vertex.

Usage

```
closeness(
  graph,
  vids = V(graph),
  mode = c("out", "in", "all", "total"),
  weights = NULL,
  normalized = FALSE,
  cutoff = -1
)
```

Arguments

`graph`	The graph to analyze.
`vids`	The vertices for which closeness will be calculated.
`mode`	Character string, defined the types of the paths used for measuring the distance in directed graphs. "in" measures the paths *to* a vertex, "out" measures paths *from* a vertex, *all* uses undirected paths. This argument is ignored for undirected graphs.
`weights`	Optional positive weight vector for calculating weighted closeness. If the graph has a `weight` edge attribute, then this is used by default. Weights are used for calculating weighted shortest paths, so they are interpreted as distances.
`normalized`	Logical scalar, whether to calculate the normalized closeness, i.e. the inverse average distance to all reachable vertices. The non-normalized closeness is the inverse of the sum of distances to all reachable vertices.
`cutoff`	The maximum path length to consider when calculating the closeness. If zero or negative then there is no such limit.

Details

The closeness centrality of a vertex is defined as the inverse of the sum of distances to all the other vertices in the graph:

$$\frac{1}{\sum_{i \neq v} d_{vi}}$$

If there is no (directed) path between vertex `v` and `i`, then `i` is omitted from the calculation. If no other vertices are reachable from `v`, then its closeness is returned as NaN.

`cutoff` or smaller. This can be run for larger graphs, as the running time is not quadratic (if `cutoff` is small). If `cutoff` is negative (which is the default), then the function calculates the exact closeness scores. Since igraph 1.6.0, a `cutoff` value of zero is treated literally, i.e. path with a length greater than zero are ignored.

Closeness centrality is meaningful only for connected graphs. In disconnected graphs, consider using the harmonic centrality with `harmonic_centrality()`

Value

Numeric vector with the closeness values of all the vertices in `v`.

Related documentation in the C library

`closeness_cutoff()`, `vcount()`, `edges()`, `get_eids()`, `ecount()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Freeman, L.C. (1979). Centrality in Social Networks I: Conceptual Clarification. *Social Networks*, 1, 215-239.

See Also

Centrality measures: `alpha_centrality()`, `authority_score()`, `betweenness()`, `diversity()`, `eigen_centrality()`, `harmonic_centrality()`, `hits_scores()`, `page_rank()`, `power_centrality()`, `spectrum()`, `strength()`, `subgraph_centrality()`

Examples

```
g <- make_ring(10)
g2 <- make_star(10)
closeness(g)
closeness(g2, mode = "in")
closeness(g2, mode = "out")
closeness(g2, mode = "all")
```

`cluster_edge_betweenness` *Community structure detection based on edge betweenness*

Description

Community structure detection based on the betweenness of the edges in the network. This method is also known as the Girvan-Newman algorithm.

Usage

```
cluster_edge_betweenness(
  graph,
  weights = NULL,
  directed = TRUE,
  edge.betweenness = TRUE,
  merges = TRUE,
  bridges = TRUE,
  modularity = TRUE,
  membership = TRUE
)
```

Arguments

`graph`	The graph to analyze.
`weights`	The weights of the edges. It must be a positive numeric vector, `NULL` or `NA`. If it is `NULL` and the input graph has a 'weight' edge attribute, then that attribute will be used. If `NULL` and no such attribute is present, then the edges will have equal weights. Set this to `NA` if the graph was a 'weight' edge attribute, but you don't want to use it for community detection. Edge weights are used to calculate weighted edge betweenness. This means that edges are interpreted as distances, not as connection strengths.
`directed`	Logical constant, whether to calculate directed edge betweenness for directed graphs. It is ignored for undirected graphs.
`edge.betweenness`	Logical constant, whether to return the edge betweenness of the edges at the time of their removal.
`merges`	Logical constant, whether to return the merge matrix representing the hierarchical community structure of the network. This argument is called `merges`, even if the community structure algorithm itself is divisive and not agglomerative: it builds the tree from top to bottom. There is one line for each merge (i.e. split) in matrix, the first line is the first merge (last split). The communities are identified by integer number starting from one. Community ids smaller than or equal to N, the number of vertices in the graph, belong to singleton communities, i.e. individual vertices. Before the first merge we have N communities numbered from one to N. The first merge, the first line of the matrix creates community $N + 1$, the second merge creates community $N + 2$, etc.

bridges	Logical constant, whether to return a list the edge removals which actually splitted a component of the graph.
modularity	Logical constant, whether to calculate the maximum modularity score, considering all possibly community structures along the edge-betweenness based edge removals.
membership	Logical constant, whether to calculate the membership vector corresponding to the highest possible modularity score.

Details

The idea behind this method is that the betweenness of the edges connecting two communities is typically high, as many of the shortest paths between vertices in separate communities pass through them. The algorithm successively removes edges with the highest betweenness, recalculating betweenness values after each removal. This way eventually the network splits into two components, then one of these components splits again, and so on, until all edges are removed. The resulting hierarhical partitioning of the vertices can be encoded as a dendrogram.

`cluster_edge_betweenness()` returns various information collected through the run of the algorithm. Specifically, `removed.edges` contains the edge IDs in order of the edges' removal; `edge.betweenness` contains the betweenness of each of these at the time of their removal; and `bridges` contains the IDs of edges whose removal caused a split.

Value

`cluster_edge_betweenness()` returns a `communities()` object, please see the `communities()` manual page for details.

Related documentation in the C library

`vcount()`, `edges()`, `get_eids()`, `ecount()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

M Newman and M Girvan: Finding and evaluating community structure in networks, *Physical Review E* 69, 026113 (2004)

See Also

`edge_betweenness()` for the definition and calculation of the edge betweenness, `cluster_walktrap()`, `cluster_fast_greedy()`, `cluster_leading_eigen()` for other community detection methods.

See `communities()` for extracting the results of the community detection.

Community detection: `as_membership()`, `cluster_fast_greedy()`, `cluster_fluid_communities()`, `cluster_infomap()`, `cluster_label_prop()`, `cluster_leading_eigen()`, `cluster_leiden()`, `cluster_louvain()`, `cluster_optimal()`, `cluster_spinglass()`, `cluster_walktrap()`, `compare()`, `groups()`, `make_clusters()`, `membership()`, `modularity.igraph()`, `plot_dendrogram()`, `split_join_distance()`, `voronoi_cells()`

Examples

```
g <- sample_pa(100, m = 2, directed = FALSE)
eb <- cluster_edge_betweenness(g)

g <- make_full_graph(10) %du% make_full_graph(10)
g <- add_edges(g, c(1, 11))
eb <- cluster_edge_betweenness(g)
eb
```

---

cluster_fast_greedy *Community structure via greedy optimization of modularity*

Description

This function tries to find dense subgraph, also called communities in graphs via directly optimizing a modularity score.

Usage

```
cluster_fast_greedy(
  graph,
  merges = TRUE,
  modularity = TRUE,
  membership = TRUE,
  weights = NULL
)
```

Arguments

graph	The input graph. It must be undirected and must not have multi-edges.
merges	Logical scalar, whether to return the merge matrix.
modularity	Logical scalar, whether to return a vector containing the modularity after each merge.
membership	Logical scalar, whether to calculate the membership vector corresponding to the maximum modularity score, considering all possible community structures along the merges.
weights	The weights of the edges. It must be a positive numeric vector, NULL or NA. If it is NULL and the input graph has a 'weight' edge attribute, then that attribute will be used. If NULL and no such attribute is present, then the edges will have equal weights. Set this to NA if the graph was a 'weight' edge attribute, but you don't want to use it for community detection. A larger edge weight means a stronger connection for this function.

Details

This function implements the fast greedy modularity optimization algorithm for finding community structure, see A Clauset, MEJ Newman, C Moore: Finding community structure in very large networks, http://www.arxiv.org/abs/cond-mat/0408187 for the details.

Value

`cluster_fast_greedy()` returns a `communities()` object, please see the `communities()` manual page for details.

Related documentation in the C library

`vcount()`, `edges()`, `get_eids()`, `ecount()`

Author(s)

Tamas Nepusz <ntamas@gmail.com> and Gabor Csardi <csardi.gabor@gmail.com> for the R interface.

References

A Clauset, MEJ Newman, C Moore: Finding community structure in very large networks, http://www.arxiv.org/abs/cond-mat/0408187

See Also

`communities()` for extracting the results.

See also `cluster_walktrap()`, `cluster_spinglass()`, `cluster_leading_eigen()` and `cluster_edge_betweenness()`, `cluster_louvain()` `cluster_leiden()` for other methods.

Community detection: `as_membership()`, `cluster_edge_betweenness()`, `cluster_fluid_communities()`, `cluster_infomap()`, `cluster_label_prop()`, `cluster_leading_eigen()`, `cluster_leiden()`, `cluster_louvain()`, `cluster_optimal()`, `cluster_spinglass()`, `cluster_walktrap()`, `compare()`, `groups()`, `make_clusters()`, `membership()`, `modularity.igraph()`, `plot_dendrogram()`, `split_join_distance()`, `voronoi_cells()`

Examples

```
g <- make_full_graph(5) %du% make_full_graph(5) %du% make_full_graph(5)
g <- add_edges(g, c(1, 6, 1, 11, 6, 11))
fc <- cluster_fast_greedy(g)
membership(fc)
sizes(fc)
```

`cluster_fluid_communities` *Community detection algorithm based on interacting fluids*

Description

The algorithm detects communities based on the simple idea of several fluids interacting in a non-homogeneous environment (the graph topology), expanding and contracting based on their interaction and density.

Usage

```
cluster_fluid_communities(graph, no.of.communities)
```

Arguments

`graph`	The input graph. The graph must be simple and connected. Empty graphs are not supported as well as single vertex graphs. Edge directions are ignored. Weights are not considered.
`no.of.communities`	The number of communities to be found. Must be greater than 0 and fewer than number of vertices in the graph.

Value

`cluster_fluid_communities()` returns a `communities()` object, please see the `communities()` manual page for details.

Related documentation in the C library

`community_fluid_communities()`, `vcount()`

Author(s)

Ferran Parés

References

Parés F, Gasulla DG, et. al. (2018) Fluid Communities: A Competitive, Scalable and Diverse Community Detection Algorithm. In: Complex Networks & Their Applications VI: Proceedings of Complex Networks 2017 (The Sixth International Conference on Complex Networks and Their Applications), Springer, vol 689, p 229, doi: 10.1007/978-3-319-72150-7_19

See Also

See communities() for extracting the membership, modularity scores, etc. from the results.

Other community detection algorithms: cluster_walktrap(), cluster_spinglass(), cluster_leading_eigen(), cluster_edge_betweenness(), cluster_fast_greedy(), cluster_label_prop() cluster_louvain(), cluster_leiden()

Community detection: as_membership(), cluster_edge_betweenness(), cluster_fast_greedy(), cluster_infomap(), cluster_label_prop(), cluster_leading_eigen(), cluster_leiden(), cluster_louvain(), cluster_optimal(), cluster_spinglass(), cluster_walktrap(), compare(), groups(), make_clusters(), membership(), modularity.igraph(), plot_dendrogram(), split_join_distance(), voronoi_cells()

Examples

```
g <- make_graph("Zachary")
comms <- cluster_fluid_communities(g, 2)
```

---

cluster_infomap *Infomap community finding*

Description

Find community structure that minimizes the expected description length of a random walker trajectory. If the graph is directed, edge directions will be taken into account.

Usage

```
cluster_infomap(
  graph,
  e.weights = NULL,
  v.weights = NULL,
  nb.trials = 10,
  modularity = TRUE
)
```

Arguments

graph	The input graph. Edge directions will be taken into account.
e.weights	If not NULL, then a numeric vector of edge weights. The length must match the number of edges in the graph. By default the 'weight' edge attribute is used as weights. If it is not present, then all edges are considered to have the same weight. Larger edge weights correspond to stronger connections.
v.weights	If not NULL, then a numeric vector of vertex weights. The length must match the number of vertices in the graph. By default the 'weight' vertex attribute is used as weights. If it is not present, then all vertices are considered to have the same weight. A larger vertex weight means a larger probability that the random surfer jumps to that vertex.

nb.trials	The number of attempts to partition the network (can be any integer value equal or larger than 1).
modularity	Logical scalar, whether to calculate the modularity score of the detected community structure.

Details

Please see the details of this method in the references given below.

Value

`cluster_infomap()` returns a `communities()` object, please see the `communities()` manual page for details.

Related documentation in the C library

`community_infomap()`, `vcount()`, `edges()`, `get_eids()`, `ecount()`

Author(s)

Martin Rosvall wrote the original C++ code. This was ported to be more igraph-like by Emmanuel Navarro. The R interface and some cosmetics was done by Gabor Csardi <csardi.gabor@gmail.com>.

References

The original paper: M. Rosvall and C. T. Bergstrom, Maps of information flow reveal community structure in complex networks, *PNAS* 105, 1118 (2008) doi:10.1073/pnas.0706851105, https://arxiv.org/abs/0707.0609

A more detailed paper: M. Rosvall, D. Axelsson, and C. T. Bergstrom, The map equation, *Eur. Phys. J. Special Topics* 178, 13 (2009). doi:10.1140/epjst/e2010011791, https://arxiv.org/abs/0906.1405.

See Also

Other community finding methods and `communities()`.

Community detection: `as_membership()`, `cluster_edge_betweenness()`, `cluster_fast_greedy()`, `cluster_fluid_communities()`, `cluster_label_prop()`, `cluster_leading_eigen()`, `cluster_leiden()`, `cluster_louvain()`, `cluster_optimal()`, `cluster_spinglass()`, `cluster_walktrap()`, `compare()`, `groups()`, `make_clusters()`, `membership()`, `modularity.igraph()`, `plot_dendrogram()`, `split_join_distance()`, `voronoi_cells()`

Examples

```
## Zachary's karate club
g <- make_graph("Zachary")

imc <- cluster_infomap(g)
membership(imc)
communities(imc)
```

`cluster_label_prop` *Finding communities based on propagating labels*

Description

This is a fast, nearly linear time algorithm for detecting community structure in networks. In works by labeling the vertices with unique labels and then updating the labels by majority voting in the neighborhood of the vertex.

Usage

```
cluster_label_prop(
  graph,
  weights = NULL,
  ...,
  mode = c("out", "in", "all"),
  initial = NULL,
  fixed = NULL
)
```

Arguments

`graph`	The input graph. Note that the algorithm was originally defined for undirected graphs. You are advised to set 'mode' to `all` if you pass a directed graph here to treat it as undirected.
`weights`	The weights of the edges. It must be a positive numeric vector, `NULL` or `NA`. If it is `NULL` and the input graph has a 'weight' edge attribute, then that attribute will be used. If `NULL` and no such attribute is present, then the edges will have equal weights. Set this to `NA` if the graph was a 'weight' edge attribute, but you don't want to use it for community detection. A larger edge weight means a stronger connection for this function.
`...`	These dots are for future extensions and must be empty.
`mode`	Logical, whether to consider edge directions for the label propagation, and if so, in which direction the labels should propagate. Ignored for undirected graphs. "all" means to ignore edge directions (even in directed graphs). "out" means to propagate labels along the natural direction of the edges. "in" means to propagate labels backwards (i.e. from head to tail).
`initial`	The initial state. If `NULL`, every vertex will have a different label at the beginning. Otherwise it must be a vector with an entry for each vertex. Non-negative values denote different labels, negative entries denote vertices without labels.
`fixed`	Logical vector denoting which labels are fixed. Of course this makes sense only if you provided an initial state, otherwise this element will be ignored. Also note that vertices without labels cannot be fixed.

Details

This function implements the community detection method described in: Raghavan, U.N. and Albert, R. and Kumara, S.: Near linear time algorithm to detect community structures in large-scale networks. Phys Rev E 76, 036106. (2007). This version extends the original method by the ability to take edge weights into consideration and also by allowing some labels to be fixed.

From the abstract of the paper: "In our algorithm every node is initialized with a unique label and at every step each node adopts the label that most of its neighbors currently have. In this iterative process densely connected groups of nodes form a consensus on a unique label to form communities."

Value

`cluster_label_prop()` returns a `communities()` object, please see the `communities()` manual page for details.

Related documentation in the C library

`community_label_propagation()`, `vcount()`, `edges()`, `get_eids()`, `ecount()`

Author(s)

Tamas Nepusz <ntamas@gmail.com> for the C implementation, Gabor Csardi <csardi.gabor@gmail.com> for this manual page.

References

Raghavan, U.N. and Albert, R. and Kumara, S.: Near linear time algorithm to detect community structures in large-scale networks. *Phys Rev E* 76, 036106. (2007)

See Also

`communities()` for extracting the actual results.

`cluster_fast_greedy()`, `cluster_walktrap()`, `cluster_spinglass()`, `cluster_louvain()` and `cluster_leiden()` for other community detection methods.

Community detection: `as_membership()`, `cluster_edge_betweenness()`, `cluster_fast_greedy()`, `cluster_fluid_communities()`, `cluster_infomap()`, `cluster_leading_eigen()`, `cluster_leiden()`, `cluster_louvain()`, `cluster_optimal()`, `cluster_spinglass()`, `cluster_walktrap()`, `compare()`, `groups()`, `make_clusters()`, `membership()`, `modularity.igraph()`, `plot_dendrogram()`, `split_join_distance()`, `voronoi_cells()`

Examples

```
g <- sample_gnp(10, 5 / 10) %du% sample_gnp(9, 5 / 9)
g <- add_edges(g, c(1, 12))
cluster_label_prop(g)
```

`cluster_leading_eigen` *Community structure detecting based on the leading eigenvector of the community matrix*

Description

This function tries to find densely connected subgraphs in a graph by calculating the leading non-negative eigenvector of the modularity matrix of the graph.

Usage

```
cluster_leading_eigen(
  graph,
  steps = -1,
  weights = NULL,
  start = NULL,
  options = arpack_defaults(),
  callback = NULL,
  extra = NULL,
  env = parent.frame()
)
```

Arguments

`graph`	The input graph. Should be undirected as the method needs a symmetric matrix.
`steps`	The number of steps to take, this is actually the number of tries to make a step. It is not a particularly useful parameter.
`weights`	The weights of the edges. It must be a positive numeric vector, `NULL` or `NA`. If it is `NULL` and the input graph has a 'weight' edge attribute, then that attribute will be used. If `NULL` and no such attribute is present, then the edges will have equal weights. Set this to `NA` if the graph was a 'weight' edge attribute, but you don't want to use it for community detection. A larger edge weight means a stronger connection for this function.
`start`	`NULL`, or a numeric membership vector, giving the start configuration of the algorithm.
`options`	A named list to override some ARPACK options.
`callback`	If not `NULL`, then it must be callback function. This is called after each iteration, after calculating the leading eigenvector of the modularity matrix. See details below.
`extra`	Additional argument to supply to the callback function.
`env`	The environment in which the callback function is evaluated.

Details

The function documented in these section implements the 'leading eigenvector' method developed by Mark Newman, see the reference below.

The heart of the method is the definition of the modularity matrix, `B`, which is `B=A-P`, `A` being the adjacency matrix of the (undirected) network, and `P` contains the probability that certain edges are present according to the 'configuration model'. In other words, a `P[i,j]` element of `P` is the probability that there is an edge between vertices `i` and `j` in a random network in which the degrees of all vertices are the same as in the input graph.

The leading eigenvector method works by calculating the eigenvector of the modularity matrix for the largest positive eigenvalue and then separating vertices into two community based on the sign of the corresponding element in the eigenvector. If all elements in the eigenvector are of the same sign that means that the network has no underlying comuunity structure. Check Newman's paper to understand why this is a good method for detecting community structure.

Value

`cluster_leading_eigen()` returns a named list with the following members:

membership The membership vector at the end of the algorithm, when no more splits are possible.

merges The merges matrix starting from the state described by the `membership` member. This is a two-column matrix and each line describes a merge of two communities, the first line is the first merge and it creates community 'N', `N` is the number of initial communities in the graph, the second line creates community `N+1`, etc.

options Information about the underlying ARPACK computation, see `arpack()` for details.

Callback functions

The `callback` argument can be used to supply a function that is called after each eigenvector calculation. The following arguments are supplied to this function:

membership The actual membership vector, with zero-based indexing.

community The community that the algorithm just tried to split, community numbering starts with zero here.

value The eigenvalue belonging to the leading eigenvector the algorithm just found.

vector The leading eigenvector the algorithm just found.

multiplier An R function that can be used to multiple the actual modularity matrix with an arbitrary vector. Supply the vector as an argument to perform this multiplication. This function can be used with ARPACK.

extra The `extra` argument that was passed to `cluster_leading_eigen()`.

The callback function should return a scalar number. If this number is non-zero, then the clustering is terminated.

Related documentation in the C library

`community_leading_eigenvector()`, `vcount()`, `edges()`, `get_eids()`, `ecount()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

MEJ Newman: Finding community structure using the eigenvectors of matrices, Physical Review E 74 036104, 2006.

See Also

modularity(), cluster_walktrap(), cluster_edge_betweenness(), cluster_fast_greedy(), as.dendrogram()

Community detection: as_membership(), cluster_edge_betweenness(), cluster_fast_greedy(), cluster_fluid_communities(), cluster_infomap(), cluster_label_prop(), cluster_leiden(), cluster_louvain(), cluster_optimal(), cluster_spinglass(), cluster_walktrap(), compare(), groups(), make_clusters(), membership(), modularity.igraph(), plot_dendrogram(), split_join_distance(), voronoi_cells()

Examples

```
g <- make_full_graph(5) %du% make_full_graph(5) %du% make_full_graph(5)
g <- add_edges(g, c(1, 6, 1, 11, 6, 11))
lec <- cluster_leading_eigen(g)
lec

cluster_leading_eigen(g, start = membership(lec))
```

---

cluster_leiden *Finding community structure of a graph using the Leiden algorithm of Traag, van Eck & Waltman.*

Description

The Leiden algorithm is similar to the Louvain algorithm, cluster_louvain(), but it is faster and yields higher quality solutions. It can optimize both modularity and the Constant Potts Model, which does not suffer from the resolution-limit (see preprint https://arxiv.org/abs/1104.3083).

Usage

```
cluster_leiden(
  graph,
  objective_function = c("CPM", "modularity"),
  ...,
  weights = NULL,
  resolution = 1,
  resolution_parameter = deprecated(),
```

```
  beta = 0.01,
  initial_membership = NULL,
  n_iterations = 2,
  vertex_weights = NULL
)
```

Arguments

graph	The input graph. It must be undirected.
objective_function	Whether to use the Constant Potts Model (CPM) or modularity. Must be either "CPM" or "modularity".
...	These dots are for future extensions and must be empty.
weights	The weights of the edges. It must be a positive numeric vector, NULL or NA. If it is NULL and the input graph has a 'weight' edge attribute, then that attribute will be used. If NULL and no such attribute is present, then the edges will have equal weights. Set this to NA if the graph was a 'weight' edge attribute, but you don't want to use it for community detection. A larger edge weight means a stronger connection for this function.
resolution	The resolution parameter to use. Higher resolutions lead to more smaller communities, while lower resolutions lead to fewer larger communities.
resolution_parameter	**[Superseded]** Use resolution instead.
beta	Parameter affecting the randomness in the Leiden algorithm. This affects only the refinement step of the algorithm.
initial_membership	If provided, the Leiden algorithm will try to improve this provided membership. If no argument is provided, the aglorithm simply starts from the singleton partition.
n_iterations	the number of iterations to iterate the Leiden algorithm. Each iteration may improve the partition further.
vertex_weights	the vertex weights used in the Leiden algorithm. If this is not provided, it will be automatically determined on the basis of the objective_function. Please see the details of this function how to interpret the vertex weights.

Details

The Leiden algorithm consists of three phases: (1) local moving of nodes, (2) refinement of the partition and (3) aggregation of the network based on the refined partition, using the non-refined partition to create an initial partition for the aggregate network. In the local move procedure in the Leiden algorithm, only nodes whose neighborhood has changed are visited. The refinement is done by restarting from a singleton partition within each cluster and gradually merging the subclusters. When aggregating, a single cluster may then be represented by several nodes (which are the subclusters identified in the refinement).

The Leiden algorithm provides several guarantees. The Leiden algorithm is typically iterated: the output of one iteration is used as the input for the next iteration. At each iteration all clusters are

guaranteed to be connected and well-separated. After an iteration in which nothing has changed, all nodes and some parts are guaranteed to be locally optimally assigned. Finally, asymptotically, all subsets of all clusters are guaranteed to be locally optimally assigned. For more details, please see Traag, Waltman & van Eck (2019).

The objective function being optimized is

$$\frac{1}{2m}\sum_{ij}(A_{ij}-\gamma n_i n_j)\delta(\sigma_i,\sigma_j)$$

where m is the total edge weight, A_{ij} is the weight of edge (i,j), γ is the so-called resolution parameter, n_i is the node weight of node i, σ_i is the cluster of node i and $\delta(x,y)=1$ if and only if $x=y$ and 0 otherwise. By setting $n_i=k_i$, the degree of node i, and dividing γ by $2m$, you effectively obtain an expression for modularity.

Hence, the standard modularity will be optimized when you supply the degrees as `vertex_weights` and by supplying as a resolution parameter $\frac{1}{2m}$, with m the number of edges. If you do not specify any `vertex_weights`, the correct vertex weights and scaling of γ is determined automatically by the `objective_function` argument.

Value

`cluster_leiden()` returns a `communities()` object, please see the `communities()` manual page for details.

Related documentation in the C library

`community_leiden()`, `strength()`, `vcount()`, `edges()`, `get_eids()`, `ecount()`

Author(s)

Vincent Traag

References

Traag, V. A., Waltman, L., & van Eck, N. J. (2019). From Louvain to Leiden: guaranteeing well-connected communities. Scientific reports, 9(1), 5233. doi: 10.1038/s41598-019-41695-z, arXiv:1810.08473v3 [cs.SI]

See Also

See `communities()` for extracting the membership, modularity scores, etc. from the results.

Other community detection algorithms: `cluster_walktrap()`, `cluster_spinglass()`, `cluster_leading_eigen()`, `cluster_edge_betweenness()`, `cluster_fast_greedy()`, `cluster_label_prop()` `cluster_louvain()` `cluster_fluid_communities()` `cluster_infomap()` `cluster_optimal()` `cluster_walktrap()`

Community detection: `as_membership()`, `cluster_edge_betweenness()`, `cluster_fast_greedy()`, `cluster_fluid_communities()`, `cluster_infomap()`, `cluster_label_prop()`, `cluster_leading_eigen()`, `cluster_louvain()`, `cluster_optimal()`, `cluster_spinglass()`, `cluster_walktrap()`, `compare()`, `groups()`, `make_clusters()`, `membership()`, `modularity.igraph()`, `plot_dendrogram()`, `split_join_distance()`, `voronoi_cells()`

Examples

```
g <- make_graph("Zachary")
# By default CPM is used
r <- quantile(strength(g))[2] / (gorder(g) - 1)
# Set seed for sake of reproducibility
set.seed(1)
ldc <- cluster_leiden(g, resolution = r)
print(ldc)
plot(ldc, g)
```

`cluster_louvain` *Finding community structure by multi-level optimization of modularity*

Description

This function implements the multi-level modularity optimization algorithm for finding community structure, see references below. It is based on the modularity measure and a hierarchical approach.

Usage

```
cluster_louvain(graph, weights = NULL, resolution = 1)
```

Arguments

`graph`	The input graph. It must be undirected.
`weights`	The weights of the edges. It must be a positive numeric vector, `NULL` or `NA`. If it is `NULL` and the input graph has a 'weight' edge attribute, then that attribute will be used. If `NULL` and no such attribute is present, then the edges will have equal weights. Set this to `NA` if the graph was a 'weight' edge attribute, but you don't want to use it for community detection. A larger edge weight means a stronger connection for this function.
`resolution`	Optional resolution parameter that allows the user to adjust the resolution parameter of the modularity function that the algorithm uses internally. Lower values typically yield fewer, larger clusters. The original definition of modularity is recovered when the resolution parameter is set to 1.

Details

This function implements the multi-level modularity optimization algorithm for finding community structure, see VD Blondel, J-L Guillaume, R Lambiotte and E Lefebvre: Fast unfolding of community hierarchies in large networks, https://arxiv.org/abs/0803.0476 for the details.

It is based on the modularity measure and a hierarchical approach. Initially, each vertex is assigned to a community on its own. In every step, vertices are re-assigned to communities in a local, greedy way: each vertex is moved to the community with which it achieves the highest contribution to modularity. When no vertices can be reassigned, each community is considered a vertex on its own, and the process starts again with the merged communities. The process stops when there is only a

single vertex left or when the modularity cannot be increased any more in a step. Since igraph 1.3, vertices are processed in a random order.

This function was contributed by Tom Gregorovic.

Value

`cluster_louvain()` returns a `communities()` object, please see the `communities()` manual page for details.

Related documentation in the C library

`community_multilevel()`, `vcount()`, `edges()`, `get_eids()`, `ecount()`

Author(s)

Tom Gregorovic, Tamas Nepusz <ntamas@gmail.com>

References

Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, Etienne Lefebvre: Fast unfolding of communities in large networks. J. Stat. Mech. (2008) P10008

See Also

See `communities()` for extracting the membership, modularity scores, etc. from the results.

Other community detection algorithms: `cluster_walktrap()`, `cluster_spinglass()`, `cluster_leading_eigen()`, `cluster_edge_betweenness()`, `cluster_fast_greedy()`, `cluster_label_prop()` `cluster_leiden()`

Community detection: `as_membership()`, `cluster_edge_betweenness()`, `cluster_fast_greedy()`, `cluster_fluid_communities()`, `cluster_infomap()`, `cluster_label_prop()`, `cluster_leading_eigen()`, `cluster_leiden()`, `cluster_optimal()`, `cluster_spinglass()`, `cluster_walktrap()`, `compare()`, `groups()`, `make_clusters()`, `membership()`, `modularity.igraph()`, `plot_dendrogram()`, `split_join_distance()`, `voronoi_cells()`

Examples

```
# This is so simple that we will have only one level
g <- make_full_graph(5) %du% make_full_graph(5) %du% make_full_graph(5)
g <- add_edges(g, c(1, 6, 1, 11, 6, 11))
cluster_louvain(g)
```

`cluster_optimal` *Optimal community structure*

Description

This function calculates the optimal community structure of a graph, by maximizing the modularity measure over all possible partitions.

Usage

```
cluster_optimal(graph, weights = NULL)
```

Arguments

`graph`	The input graph. It may be undirected or directed.
`weights`	The weights of the edges. It must be a positive numeric vector, `NULL` or `NA`. If it is `NULL` and the input graph has a 'weight' edge attribute, then that attribute will be used. If `NULL` and no such attribute is present, then the edges will have equal weights. Set this to `NA` if the graph was a 'weight' edge attribute, but you don't want to use it for community detection. A larger edge weight means a stronger connection for this function.

Details

This function calculates the optimal community structure for a graph, in terms of maximal modularity score.

The calculation is done by transforming the modularity maximization into an integer programming problem, and then calling the GLPK library to solve that. Please the reference below for details.

Note that modularity optimization is an NP-complete problem, and all known algorithms for it have exponential time complexity. This means that you probably don't want to run this function on larger graphs. Graphs with up to fifty vertices should be fine, graphs with a couple of hundred vertices might be possible.

Value

`cluster_optimal()` returns a `communities()` object, please see the `communities()` manual page for details.

Examples

```
## Zachary's karate club
g <- make_graph("Zachary")

## We put everything into a big 'try' block, in case
## igraph was compiled without GLPK support
```

```
## The calculation only takes a couple of seconds
oc <- cluster_optimal(g)

## Double check the result
print(modularity(oc))
print(modularity(g, membership(oc)))

## Compare to the greedy optimizer
fc <- cluster_fast_greedy(g)
print(modularity(fc))
```

Related documentation in the C library

community_optimal_modularity(), vcount(), edges(), get_eids(), ecount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Gorke, Martin Hoefer, Zoran Nikoloski, Dorothea Wagner: On Modularity Clustering, *IEEE Transactions on Knowledge and Data Engineering* 20(2):172-188, 2008.

See Also

communities() for the documentation of the result, modularity(). See also cluster_fast_greedy() for a fast greedy optimizer.

Community detection: as_membership(), cluster_edge_betweenness(), cluster_fast_greedy(), cluster_fluid_communities(), cluster_infomap(), cluster_label_prop(), cluster_leading_eigen(), cluster_leiden(), cluster_louvain(), cluster_spinglass(), cluster_walktrap(), compare(), groups(), make_clusters(), membership(), modularity.igraph(), plot_dendrogram(), split_join_distance(), voronoi_cells()

---

cluster_spinglass *Finding communities in graphs based on statistical meachanics*

Description

This function tries to find communities in graphs via a spin-glass model and simulated annealing.

Usage

```
cluster_spinglass(
  graph,
  weights = NULL,
  vertex = NULL,
  spins = 25,
  parupdate = FALSE,
  start.temp = 1,
  stop.temp = 0.01,
  cool.fact = 0.99,
  update.rule = c("config", "random", "simple"),
  gamma = 1,
  implementation = c("orig", "neg"),
  gamma.minus = 1
)
```

Arguments

graph	The input graph. Edge directions are ignored in directed graphs.
weights	The weights of the edges. It must be a positive numeric vector, NULL or NA. If it is NULL and the input graph has a 'weight' edge attribute, then that attribute will be used. If NULL and no such attribute is present, then the edges will have equal weights. Set this to NA if the graph was a 'weight' edge attribute, but you don't want to use it for community detection. A larger edge weight means a stronger connection for this function.
vertex	This parameter can be used to calculate the community of a given vertex without calculating all communities. Note that if this argument is present then some other arguments are ignored.
spins	Integer constant, the number of spins to use. This is the upper limit for the number of communities. It is not a problem to supply a (reasonably) big number here, in which case some spin states will be unpopulated.
parupdate	Logical constant, whether to update the spins of the vertices in parallel (synchronously) or not. This argument is ignored if the second form of the function is used (i.e. the 'vertex' argument is present). It is also not implemented in the "neg" implementation.
start.temp	Real constant, the start temperature. This argument is ignored if the second form of the function is used (i.e. the 'vertex' argument is present).
stop.temp	Real constant, the stop temperature. The simulation terminates if the temperature lowers below this level. This argument is ignored if the second form of the function is used (i.e. the 'vertex' argument is present).
cool.fact	Cooling factor for the simulated annealing. This argument is ignored if the second form of the function is used (i.e. the 'vertex' argument is present).
update.rule	Character constant giving the 'null-model' of the simulation. Possible values: "simple" and "config". "simple" uses a random graph with the same number of edges as the baseline probability and "config" uses a random graph with the same vertex degrees as the input graph.

`gamma` Real constant, the gamma argument of the algorithm. This specifies the balance between the importance of present and non-present edges in a community. Roughly, a comunity is a set of vertices having many edges inside the community and few edges outside the community. The default 1.0 value makes existing and non-existing links equally important. Smaller values make the existing links, greater values the missing links more important.

`implementation` Character scalar. Currently igraph contains two implementations for the Spinglass community finding algorithm. The faster original implementation is the default. The other implementation, that takes into account negative weights, can be chosen by supplying 'neg' here.

`gamma.minus` Real constant, the gamma.minus parameter of the algorithm. This specifies the balance between the importance of present and non-present negative weighted edges in a community. Smaller values of gamma.minus, leads to communities with lesser negative intra-connectivity. If this argument is set to zero, the algorithm reduces to a graph coloring algorithm, using the number of spins as the number of colors. This argument is ignored if the 'orig' implementation is chosen.

Details

This function tries to find communities in a graph. A community is a set of nodes with many edges inside the community and few edges between outside it (i.e. between the community itself and the rest of the graph.)

This idea is reversed for edges having a negative weight, i.e. few negative edges inside a community and many negative edges between communities. Note that only the 'neg' implementation supports negative edge weights.

The `spinglass.cummunity` function can solve two problems related to community detection. If the `vertex` argument is not given (or it is `NULL`), then the regular community detection problem is solved (approximately), i.e. partitioning the vertices into communities, by optimizing the an energy function.

If the `vertex` argument is given and it is not `NULL`, then it must be a vertex id, and the same energy function is used to find the community of the the given vertex. See also the examples below.

Value

If the `vertex` argument is not given, i.e. the first form is used then a `cluster_spinglass()` returns a `communities()` object.

If the `vertex` argument is present, i.e. the second form is used then a named list is returned with the following components:

community Numeric vector giving the ids of the vertices in the same community as `vertex`.

cohesion The cohesion score of the result, see references.

adhesion The adhesion score of the result, see references.

inner.links The number of edges within the community of `vertex`.

outer.links The number of edges between the community of `vertex` and the rest of the graph.

Related documentation in the C library

vcount(), edges(), get_eids(), ecount()

Author(s)

Jorg Reichardt for the original code and Gabor Csardi <csardi.gabor@gmail.com> for the igraph glue code.

Changes to the original function for including the possibility of negative ties were implemented by Vincent Traag (https://www.traag.net/).

References

J. Reichardt and S. Bornholdt: Statistical Mechanics of Community Detection, *Phys. Rev. E*, 74, 016110 (2006), https://arxiv.org/abs/cond-mat/0603718

M. E. J. Newman and M. Girvan: Finding and evaluating community structure in networks, *Phys. Rev. E* 69, 026113 (2004)

V.A. Traag and Jeroen Bruggeman: Community detection in networks with positive and negative links, https://arxiv.org/abs/0811.2329 (2008).

See Also

communities(), components()

Community detection: as_membership(), cluster_edge_betweenness(), cluster_fast_greedy(), cluster_fluid_communities(), cluster_infomap(), cluster_label_prop(), cluster_leading_eigen(), cluster_leiden(), cluster_louvain(), cluster_optimal(), cluster_walktrap(), compare(), groups(), make_clusters(), membership(), modularity.igraph(), plot_dendrogram(), split_join_distance(), voronoi_cells()

Examples

```
g <- sample_gnp(10, 5 / 10) %du% sample_gnp(9, 5 / 9)
g <- add_edges(g, c(1, 12))
g <- induced_subgraph(g, subcomponent(g, 1))
cluster_spinglass(g, spins = 2)
cluster_spinglass(g, vertex = 1)
```

---

cluster_walktrap *Community structure via short random walks*

Description

This function tries to find densely connected subgraphs, also called communities in a graph via random walks. The idea is that short random walks tend to stay in the same community.

Usage

```
cluster_walktrap(
  graph,
  weights = NULL,
  steps = 4,
  merges = TRUE,
  modularity = TRUE,
  membership = TRUE
)
```

Arguments

graph	The input graph. Edge directions are ignored in directed graphs.
weights	The weights of the edges. It must be a positive numeric vector, NULL or NA. If it is NULL and the input graph has a 'weight' edge attribute, then that attribute will be used. If NULL and no such attribute is present, then the edges will have equal weights. Set this to NA if the graph was a 'weight' edge attribute, but you don't want to use it for community detection. Larger edge weights increase the probability that an edge is selected by the random walker. In other words, larger edge weights correspond to stronger connections.
steps	The length of the random walks to perform.
merges	Logical scalar, whether to include the merge matrix in the result.
modularity	Logical scalar, whether to include the vector of the modularity scores in the result. If the membership argument is true, then it will always be calculated.
membership	Logical scalar, whether to calculate the membership vector for the split corresponding to the highest modularity value.

Details

This function is the implementation of the Walktrap community finding algorithm, see Pascal Pons, Matthieu Latapy: Computing communities in large networks using random walks, https://arxiv.org/abs/physics/0512106

Value

cluster_walktrap() returns a communities() object, please see the communities() manual page for details.

Related documentation in the C library

vcount(), edges(), get_eids(), ecount()

Author(s)

Pascal Pons (http://psl.pons.free.fr/) and Gabor Csardi <csardi.gabor@gmail.com> for the R and igraph interface

References

Pascal Pons, Matthieu Latapy: Computing communities in large networks using random walks, https://arxiv.org/abs/physics/0512106

See Also

See communities() on getting the actual membership vector, merge matrix, modularity score, etc.

modularity() and cluster_fast_greedy(), cluster_spinglass(), cluster_leading_eigen(), cluster_edge_betweenness(), cluster_louvain(), and cluster_leiden() for other community detection methods.

Community detection: as_membership(), cluster_edge_betweenness(), cluster_fast_greedy(), cluster_fluid_communities(), cluster_infomap(), cluster_label_prop(), cluster_leading_eigen(), cluster_leiden(), cluster_louvain(), cluster_optimal(), cluster_spinglass(), compare(), groups(), make_clusters(), membership(), modularity.igraph(), plot_dendrogram(), split_join_distance(), voronoi_cells()

Examples

```
g <- make_full_graph(5) %du% make_full_graph(5) %du% make_full_graph(5)
g <- add_edges(g, c(1, 6, 1, 11, 6, 11))
cluster_walktrap(g)
```

---

cocitation *Cocitation coupling*

Description

Two vertices are cocited if there is another vertex citing both of them. cocitation() simply counts how many types two vertices are cocited. The bibliographic coupling of two vertices is the number of other vertices they both cite, bibcoupling() calculates this.

Usage

```
cocitation(graph, v = V(graph))

bibcoupling(graph, v = V(graph))
```

Arguments

graph	The graph object to analyze
v	Vertex sequence or numeric vector, the vertex ids for which the cocitation or bibliographic coupling values we want to calculate. The default is all vertices.

Details

cocitation() calculates the cocitation counts for the vertices in the v argument and all vertices in the graph.

bibcoupling() calculates the bibliographic coupling for vertices in v and all vertices in the graph.

Calculating the cocitation or bibliographic coupling for only one vertex costs the same amount of computation as for all vertices. This might change in the future.

Value

A numeric matrix with length(v) lines and vcount(graph) columns. Element (i,j) contains the cocitation or bibliographic coupling for vertices v[i] and j.

Related documentation in the C library

cocitation(), vcount(), bibcoupling()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Other cocitation: similarity()

Examples

```
g <- make_kautz_graph(2, 3)
cocitation(g)
bibcoupling(g)
```

---

cohesive_blocks *Calculate Cohesive Blocks*

Description

Calculates cohesive blocks for objects of class igraph.

Usage

```
cohesive_blocks(graph, labels = TRUE)

## S3 method for class 'cohesiveBlocks'
length(x)

blocks(blocks)
```

```
graphs_from_cohesive_blocks(blocks, graph)

## S3 method for class 'cohesiveBlocks'
cohesion(x, ...)

hierarchy(blocks)

parent(blocks)

## S3 method for class 'cohesiveBlocks'
print(x, ...)

## S3 method for class 'cohesiveBlocks'
summary(object, ...)

## S3 method for class 'cohesiveBlocks'
plot(
  x,
  y,
  colbar = rainbow(max(cohesion(x)) + 1),
  col = colbar[max_cohesion(x) + 1],
  mark.groups = blocks(x)[-1],
  ...
)

plot_hierarchy(
  blocks,
  layout = layout_as_tree(hierarchy(blocks), root = 1),
  ...
)

export_pajek(blocks, graph, file, project.file = TRUE)

max_cohesion(blocks)
```

Arguments

graph	For cohesive_blocks() a graph object of class igraph. It must be undirected and simple. (See is_simple().) For graphs_from_cohesive_blocks() and export_pajek() the same graph must be supplied whose cohesive block structure is given in the blocks() argument.
labels	Logical scalar, whether to add the vertex labels to the result object. These labels can be then used when reporting and plotting the cohesive blocks.
blocks, x, object	A cohesiveBlocks object, created with the cohesive_blocks() function.
...	Additional arguments. plot_hierarchy() and plot() pass them to plot.igraph(). print() and summary() ignore them.

y	The graph whose cohesive blocks are supplied in the x argument.
colbar	Color bar for the vertex colors. Its length should be at least $m + 1$, where m is the maximum cohesion in the graph. Alternatively, the vertex colors can also be directly specified via the col argument.
col	A vector of vertex colors, in any of the usual formats. (Symbolic color names (e.g. 'red', 'blue', etc.) , RGB colors (e.g. '#FF9900FF'), integer numbers referring to the current palette. By default the given colbar is used and vertices with the same maximal cohesion will have the same color.
mark.groups	A list of vertex sets to mark on the plot by circling them. By default all cohesive blocks are marked, except the one corresponding to the all vertices.
layout	The layout of a plot, it is simply passed on to plot.igraph(), see the possible formats there. By default the Reingold-Tilford layout generator is used.
file	Defines the file (or connection) the Pajek file is written to. If the project.file argument is TRUE, then it can be a filename (with extension), a file object, or in general any king of connection object. The file/connection will be opened if it wasn't already. If the project.file argument is FALSE, then several files are created and file must be a character scalar containing the base name of the files, without extension. (But it can contain the path to the files.) See also details below.
project.file	Logical scalar, whether to create a single Pajek project file containing all the data, or to create separated files for each item. See details below.

Details

Cohesive blocking is a method of determining hierarchical subsets of graph vertices based on their structural cohesion (or vertex connectivity). For a given graph G, a subset of its vertices $S \subset V(G)$ is said to be maximally k-cohesive if there is no superset of S with vertex connectivity greater than or equal to k. Cohesive blocking is a process through which, given a k-cohesive set of vertices, maximally l-cohesive subsets are recursively identified with $l > k$. Thus a hierarchy of vertex subsets is found, with the entire graph G at its root.

The function cohesive_blocks() implements cohesive blocking. It returns a cohesiveBlocks object. cohesiveBlocks should be handled as an opaque class, i.e. its internal structure should not be accessed directly, but through the functions listed here.

The function length can be used on cohesiveBlocks objects and it gives the number of blocks.

The function blocks() returns the actual blocks stored in the cohesiveBlocks object. They are returned in a list of numeric vectors, each containing vertex ids.

The function graphs_from_cohesive_blocks() is similar, but returns the blocks as (induced) subgraphs of the input graph. The various (graph, vertex and edge) attributes are kept in the subgraph.

The function cohesion() returns a numeric vector, the cohesion of the different blocks. The order of the blocks is the same as for the blocks() and graphs_from_cohesive_blocks() functions.

The block hierarchy can be queried using the hierarchy() function. It returns an igraph graph, its vertex ids are ordered according the order of the blocks in the blocks() and graphs_from_cohesive_blocks(), cohesion(), etc. functions.

`parent()` gives the parent vertex of each block, in the block hierarchy, for the root vertex it gives 0.

`plot_hierarchy()` plots the hierarchy tree of the cohesive blocks on the active graphics device, by calling `igraph.plot`.

The `export_pajek()` function can be used to export the graph and its cohesive blocks in Pajek format. It can either export a single Pajek project file with all the information, or a set of files, depending on its `project.file` argument. If `project.file` is `TRUE`, then the following information is written to the file (or connection) given in the `file` argument: (1) the input graph, together with its attributes, see `write_graph()` for details; (2) the hierarchy graph; and (3) one binary partition for each cohesive block. If `project.file` is `FALSE`, then the `file` argument must be a character scalar and it is used as the base name for the generated files. If `file` is 'basename', then the following files are created: (1) 'basename.net' for the original graph; (2) 'basename_hierarchy.net' for the hierarchy graph; (3) 'basename_block_x.net' for each cohesive block, where 'x' is the number of the block, starting with one.

`max_cohesion()` returns the maximal cohesion of each vertex, i.e. the cohesion of the most cohesive block of the vertex.

The generic function `summary()` works on `cohesiveBlocks` objects and it prints a one line summary to the terminal.

The generic function `print()` is also defined on `cohesiveBlocks` objects and it is invoked automatically if the name of the `cohesiveBlocks` object is typed in. It produces an output like this:

```
 Cohesive block structure:
B-1 c 1, n 23
'- B-2 c 2, n 14 oooooooo.. .o......oo ooo
'- B-4 c 5, n  7 ooooooo... .......... ...
'- B-3 c 2, n 10 ......o.oo o.oooooo.. ...
'- B-5 c 3, n  4 ......o.oo o......... ...
```

The left part shows the block structure, in this case for five blocks. The first block always corresponds to the whole graph, even if its cohesion is zero. Then cohesion of the block and the number of vertices in the block are shown. The last part is only printed if the display is wide enough and shows the vertices in the blocks, ordered by vertex ids. 'o' means that the vertex is included, a dot means that it is not, and the vertices are shown in groups of ten.

The generic function `plot()` plots the graph, showing one or more cohesive blocks in it.

Value

`cohesive_blocks()` returns a `cohesiveBlocks` object.

`blocks()` returns a list of numeric vectors, containing vertex ids.

`graphs_from_cohesive_blocks()` returns a list of igraph graphs, corresponding to the cohesive blocks.

`cohesion()` returns a numeric vector, the cohesion of each block.

`hierarchy()` returns an igraph graph, the representation of the cohesive block hierarchy.

`parent()` returns a numeric vector giving the parent block of each cohesive block, in the block hierarchy. The block at the root of the hierarchy has no parent and `0` is returned for it.

`plot_hierarchy()`, `plot()` and `export_pajek()` return `NULL`, invisibly.

`max_cohesion()` returns a numeric vector with one entry for each vertex, giving the cohesion of its most cohesive block.

`print()` and `summary()` return the `cohesiveBlocks` object itself, invisibly.

`length` returns a numeric scalar, the number of blocks.

Related documentation in the C library

`cohesive_blocks()`, `vcount()`, `write_graph_edgelist()`, `write_graph_pajek()`, `write_graph_graphml()`, `write_graph_gml()`, `write_graph_dot()`, `write_graph_leda()`, `edges()`, `get_eids()`, `ecount()`, `is_directed()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com> for the current implementation, Peter McMahan (https://socialsciences.uchicago.edu/news/alumni-profile-peter-mcmahan-phd17-sociology) wrote the first version in R.

References

J. Moody and D. R. White. Structural cohesion and embeddedness: A hierarchical concept of social groups. *American Sociological Review*, 68(1):103–127, Feb 2003, doi:10.2307/3088904.

See Also

`cohesion()`

Examples

```
## The graph from the Moody-White paper
mw <- graph_from_literal(
  1 - 2:3:4:5:6, 2 - 3:4:5:7, 3 - 4:6:7, 4 - 5:6:7,
  5 - 6:7:21, 6 - 7, 7 - 8:11:14:19, 8 - 9:11:14, 9 - 10,
  10 - 12:13, 11 - 12:14, 12 - 16, 13 - 16, 14 - 15, 15 - 16,
  17 - 18:19:20, 18 - 20:21, 19 - 20:22:23, 20 - 21,
  21 - 22:23, 22 - 23
)

mwBlocks <- cohesive_blocks(mw)

# Inspect block membership and cohesion
mwBlocks
blocks(mwBlocks)
cohesion(mwBlocks)

# Save results in a Pajek file
file <- tempfile(fileext = ".paj")
export_pajek(mwBlocks, mw, file = file)
if (!interactive()) {
  unlink(file)
```

```
}

# Plot the results
plot(mwBlocks, mw)

## The science camp network
camp <- graph_from_literal(
  Harry:Steve:Don:Bert - Harry:Steve:Don:Bert,
  Pam:Brazey:Carol:Pat - Pam:Brazey:Carol:Pat,
  Holly - Carol:Pat:Pam:Jennie:Bill,
  Bill - Pauline:Michael:Lee:Holly,
  Pauline - Bill:Jennie:Ann,
  Jennie - Holly:Michael:Lee:Ann:Pauline,
  Michael - Bill:Jennie:Ann:Lee:John,
  Ann - Michael:Jennie:Pauline,
  Lee - Michael:Bill:Jennie,
  Gery - Pat:Steve:Russ:John,
  Russ - Steve:Bert:Gery:John,
  John - Gery:Russ:Michael
)
campBlocks <- cohesive_blocks(camp)
campBlocks

plot(campBlocks, camp,
  vertex.label = V(camp)$name, margin = -0.2,
  vertex.shape = "rectangle", vertex.size = 24, vertex.size2 = 8,
  mark.border = 1, colbar = c(NA, NA, "cyan", "orange")
)
```

`compare` *Compares community structures using various metrics*

Description

This function assesses the distance between two community structures.

Usage

```
compare(
  comm1,
  comm2,
  method = c("vi", "nmi", "split.join", "rand", "adjusted.rand")
)
```

Arguments

`comm1`	A `communities()` object containing a community structure; or a numeric vector, the membership vector of the first community structure. The membership

vector should contain the community id of each vertex, the numbering of the communities starts with one.

`comm2` A `communities()` object containing a community structure; or a numeric vector, the membership vector of the second community structure, in the same format as for the previous argument.

`method` Character scalar, the comparison method to use. Possible values: 'vi' is the variation of information (VI) metric of Meila (2003), 'nmi' is the normalized mutual information measure proposed by Danon et al. (2005), 'split.join' is the split-join distance of can Dongen (2000), 'rand' is the Rand index of Rand (1971), 'adjusted.rand' is the adjusted Rand index by Hubert and Arabie (1985).

Value

A real number.

Related documentation in the C library

`compare_communities()`

Author(s)

Tamas Nepusz <ntamas@gmail.com>

References

Meila M: Comparing clusterings by the variation of information. In: Scholkopf B, Warmuth MK (eds.). *Learning Theory and Kernel Machines: 16th Annual Conference on Computational Learning Theory and 7th Kernel Workshop*, COLT/Kernel 2003, Washington, DC, USA. Lecture Notes in Computer Science, vol. 2777, Springer, 2003. ISBN: 978-3-540-40720-1.

Danon L, Diaz-Guilera A, Duch J, Arenas A: Comparing community structure identification. *J Stat Mech* P09008, 2005.

van Dongen S: Performance criteria for graph clustering and Markov cluster experiments. Technical Report INS-R0012, National Research Institute for Mathematics and Computer Science in the Netherlands, Amsterdam, May 2000.

Rand WM: Objective criteria for the evaluation of clustering methods. *J Am Stat Assoc* 66(336):846-850, 1971.

Hubert L and Arabie P: Comparing partitions. *Journal of Classification* 2:193-218, 1985.

See Also

Community detection: `as_membership()`, `cluster_edge_betweenness()`, `cluster_fast_greedy()`, `cluster_fluid_communities()`, `cluster_infomap()`, `cluster_label_prop()`, `cluster_leading_eigen()`, `cluster_leiden()`, `cluster_louvain()`, `cluster_optimal()`, `cluster_spinglass()`, `cluster_walktrap()`, `groups()`, `make_clusters()`, `membership()`, `modularity.igraph()`, `plot_dendrogram()`, `split_join_distance()`, `voronoi_cells()`

Examples

```
g <- make_graph("Zachary")
sg <- cluster_spinglass(g)
le <- cluster_leading_eigen(g)
compare(sg, le, method = "rand")
compare(membership(sg), membership(le))
```

`complementer` *Complementer of a graph*

Description

A complementer graph contains all edges that were not present in the input graph.

Usage

```
complementer(graph, loops = FALSE)
```

Arguments

`graph`	The input graph, can be directed or undirected.
`loops`	Logical constant, whether to generate loop edges.

Details

`complementer()` creates the complementer of a graph. Only edges which are *not* present in the original graph will be included in the new graph.

`complementer()` keeps graph and vertex attriubutes, edge attributes are lost.

Value

A new graph object.

Related documentation in the C library

`complementer()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Other functions for manipulating graph structure: `+.igraph()`, `add_edges()`, `add_vertices()`, `compose()`, `connect()`, `contract()`, `delete_edges()`, `delete_vertices()`, `difference()`, `difference.igraph()`, `disjoint_union()`, `edge()`, `igraph-minus`, `intersection()`, `intersection.igraph()`, `path()`, `permute()`, `rep.igraph()`, `reverse_edges()`, `simplify()`, `transitive_closure()`, `union()`, `union.igraph()`, `vertex()`

Examples

```
## Complementer of a ring
g <- make_ring(10)
complementer(g)

## A graph and its complementer give together the full graph
g <- make_ring(10)
gc <- complementer(g)
gu <- union(g, gc)
gu
isomorphic(gu, make_full_graph(vcount(g)))
```

---

component_distribution

Connected components of a graph

Description

Calculate the maximal (weakly or strongly) connected components of a graph

Usage

```
component_distribution(graph, cumulative = FALSE, mul.size = FALSE, ...)

largest_component(graph, mode = c("weak", "strong"))

components(graph, mode = c("weak", "strong"))

is_connected(graph, mode = c("weak", "strong"))

count_components(graph, mode = c("weak", "strong"))
```

Arguments

graph	The graph to analyze.
cumulative	Logical, if TRUE the cumulative distirubution (relative frequency) is calculated.
mul.size	Logical. If TRUE the relative frequencies will be multiplied by the cluster sizes.
...	Additional attributes to pass to cluster, right now only mode makes sense.
mode	Character string, either "weak" or "strong". For directed graphs "weak" implies weakly, "strong" strongly connected components to search. It is ignored for undirected graphs.

Details

`is_connected()` decides whether the graph is weakly or strongly connected. The null graph is considered disconnected.

`components()` finds the maximal (weakly or strongly) connected components of a graph.

`count_components()` does almost the same as `components()` but returns only the number of clusters found instead of returning the actual clusters.

`component_distribution()` creates a histogram for the maximal connected component sizes.

`largest_component()` returns the largest connected component of a graph. For directed graphs, optionally the largest weakly or strongly connected component. In case of a tie, the first component by vertex ID order is returned. Vertex IDs from the original graph are not retained in the returned graph.

The weakly connected components are found by a simple breadth-first search. The strongly connected components are implemented by two consecutive depth-first searches.

Value

For `is_connected()` a logical constant.

For `components()` a named list with three components:

membership numeric vector giving the cluster id to which each vertex belongs.

csize numeric vector giving the sizes of the clusters.

no numeric constant, the number of clusters.

For `count_components()` an integer constant is returned.

For `component_distribution()` a numeric vector with the relative frequencies. The length of the vector is the size of the largest component plus one. Note that (for currently unknown reasons) the first element of the vector is the number of clusters of size zero, so this is always zero.

For `largest_component()` the largest connected component of the graph.

Related documentation in the C library

`connected_components()`, `vcount()`, `induced_subgraph()`, `is_connected()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

`decompose()`, `subcomponent()`, `groups()`

Connected components: `articulation_points()`, `biconnected_components()`, `count_reachable()`, `decompose()`, `is_biconnected()`

Other structural.properties: `bfs()`, `connect()`, `constraint()`, `coreness()`, `degree()`, `dfs()`, `distance_table()`, `edge_density()`, `feedback_arc_set()`, `feedback_vertex_set()`, `girth()`, `is_acyclic()`, `is_dag()`, `is_matching()`, `k_shortest_paths()`, `knn()`, `reciprocity()`, `subcomponent()`, `subgraph()`, `topo_sort()`, `transitivity()`, `unfold_tree()`, `which_multiple()`, `which_mutual()`

Examples

```
g <- sample_gnp(20, 1 / 20)
clu <- components(g)
groups(clu)
largest_component(g)
```

---

component_wise *Component-wise layout*

Description

This is a layout modifier function, and it can be used to calculate the layout separately for each component of the graph.

Usage

```
component_wise(merge_method = "dla")
```

Arguments

merge_method	Merging algorithm, the method argument of merge_coords().

Related documentation in the C library

decompose(), vcount()

See Also

merge_coords(), layout_().

Other layout modifiers: layout_modifier(), normalize()

Other graph layouts: add_layout_(), layout_(), layout_as_bipartite(), layout_as_star(), layout_as_tree(), layout_in_circle(), layout_nicely(), layout_on_grid(), layout_on_sphere(), layout_randomly(), layout_with_dh(), layout_with_fr(), layout_with_gem(), layout_with_graphopt(), layout_with_kk(), layout_with_lgl(), layout_with_mds(), layout_with_sugiyama(), merge_coords(), norm_coords(), normalize()

Examples

```
g <- make_ring(10) + make_ring(10)
g %>%
  add_layout_(in_circle(), component_wise()) %>%
  plot()
```

compose *Compose two graphs as binary relations*

Description

Relational composition of two graph.

Usage

```
compose(g1, g2, byname = "auto")
```

Arguments

g1	The first input graph.
g2	The second input graph.
byname	A logical scalar, or the character scalar auto. Whether to perform the operation based on symbolic vertex names. If it is auto, that means TRUE if both graphs are named and FALSE otherwise. A warning is generated if auto and one graph, but not both graphs are named.

Details

compose() creates the relational composition of two graphs. The new graph will contain an (a,b) edge only if there is a vertex c, such that edge (a,c) is included in the first graph and (c,b) is included in the second graph. The corresponding operator is %c%.

The function gives an error if one of the input graphs is directed and the other is undirected.

If the byname argument is TRUE (or auto and the graphs are all named), then the operation is performed based on symbolic vertex names. Otherwise numeric vertex ids are used.

compose() keeps the attributes of both graphs. All graph, vertex and edge attributes are copied to the result. If an attribute is present in multiple graphs and would result a name clash, then this attribute is renamed by adding suffixes: _1, _2, etc.

The name vertex attribute is treated specially if the operation is performed based on symbolic vertex names. In this case name must be present in both graphs, and it is not renamed in the result graph.

Note that an edge in the result graph corresponds to two edges in the input, one in the first graph, one in the second. This mapping is not injective and several edges in the result might correspond to the same edge in the first (and/or the second) graph. The edge attributes in the result graph are updated accordingly.

Also note that the function may generate multigraphs, if there are more than one way to find edges (a,b) in g1 and (b,c) in g2 for an edge (a,c) in the result. See simplify() if you want to get rid of the multiple edges.

The function may create loop edges, if edges (a,b) and (b,a) are present in g1 and g2, respectively, then (a,a) is included in the result. See simplify() if you want to get rid of the self-loops.

Value

A new graph object.

Related documentation in the C library

vcount(), permute_vertices(), edges(), get_eids(), ecount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Other functions for manipulating graph structure: +.igraph(), add_edges(), add_vertices(), complementer(), connect(), contract(), delete_edges(), delete_vertices(), difference(), difference.igraph(), disjoint_union(), edge(), igraph-minus, intersection(), intersection.igraph(), path(), permute(), rep.igraph(), reverse_edges(), simplify(), transitive_closure(), union(), union.igraph(), vertex()

Examples

```
g1 <- make_ring(10)
g2 <- make_star(10, mode = "undirected")
gc <- compose(g1, g2)
print_all(gc)
print_all(simplify(gc))
```

---

connect *Neighborhood of graph vertices*

Description

These functions find the vertices not farther than a given limit from another fixed vertex, these are called the neighborhood of the vertex. Note that ego() and neighborhood(), ego_size() and neighborhood_size(), make_ego_graph() and make_neighborhood()_graph(), are synonyms (aliases).

Usage

```
connect(graph, order, mode = c("all", "out", "in", "total"))

ego_size(
  graph,
  order = 1,
  nodes = V(graph),
  mode = c("all", "out", "in"),
  mindist = 0
```

```
)

neighborhood_size(
  graph,
  order = 1,
  nodes = V(graph),
  mode = c("all", "out", "in"),
  mindist = 0
)

ego(
  graph,
  order = 1,
  nodes = V(graph),
  mode = c("all", "out", "in"),
  mindist = 0
)

neighborhood(
  graph,
  order = 1,
  nodes = V(graph),
  mode = c("all", "out", "in"),
  mindist = 0
)

make_ego_graph(
  graph,
  order = 1,
  nodes = V(graph),
  mode = c("all", "out", "in"),
  mindist = 0
)

make_neighborhood_graph(
  graph,
  order = 1,
  nodes = V(graph),
  mode = c("all", "out", "in"),
  mindist = 0
)
```

Arguments

graph	The input graph.
order	Integer giving the order of the neighborhood. Negative values indicate an infinite order.
mode	Character constant, it specifies how to use the direction of the edges if a directed

graph is analyzed. For 'out' only the outgoing edges are followed, so all vertices reachable from the source vertex in at most `order` steps are counted. For '"in"' all vertices from which the source vertex is reachable in at most `order` steps are counted. '"all"' ignores the direction of the edges. This argument is ignored for undirected graphs.

`nodes` The vertices for which the calculation is performed.

`mindist` The minimum distance to include the vertex in the result.

Details

The neighborhood of a given order `r` of a vertex `v` includes all vertices which are closer to `v` than the order. I.e. order 0 is always `v` itself, order 1 is `v` plus its immediate neighbors, order 2 is order 1 plus the immediate neighbors of the vertices in order 1, etc.

`ego_size()`/`neighborhood_size()` (synonyms) returns the size of the neighborhoods of the given order, for each given vertex.

`ego()`/`neighborhood()` (synonyms) returns the vertices belonging to the neighborhoods of the given order, for each given vertex.

`make_ego_graph()`/`make_neighborhood()_graph()` (synonyms) is creates (sub)graphs from all neighborhoods of the given vertices with the given order parameter. This function preserves the vertex, edge and graph attributes.

`connect()` creates a new graph by connecting each vertex to all other vertices in its neighborhood.

Value

- `ego_size()`/`neighborhood_size()` returns with an integer vector.
- `ego()`/`neighborhood()` (synonyms) returns A list of `igraph.vs` or a list of numeric vectors depending on the value of `igraph_opt("return.vs.es")`, see details for performance characteristics.
- `make_ego_graph()`/`make_neighborhood_graph()` returns with a list of graphs.
- `connect()` returns with a new graph object.

Related documentation in the C library

`connect_neighborhood()`, `vcount()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>, the first version was done by Vincent Matossian

See Also

Other functions for manipulating graph structure: `+.igraph()`, `add_edges()`, `add_vertices()`, `complementer()`, `compose()`, `contract()`, `delete_edges()`, `delete_vertices()`, `difference()`, `difference.igraph()`, `disjoint_union()`, `edge()`, `igraph-minus`, `intersection()`, `intersection.igraph()`, `path()`, `permute()`, `rep.igraph()`, `reverse_edges()`, `simplify()`, `transitive_closure()`, `union()`, `union.igraph()`, `vertex()`

Other structural.properties: bfs(), component_distribution(), constraint(), coreness(), degree(), dfs(), distance_table(), edge_density(), feedback_arc_set(), feedback_vertex_set(), girth(), is_acyclic(), is_dag(), is_matching(), k_shortest_paths(), knn(), reciprocity(), subcomponent(), subgraph(), topo_sort(), transitivity(), unfold_tree(), which_multiple(), which_mutual()

Examples

```
g <- make_ring(10)

ego_size(g, order = 0, 1:3)
ego_size(g, order = 1, 1:3)
ego_size(g, order = 2, 1:3)

# neighborhood_size() is an alias of ego_size()
neighborhood_size(g, order = 0, 1:3)
neighborhood_size(g, order = 1, 1:3)
neighborhood_size(g, order = 2, 1:3)

ego(g, order = 0, 1:3)
ego(g, order = 1, 1:3)
ego(g, order = 2, 1:3)

# neighborhood() is an alias of ego()
neighborhood(g, order = 0, 1:3)
neighborhood(g, order = 1, 1:3)
neighborhood(g, order = 2, 1:3)

# attributes are preserved
V(g)$name <- c("a", "b", "c", "d", "e", "f", "g", "h", "i", "j")
make_ego_graph(g, order = 2, 1:3)
# make_neighborhood_graph() is an alias of make_ego_graph()
make_neighborhood_graph(g, order = 2, 1:3)

# connecting to the neighborhood
g <- make_ring(10)
g <- connect(g, 2)
```

`consensus_tree` *Create a consensus tree from several hierarchical random graph models*

Description

`consensus_tree()` creates a consensus tree from several fitted hierarchical random graph models, using phylogeny methods. If the `hrg()` argument is given and `start` is set to `TRUE`, then it starts sampling from the given HRG. Otherwise it optimizes the HRG log-likelihood first, and then samples starting from the optimum.

Usage

```
consensus_tree(graph, hrg = NULL, start = FALSE, num.samples = 10000)
```

Arguments

graph	The graph the models were fitted to.
hrg	A hierarchical random graph model, in the form of an igraphHRG object. consensus_tree() allows this to be NULL as well, then a HRG is fitted to the graph first, from a random starting point.
start	Logical, whether to start the fitting/sampling from the supplied igraphHRG object, or from a random starting point.
num.samples	Number of samples to use for consensus generation or missing edge prediction.

Value

consensus_tree() returns a list of two objects. The first is an igraphHRGConsensus object, the second is an igraphHRG object. The igraphHRGConsensus object has the following members:

parents For each vertex, the id of its parent vertex is stored, or zero, if the vertex is the root vertex in the tree. The first n vertex ids (from 0) refer to the original vertices of the graph, the other ids refer to vertex groups.

weights Numeric vector, counts the number of times a given tree split occurred in the generated network samples, for each internal vertices. The order is the same as in the parents vector.

Related documentation in the C library

hrg_consensus()

See Also

Other hierarchical random graph functions: fit_hrg(), hrg(), hrg-methods, hrg_tree(), predict_edges(), print.igraphHRG(), print.igraphHRGConsensus(), sample_hrg()

---

console *The igraph console*

Description

The igraph console is a GUI window that shows what the currently running igraph function is doing.

Usage

```
console()
```

Details

The console can be started by calling the `console()` function. Then it stays open, until the user closes it.

Another way to start it to set the `verbose` igraph option to "tkconsole" via `igraph_options()`. Then the console (re)opens each time an igraph function supporting it starts; to close it, set the `verbose` option to another value.

The console is written in Tcl/Tk and required the `tcltk` package.

Value

`NULL`, invisibly.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

`igraph_options()` and the `verbose` option.

---

constraint *Burt's constraint*

Description

Given a graph, `constraint()` calculates Burt's constraint for each vertex.

Usage

```
constraint(graph, nodes = V(graph), weights = NULL)
```

Arguments

`graph`	A graph object, the input graph.
`nodes`	The vertices for which the constraint will be calculated. Defaults to all vertices.
`weights`	The weights of the edges. If this is `NULL` and there is a `weight` edge attribute this is used. If there is no such edge attribute all edges will have the same weight.

Details

Burt's constraint is higher if ego has less, or mutually stronger related (i.e. more redundant) contacts. Burt's measure of constraint, C_i, of vertex i's ego network V_i, is defined for directed and valued graphs,

$$C_i = \sum_{j \in V_i \setminus \{i\}} (p_{ij} + \sum_{q \in V_i \setminus \{i,j\}} p_{iq} p_{qj})^2$$

for a graph of order (i.e. number of vertices) N, where proportional tie strengths are defined as

$$p_{ij} = \frac{a_{ij} + a_{ji}}{\sum_{k \in V_i \setminus \{i\}} (a_{ik} + a_{ki})},$$

a_{ij} are elements of A and the latter being the graph adjacency matrix. For isolated vertices, constraint is undefined.

Value

A numeric vector of constraint scores

Related documentation in the C library

constraint(), vcount(), edges(), get_eids(), ecount()

Author(s)

Jeroen Bruggeman (https://sites.google.com/site/jebrug/jeroen-bruggeman-social-science) and Gabor Csardi <csardi.gabor@gmail.com>

References

Burt, R.S. (2004). Structural holes and good ideas. *American Journal of Sociology* 110, 349-399.

See Also

Other structural.properties: bfs(), component_distribution(), connect(), coreness(), degree(), dfs(), distance_table(), edge_density(), feedback_arc_set(), feedback_vertex_set(), girth(), is_acyclic(), is_dag(), is_matching(), k_shortest_paths(), knn(), reciprocity(), subcomponent(), subgraph(), topo_sort(), transitivity(), unfold_tree(), which_multiple(), which_mutual()

Examples

```
g <- sample_gnp(20, 5 / 20)
constraint(g)
```

---

contract *Contract several vertices into a single one*

Description

This function creates a new graph, by merging several vertices into one. The vertices in the new graph correspond to sets of vertices in the input graph.

Usage

```
contract(graph, mapping, vertex.attr.comb = igraph_opt("vertex.attr.comb"))
```

Arguments

graph	The input graph, it can be directed or undirected.
mapping	A numeric vector that specifies the mapping. Its elements correspond to the vertices, and for each element the id in the new graph is given.
vertex.attr.comb	Specifies how to combine the vertex attributes in the new graph. Please see attribute.combination() for details.

Details

The attributes of the graph are kept. Graph and edge attributes are unchanged, vertex attributes are combined, according to the vertex.attr.comb parameter.

Value

A new graph object.

Related documentation in the C library

contract_vertices()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Other functions for manipulating graph structure: +.igraph(), add_edges(), add_vertices(), complementer(), compose(), connect(), delete_edges(), delete_vertices(), difference(), difference.igraph(), disjoint_union(), edge(), igraph-minus, intersection(), intersection.igraph(), path(), permute(), rep.igraph(), reverse_edges(), simplify(), transitive_closure(), union(), union.igraph(), vertex()

Examples

```
g <- make_ring(10)
g$name <- "Ring"
V(g)$name <- letters[1:vcount(g)]
E(g)$weight <- runif(ecount(g))

g2 <- contract(g, rep(1:5, each = 2),
  vertex.attr.comb = toString
)

## graph and edge attributes are kept, vertex attributes are
## combined using the 'toString' function.
print(g2, g = TRUE, v = TRUE, e = TRUE)
```

convex_hull *Convex hull of a set of vertices*

Description

Calculate the convex hull of a set of points, i.e. the covering polygon that has the smallest area.

Usage

```
convex_hull(data)
```

Arguments

data	The data points, a numeric matrix with two columns.

Value

A named list with components:

resverts The indices of the input vertices that constitute the convex hull.

rescoords The coordinates of the corners of the convex hull.

Related documentation in the C library

convex_hull_2d()

Author(s)

Tamas Nepusz <ntamas@gmail.com>

References

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0262032937. Pages 949-955 of section 33.3: Finding the convex hull.

See Also

Other other: running_mean(), sample_seq()

Examples

```
M <- cbind(runif(100), runif(100))
convex_hull(M)
```

`coreness` *K-core decomposition of graphs*

Description

The k-core of graph is a maximal subgraph in which each vertex has at least degree k. The coreness of a vertex is k if it belongs to the k-core but not to the (k+1)-core.

Usage

```
coreness(graph, mode = c("all", "out", "in"))
```

Arguments

`graph`	The input graph, it can be directed or undirected
`mode`	The type of the core in directed graphs. Character constant, possible values: `in`: in-cores are computed, `out`: out-cores are computed, `all`: the corresponding undirected graph is considered. This argument is ignored for undirected graphs.

Details

The k-core of a graph is the maximal subgraph in which every vertex has at least degree k. The cores of a graph form layers: the (k+1)-core is always a subgraph of the k-core.

This function calculates the coreness for each vertex.

Value

Numeric vector of integer numbers giving the coreness of each vertex.

Related documentation in the C library

`coreness()`, `vcount()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Vladimir Batagelj, Matjaz Zaversnik: An O(m) Algorithm for Cores Decomposition of Networks, 2002

Seidman S. B. (1983) Network structure and minimum degree, *Social Networks*, 5, 269–287.

See Also

degree()

Other structural.properties: bfs(), component_distribution(), connect(), constraint(), degree(), dfs(), distance_table(), edge_density(), feedback_arc_set(), feedback_vertex_set(), girth(), is_acyclic(), is_dag(), is_matching(), k_shortest_paths(), knn(), reciprocity(), subcomponent(), subgraph(), topo_sort(), transitivity(), unfold_tree(), which_multiple(), which_mutual()

Examples

```
g <- make_ring(10)
g <- add_edges(g, c(1, 2, 2, 3, 1, 3))
coreness(g) # small core triangle in a ring
```

---

count_automorphisms *Number of automorphisms*

Description

Calculate the number of automorphisms of a graph, i.e. the number of isomorphisms to itself.

Usage

```
count_automorphisms(
  graph,
  colors = NULL,
  sh = c("fm", "f", "fs", "fl", "flm", "fsm")
)
```

Arguments

graph	The input graph, it is treated as undirected.
colors	The colors of the individual vertices of the graph; only vertices having the same color are allowed to match each other in an automorphism. When omitted, igraph uses the color attribute of the vertices, or, if there is no such vertex attribute, it simply assumes that all vertices have the same color. Pass NULL explicitly if the graph has a color vertex attribute but you do not want to use it.
sh	The splitting heuristics for the BLISS algorithm. Possible values are: 'f': first non-singleton cell, 'fl': first largest non-singleton cell, 'fs': first smallest non-singleton cell, 'fm': first maximally non-trivially connected non-singleton cell, 'flm': first largest maximally non-trivially connected non-singleton cell, 'fsm': first smallest maximally non-trivially connected non-singleton cell.

Details

An automorphism of a graph is a permutation of its vertices which brings the graph into itself.

This function calculates the number of automorphism of a graph using the BLISS algorithm. See also the BLISS homepage at http://www.tcs.hut.fi/Software/bliss/index.html. If you need the automorphisms themselves, use automorphism_group() to obtain a compact representation of the automorphism group.

Value

A named list with the following members:

group_size The size of the automorphism group of the input graph, as a string. This number is exact if igraph was compiled with the GMP library, and approximate otherwise.

nof_nodes The number of nodes in the search tree.

nof_leaf_nodes The number of leaf nodes in the search tree.

nof_bad_nodes Number of bad nodes.

nof_canupdates Number of canrep updates.

max_level Maximum level.

Related documentation in the C library

count_automorphisms(), vcount()

Author(s)

Tommi Junttila (https://users.ics.aalto.fi/tjunttil/) for BLISS and Gabor Csardi <csardi.gabor@gmail.com> for the igraph glue code and this manual page.

References

Tommi Junttila and Petteri Kaski: Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs, *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics.* 2007.

See Also

canonical_permutation(), permute(), and automorphism_group() for a compact representation of all automorphisms

Other graph automorphism: automorphism_group()

Examples

```
## A ring has n*2 automorphisms, you can "turn" it by 0-9 vertices
## and each of these graphs can be "flipped"
g <- make_ring(10)
count_automorphisms(g)

## A full graph has n! automorphisms; however, we restrict the vertex
```

```
## matching by colors, leading to only 4 automorphisms
g <- make_full_graph(4)
count_automorphisms(g, colors = c(1, 2, 1, 2))
```

`count_isomorphisms` *Count the number of isomorphic mappings between two graphs*

Description

Count the number of isomorphic mappings between two graphs

Usage

```
count_isomorphisms(graph1, graph2, method = "vf2", ...)
```

Arguments

graph1	The first graph.
graph2	The second graph.
method	Currently only 'vf2' is supported, see isomorphic() for details about it and extra arguments.
...	Passed to the individual methods.

Value

Number of isomorphic mappings between the two graphs.

Related documentation in the C library

count_isomorphisms_vf2(), vcount(), edges(), get_eids(), ecount()

References

LP Cordella, P Foggia, C Sansone, and M Vento: An improved algorithm for matching large graphs, *Proc. of the 3rd IAPR TC-15 Workshop on Graphbased Representations in Pattern Recognition*, 149–159, 2001.

See Also

Other graph isomorphism: canonical_permutation(), count_subgraph_isomorphisms(), graph_from_isomorphism_c
isomorphic(), isomorphism_class(), isomorphisms(), subgraph_isomorphic(), subgraph_isomorphisms()

Examples

```
# colored graph isomorphism
g1 <- make_ring(10)
g2 <- make_ring(10)
isomorphic(g1, g2)

V(g1)$color <- rep(1:2, length = vcount(g1))
V(g2)$color <- rep(2:1, length = vcount(g2))
# consider colors by default
count_isomorphisms(g1, g2)
# ignore colors
count_isomorphisms(g1, g2,
  vertex.color1 = NULL,
  vertex.color2 = NULL
)
```

---

count_motifs *Graph motifs*

Description

Graph motifs are small connected induced subgraphs with a well-defined structure. These functions search a graph for various motifs.

Usage

```
count_motifs(graph, size = 3, cut.prob = NULL)
```

Arguments

graph	Graph object, the input graph.
size	The size of the motif.
cut.prob	Numeric vector giving the probabilities that the search graph is cut at a certain level. Its length should be the same as the size of the motif (the size argument). If NULL, the default, no cuts are made.

Details

count_motifs() calculates the total number of motifs of a given size in graph.

Value

count_motifs() returns a numeric scalar.

Related documentation in the C library

motifs_randesu_no()

See Also

isomorphism_class()

Other graph motifs: dyad_census(), motifs(), sample_motifs()

Examples

```
g <- sample_pa(100)
motifs(g, 3)
count_motifs(g, 3)
sample_motifs(g, 3)
```

---

count_reachable *Count reachable vertices*

Description

[Experimental]

Usage

```
count_reachable(graph, mode = c("out", "in", "all", "total"))
```

Arguments

graph	The input graph.
mode	Character constant, defines how edge directions are considered in directed graphs. "out" counts vertices reachable via outgoing edges, "in" counts vertices from which the current vertex is reachable via incoming edges, "all" or "total" ignores edge directions. This parameter is ignored for undirected graphs.

Details

Counts the number of vertices reachable from each vertex in the graph.

For each vertex in the graph, this function counts how many vertices are reachable from it, including the vertex itself. A vertex is reachable from another if there is a directed path between them. For undirected graphs, two vertices are reachable from each other if they are in the same connected component.

Value

An integer vector of length vcount(graph). The i-th element is the number of vertices reachable from vertex i (including vertex i itself).

Related documentation in the C library

count_reachable()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

components(), subcomponent(), is_connected()

Connected components: articulation_points(), biconnected_components(), component_distribution(), decompose(), is_biconnected()

Examples

```
# In a directed path graph, the reachability depends on direction
g <- make_graph(~ 1 -+ 2 -+ 3 -+ 4 -+ 5)
count_reachable(g, mode = "out")
count_reachable(g, mode = "in")

# In an undirected graph, reachability is the same in all directions
g2 <- make_graph(~ 1 - 2 - 3 - 4 - 5)
count_reachable(g2, mode = "out")

# A graph with multiple components
g3 <- make_graph(~ 1 - 2 - 3, 4 - 5, 6)
count_reachable(g3, mode = "all")
```

---

count_subgraph_isomorphisms

Count the isomorphic mappings between a graph and the subgraphs of another graph

---

Description

Count the isomorphic mappings between a graph and the subgraphs of another graph

Usage

```
count_subgraph_isomorphisms(pattern, target, method = c("lad", "vf2"), ...)
```

Arguments

pattern	The smaller graph, it might be directed or undirected. Undirected graphs are treated as directed graphs with mutual edges.
target	The bigger graph, it might be directed or undirected. Undirected graphs are treated as directed graphs with mutual edges.
method	The method to use. Possible values: 'lad', 'vf2'. See their details below.
...	Additional arguments, passed to the various methods.

Value

Logical scalar, TRUE if the pattern is isomorphic to a (possibly induced) subgraph of target.

'lad' method

This is the LAD algorithm by Solnon, see the reference below. It has the following extra arguments:

domains If not NULL, then it specifies matching restrictions. It must be a list of target vertex sets, given as numeric vertex ids or symbolic vertex names. The length of the list must be vcount(pattern) and for each vertex in pattern it gives the allowed matching vertices in target. Defaults to NULL.

induced Logical scalar, whether to search for an induced subgraph. It is FALSE by default.

time.limit The processor time limit for the computation, in seconds. It defaults to Inf, which means no limit.

'vf2' method

This method uses the VF2 algorithm by Cordella, Foggia et al., see references below. It supports vertex and edge colors and have the following extra arguments:

vertex.color1, vertex.color2 Optional integer vectors giving the colors of the vertices for colored graph isomorphism. If they are not given, but the graph has a "color" vertex attribute, then it will be used. If you want to ignore these attributes, then supply NULL for both of these arguments. See also examples below.

edge.color1, edge.color2 Optional integer vectors giving the colors of the edges for edge-colored (sub)graph isomorphism. If they are not given, but the graph has a "color" edge attribute, then it will be used. If you want to ignore these attributes, then supply NULL for both of these arguments.

Related documentation in the C library

count_subisomorphisms_vf2(), vcount(), edges(), get_eids(), ecount()

References

LP Cordella, P Foggia, C Sansone, and M Vento: An improved algorithm for matching large graphs, *Proc. of the 3rd IAPR TC-15 Workshop on Graphbased Representations in Pattern Recognition*, 149–159, 2001.

C. Solnon: AllDifferent-based Filtering for Subgraph Isomorphism, *Artificial Intelligence* 174(12-13):850–864, 2010.

See Also

Other graph isomorphism: canonical_permutation(), count_isomorphisms(), graph_from_isomorphism_class(), isomorphic(), isomorphism_class(), isomorphisms(), subgraph_isomorphic(), subgraph_isomorphisms()

curve_multiple *Optimal edge curvature when plotting graphs*

Description

If graphs have multiple edges, then drawing them as straight lines does not show them when plotting the graphs; they will be on top of each other. One solution is to bend the edges, with diffenent curvature, so that all of them are visible.

Usage

```
curve_multiple(graph, start = 0.5)
```

Arguments

graph	The input graph.
start	The curvature at the two extreme edges. All edges will have a curvature between -start and start, spaced equally.

Details

curve_multiple() calculates the optimal edge.curved vector for plotting a graph with multiple edges, so that all edges are visible.

Value

A numeric vector, its length is the number of edges in the graph.

Related documentation in the C library

get_edgelist(), vcount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

igraph.plotting for all plotting parameters, plot.igraph(), tkplot() and rglplot() for plotting functions.

Examples

```
g <- make_graph(c(
  0, 1, 1, 0, 1, 2, 1, 3, 1, 3, 1, 3,
  2, 3, 2, 3, 2, 3, 2, 3, 0, 1
) + 1)

curve_multiple(g)

set.seed(42)
plot(g)
```

---

decompose *Decompose a graph into components*

Description

Creates a separate graph for each connected component of a graph.

Usage

```
decompose(graph, mode = c("weak", "strong"), max.comps = NA, min.vertices = 0)
```

Arguments

graph	The original graph.
mode	Character constant giving the type of the components, wither weak for weakly connected components or strong for strongly connected components.
max.comps	The maximum number of components to return. The first max.comps components will be returned (which hold at least min.vertices vertices, see the next parameter), the others will be ignored. Supply NA here if you don't want to limit the number of components.
min.vertices	The minimum number of vertices a component should contain in order to place it in the result list. E.g. supply 2 here to ignore isolate vertices.

Value

A list of graph objects.

Related documentation in the C library

decompose()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

is_connected() to decide whether a graph is connected, components() to calculate the connected components of a graph.

Connected components: articulation_points(), biconnected_components(), component_distribution(), count_reachable(), is_biconnected()

Examples

```
# the diameter of each component in a random graph
g <- sample_gnp(1000, 1 / 1000)
components <- decompose(g, min.vertices = 2)
sapply(components, diameter)
```

---

degree *Degree and degree distribution of the vertices*

Description

The degree of a vertex is its most basic structural property, the number of its adjacent edges.

Usage

```
degree(
  graph,
  v = V(graph),
  mode = c("all", "out", "in", "total"),
  loops = TRUE,
  normalized = FALSE
)

max_degree(
  graph,
  ...,
  v = V(graph),
  mode = c("all", "out", "in", "total"),
  loops = TRUE
)

mean_degree(graph, loops = TRUE)

degree_distribution(graph, cumulative = FALSE, ...)
```

Arguments

graph	The graph to analyze.
v	The ids of vertices of which the degree will be calculated.
mode	Character string, "out" for out-degree, "in" for in-degree or "total" for the sum of the two. For undirected graphs this argument is ignored. "all" is a synonym of "total".
loops	Logical; whether the loop edges are also counted.
normalized	Logical scalar, whether to normalize the degree. If TRUE then the result is divided by $n - 1$, where n is the number of vertices in the graph.
...	These dots are for future extensions and must be empty.
cumulative	Logical; whether the cumulative degree distribution is to be calculated.

Value

For degree() a numeric vector of the same length as argument v.

For degree_distribution() a numeric vector of the same length as the maximum degree plus one. The first element is the relative frequency zero degree vertices, the second vertices with degree one, etc.

For max_degree(), the largest degree in the graph. When no vertices are selected, or when the input is the null graph, zero is returned as this is the smallest possible degree.

For mean_degree(), the average degree in the graph as a single number. For graphs with no vertices, NaN is returned. **[Experimental]**

Related documentation in the C library

degree(), vcount(), maxdegree(), mean_degree()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Other structural.properties: bfs(), component_distribution(), connect(), constraint(), coreness(), dfs(), distance_table(), edge_density(), feedback_arc_set(), feedback_vertex_set(), girth(), is_acyclic(), is_dag(), is_matching(), k_shortest_paths(), knn(), reciprocity(), subcomponent(), subgraph(), topo_sort(), transitivity(), unfold_tree(), which_multiple(), which_mutual()

Examples

```
g <- make_ring(10)
degree(g)
g2 <- sample_gnp(1000, 10 / 1000)
max_degree(g2)
mean_degree(g2)
degree_distribution(g2)
```

delete_edge_attr *Delete an edge attribute*

Description

Delete an edge attribute

Usage

```
delete_edge_attr(graph, name)
```

Arguments

graph	The graph
name	The name of the edge attribute to delete.

Value

The graph, with the specified edge attribute removed.

See Also

Vertex, edge and graph attributes: delete_graph_attr(), delete_vertex_attr(), edge_attr(), edge_attr<-(), edge_attr_names(), graph_attr(), graph_attr<-(), graph_attr_names(), igraph-attribute-combination, igraph-dollar, igraph-vs-attributes, set_edge_attr(), set_graph_attr(), set_vertex_attr(), set_vertex_attrs(), vertex_attr(), vertex_attr<-(), vertex_attr_names()

Examples

```
g <- make_ring(10) %>%
  set_edge_attr("name", value = LETTERS[1:10])
edge_attr_names(g)
g2 <- delete_edge_attr(g, "name")
edge_attr_names(g2)
```

delete_edges *Delete edges from a graph*

Description

Delete edges from a graph

Usage

```
delete_edges(graph, edges)
```

Arguments

graph	The input graph.
edges	The edges to remove, specified as an edge sequence. Typically this is either a numeric vector containing edge IDs, or a character vector containing the IDs or names of the source and target vertices, separated by |

Value

The graph, with the edges removed.

Related documentation in the C library

delete_edges(), get_eids(), edges(), ecount(), vcount()

See Also

Other functions for manipulating graph structure: +.igraph(), add_edges(), add_vertices(), complementer(), compose(), connect(), contract(), delete_vertices(), difference(), difference.igraph(), disjoint_union(), edge(), igraph-minus, intersection(), intersection.igraph(), path(), permute(), rep.igraph(), reverse_edges(), simplify(), transitive_closure(), union(), union.igraph(), vertex()

Examples

```
g <- make_ring(10) %>%
  delete_edges(seq(1, 9, by = 2))
g

g <- make_ring(10) %>%
  delete_edges("10|1")
g

g <- make_ring(5)
g <- delete_edges(g, get_edge_ids(g, c(1, 5, 4, 5)))
g
```

---

delete_graph_attr *Delete a graph attribute*

Description

Delete a graph attribute

Usage

```
delete_graph_attr(graph, name)
```

Arguments

graph	The graph.
name	Name of the attribute to delete.

Value

The graph, with the specified attribute removed.

See Also

Vertex, edge and graph attributes: delete_edge_attr(), delete_vertex_attr(), edge_attr(), edge_attr<-(), edge_attr_names(), graph_attr(), graph_attr<-(), graph_attr_names(), igraph-attribute-combination, igraph-dollar, igraph-vs-attributes, set_edge_attr(), set_graph_attr(), set_vertex_attr(), set_vertex_attrs(), vertex_attr(), vertex_attr<-(), vertex_attr_names()

Examples

```
g <- make_ring(10)
graph_attr_names(g)
g2 <- delete_graph_attr(g, "name")
graph_attr_names(g2)
```

---

delete_vertex_attr *Delete a vertex attribute*

Description

Delete a vertex attribute

Usage

```
delete_vertex_attr(graph, name)
```

Arguments

graph	The graph
name	The name of the vertex attribute to delete.

Value

The graph, with the specified vertex attribute removed.

See Also

Vertex, edge and graph attributes: delete_edge_attr(), delete_graph_attr(), edge_attr(), edge_attr<-(), edge_attr_names(), graph_attr(), graph_attr<-(), graph_attr_names(), igraph-attribute-combination, igraph-dollar, igraph-vs-attributes, set_edge_attr(), set_graph_attr(), set_vertex_attr(), set_vertex_attrs(), vertex_attr(), vertex_attr<-(), vertex_attr_names()

Examples

```
g <- make_ring(10) %>%
  set_vertex_attr("name", value = LETTERS[1:10])
vertex_attr_names(g)
g2 <- delete_vertex_attr(g, "name")
vertex_attr_names(g2)
```

---

delete_vertices *Delete vertices from a graph*

Description

Delete vertices from a graph

Usage

```
delete_vertices(graph, v)
```

Arguments

graph	The input graph.
v	The vertices to remove, a vertex sequence.

Value

The graph, with the vertices removed.

Related documentation in the C library

delete_vertices(), vcount()

See Also

Other functions for manipulating graph structure: +.igraph(), add_edges(), add_vertices(), complementer(), compose(), connect(), contract(), delete_edges(), difference(), difference.igraph(), disjoint_union(), edge(), igraph-minus, intersection(), intersection.igraph(), path(), permute(), rep.igraph(), reverse_edges(), simplify(), transitive_closure(), union(), union.igraph(), vertex()

Examples

```
g <- make_ring(10) %>%
  set_vertex_attr("name", value = LETTERS[1:10])
g
V(g)

g2 <- delete_vertices(g, c(1, 5)) %>%
  delete_vertices("B")
g2
V(g2)
```

---

dfs *Depth-first search*

Description

Depth-first search is an algorithm to traverse a graph. It starts from a root vertex and tries to go quickly as far from as possible.

Usage

```
dfs(
  graph,
  root,
  mode = c("out", "in", "all", "total"),
  ...,
  unreachable = TRUE,
  order = TRUE,
  order.out = FALSE,
  parent = FALSE,
  dist = FALSE,
  in.callback = NULL,
  out.callback = NULL,
  extra = NULL,
  rho = parent.frame(),
  neimode = deprecated(),
  father = deprecated()
)
```

Arguments

graph	The input graph.
root	The single root vertex to start the search from.
mode	For directed graphs specifies the type of edges to follow. 'out' follows outgoing, 'in' incoming edges. 'all' ignores edge directions completely. 'total' is a synonym for 'all'. This argument is ignored for undirected graphs.
...	These dots are for future extensions and must be empty.

unreachable	Logical scalar, whether the search should visit the vertices that are unreachable from the given root vertex (or vertices). If TRUE, then additional searches are performed until all vertices are visited.
order	Logical scalar, whether to return the DFS ordering of the vertices.
order.out	Logical scalar, whether to return the ordering based on leaving the subtree of the vertex.
parent	Logical scalar, whether to return the parent of the vertices.
dist	Logical scalar, whether to return the distance from the root of the search tree.
in.callback	If not NULL, then it must be callback function. This is called whenever a vertex is visited. See details below.
out.callback	If not NULL, then it must be callback function. This is called whenever the subtree of a vertex is completed by the algorithm. See details below.
extra	Additional argument to supply to the callback function.
rho	The environment in which the callback function is evaluated.
neimode	**[Deprecated]** This argument is deprecated from igraph 1.3.0; use mode instead.
father	**[Deprecated]**, use parent instead.

Details

The callback functions must have the following arguments:

graph The input graph is passed to the callback function here.

data A named numeric vector, with the following entries: 'vid', the vertex that was just visited and 'dist', its distance from the root of the search tree.

extra The extra argument.

The callback must return FALSE to continue the search or TRUE to terminate it. See examples below on how to use the callback functions.

Value

A named list with the following entries:

root Numeric scalar. The root vertex that was used as the starting point of the search.

neimode Character scalar. The mode argument of the function call. Note that for undirected graphs this is always 'all', irrespectively of the supplied value.

order Numeric vector. The vertex ids, in the order in which they were visited by the search.

order.out Numeric vector, the vertex ids, in the order of the completion of their subtree.

parent Numeric vector. The parent of each vertex, i.e. the vertex it was discovered from.

father Like parent, kept for compatibility for now.

dist Numeric vector, for each vertex its distance from the root of the search tree.

Note that order, order.out, parent, and dist might be NULL if their corresponding argument is FALSE, i.e. if their calculation is not requested.

Related documentation in the C library

vcount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

bfs() for breadth-first search.

Other structural.properties: bfs(), component_distribution(), connect(), constraint(), coreness(), degree(), distance_table(), edge_density(), feedback_arc_set(), feedback_vertex_set(), girth(), is_acyclic(), is_dag(), is_matching(), k_shortest_paths(), knn(), reciprocity(), subcomponent(), subgraph(), topo_sort(), transitivity(), unfold_tree(), which_multiple(), which_mutual()

Examples

```
## A graph with two separate trees
dfs(
  graph = make_tree(10) %du% make_tree(10),
  root = 1, mode = "out",
  unreachable = TRUE,
  order = TRUE,
  order.out = TRUE,
  parent = TRUE
)

## How to use a callback
f.in <- function(graph, data, extra) {
  cat("in:", paste(collapse = ", ", data), "\n")
  FALSE
}
f.out <- function(graph, data, extra) {
  cat("out:", paste(collapse = ", ", data), "\n")
  FALSE
}
tmp <- dfs(
  graph = make_tree(10),
  root = 1, mode = "out",
  in.callback = f.in, out.callback = f.out
)

## Terminate after the first component, using a callback
f.out <- function(graph, data, extra) {
  data["vid"] == 1
}
tmp <- dfs(
  graph = make_tree(10) %du% make_tree(10),
  root = 1,
  out.callback = f.out
```

```
)
```

`diameter` *Diameter of a graph*

Description

The diameter of a graph is the length of the longest geodesic.

Usage

```
diameter(graph, directed = TRUE, unconnected = TRUE, weights = NULL)

get_diameter(graph, directed = TRUE, unconnected = TRUE, weights = NULL)

farthest_vertices(graph, directed = TRUE, unconnected = TRUE, weights = NULL)
```

Arguments

`graph`	The graph to analyze.
`directed`	Logical, whether directed or undirected paths are to be considered. This is ignored for undirected graphs.
`unconnected`	Logical, what to do if the graph is unconnected. If FALSE, the function will return a number that is one larger the largest possible diameter, which is always the number of vertices. If TRUE, the diameters of the connected components will be calculated and the largest one will be returned.
`weights`	Optional positive weight vector for calculating weighted distances. If the graph has a `weight` edge attribute, then this is used by default.

Details

The diameter is calculated by using a breadth-first search like method.

`get_diameter()` returns a path with the actual diameter. If there are many shortest paths of the length of the diameter, then it returns the first one found.

`farthest_vertices()` returns two vertex ids, the vertices which are connected by the diameter path.

Value

A numeric constant for `diameter()`, a numeric vector for `get_diameter()`. `farthest_vertices()` returns a list with two entries:

`vertices` The two vertices that are the farthest.

`distance` Their distance.

Related documentation in the C library

edges(), get_eids(), vcount(), ecount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

distances()

Other paths: all_simple_paths(), distance_table(), eccentricity(), graph_center(), radius()

Examples

```
g <- make_ring(10)
g2 <- delete_edges(g, c(1, 2, 1, 10))
diameter(g2, unconnected = TRUE)
diameter(g2, unconnected = FALSE)

## Weighted diameter
set.seed(1)
g <- make_ring(10)
E(g)$weight <- sample(seq_len(ecount(g)))
diameter(g)
get_diameter(g)
diameter(g, weights = NA)
get_diameter(g, weights = NA)
```

---

difference *Difference of two sets*

Description

This is an S3 generic function. See methods("difference") for the actual implementations for various S3 classes. Initially it is implemented for igraph graphs (difference of edges in two graphs), and igraph vertex and edge sequences. See difference.igraph(), and difference.igraph.vs().

Usage

```
difference(...)
```

Arguments

...	Arguments, their number and interpretation depends on the function that implements difference().

Value

Depends on the function that implements this method.

See Also

Other functions for manipulating graph structure: `+.igraph()`, `add_edges()`, `add_vertices()`, `complementer()`, `compose()`, `connect()`, `contract()`, `delete_edges()`, `delete_vertices()`, `difference.igraph()`, `disjoint_union()`, `edge()`, `igraph-minus`, `intersection()`, `intersection.igraph()`, `path()`, `permute()`, `rep.igraph()`, `reverse_edges()`, `simplify()`, `transitive_closure()`, `union()`, `union.igraph()`, `vertex()`

---

difference.igraph *Difference of graphs*

Description

The difference of two graphs are created.

Usage

```
## S3 method for class 'igraph'
difference(big, small, byname = "auto", ...)
```

Arguments

big	The left hand side argument of the minus operator. A directed or undirected graph.
small	The right hand side argument of the minus operator. A directed ot undirected graph.
byname	A logical scalar, or the character scalar `auto`. Whether to perform the operation based on symbolic vertex names. If it is `auto`, that means `TRUE` if both graphs are named and `FALSE` otherwise. A warning is generated if `auto` and one graph, but not both graphs are named.
...	Ignored, included for S3 compatibility.

Details

`difference()` creates the difference of two graphs. Only edges present in the first graph but not in the second will be be included in the new graph. The corresponding operator is `%m%`.

If the `byname` argument is `TRUE` (or `auto` and the graphs are all named), then the operation is performed based on symbolic vertex names. Otherwise numeric vertex ids are used.

`difference()` keeps all attributes (graph, vertex and edge) of the first graph.

Note that `big` and `small` must both be directed or both be undirected, otherwise an error message is given.

Value

A new graph object.

Related documentation in the C library

difference(), vcount(), permute_vertices()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Other functions for manipulating graph structure: +.igraph(), add_edges(), add_vertices(), complementer(), compose(), connect(), contract(), delete_edges(), delete_vertices(), difference(), disjoint_union(), edge(), igraph-minus, intersection(), intersection.igraph(), path(), permute(), rep.igraph(), reverse_edges(), simplify(), transitive_closure(), union(), union.igraph(), vertex()

Examples

```
## Create a wheel graph
wheel <- union(
  make_ring(10),
  make_star(11, center = 11, mode = "undirected")
)
V(wheel)$name <- letters[seq_len(vcount(wheel))]

## Subtract a star graph from it
sstar <- make_star(6, center = 6, mode = "undirected")
V(sstar)$name <- letters[c(1, 3, 5, 7, 9, 11)]
G <- wheel %m% sstar
print_all(G)
plot(G, layout = layout_nicely(wheel))
```

---

difference.igraph.es *Difference of edge sequences*

Description

Difference of edge sequences

Usage

```
## S3 method for class 'igraph.es'
difference(big, small, ...)
```

Arguments

big	The 'big' edge sequence.
small	The 'small' edge sequence.
...	Ignored, included for S3 signature compatibility.

Details

They must belong to the same graph. Note that this function has 'set' semantics and the multiplicity of edges is lost in the result.

Value

An edge sequence that contains only edges that are part of big, but not part of small.

See Also

Other vertex and edge sequence operations: c.igraph.es(), c.igraph.vs(), difference.igraph.vs(), igraph-es-indexing, igraph-es-indexing2, igraph-vs-indexing, igraph-vs-indexing2, intersection.igraph.es intersection.igraph.vs(), rev.igraph.es(), rev.igraph.vs(), union.igraph.es(), union.igraph.vs(), unique.igraph.es(), unique.igraph.vs()

Examples

```
g <- make_(ring(10), with_vertex_(name = LETTERS[1:10]))
difference(V(g), V(g)[6:10])
```

---

difference.igraph.vs *Difference of vertex sequences*

Description

Difference of vertex sequences

Usage

```
## S3 method for class 'igraph.vs'
difference(big, small, ...)
```

Arguments

big	The 'big' vertex sequence.
small	The 'small' vertex sequence.
...	Ignored, included for S3 signature compatibility.

Details

They must belong to the same graph. Note that this function has 'set' semantics and the multiplicity of vertices is lost in the result.

Value

A vertex sequence that contains only vertices that are part of `big`, but not part of `small`.

See Also

Other vertex and edge sequence operations: `c.igraph.es()`, `c.igraph.vs()`, `difference.igraph.es()`, `igraph-es-indexing`, `igraph-es-indexing2`, `igraph-vs-indexing`, `igraph-vs-indexing2`, `intersection.igraph.es`, `intersection.igraph.vs()`, `rev.igraph.es()`, `rev.igraph.vs()`, `union.igraph.es()`, `union.igraph.vs()`, `unique.igraph.es()`, `unique.igraph.vs()`

Examples

```
g <- make_(ring(10), with_vertex_(name = LETTERS[1:10]))
difference(V(g), V(g)[6:10])
```

---

dim_select *Dimensionality selection for singular values using profile likelihood.*

Description

Select the number of significant singular values, by finding the 'elbow' of the scree plot, in a principled way.

Usage

```
dim_select(sv)
```

Arguments

`sv`	A numeric vector, the ordered singular values.

Details

The input of the function is a numeric vector which contains the measure of 'importance' for each dimension.

For spectral embedding, these are the singular values of the adjacency matrix. The singular values are assumed to be generated from a Gaussian mixture distribution with two components that have different means and same variance. The dimensionality d is chosen to maximize the likelihood when the d largest singular values are assigned to one component of the mixture and the rest of the singular values assigned to the other component.

This function can also be used for the general separation problem, where we assume that the left and the right of the vector are coming from two Normal distributions, with different means, and we want to know their border. See examples below.

Value

A numeric scalar, the estimate of d.

Related documentation in the C library

dim_select()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

M. Zhu, and A. Ghodsi (2006). Automatic dimensionality selection from the scree plot via the use of profile likelihood. *Computational Statistics and Data Analysis*, Vol. 51, 918–930.

See Also

embed_adjacency_matrix()

Other embedding: embed_adjacency_matrix(), embed_laplacian_matrix()

Examples

```
# Generate the two groups of singular values with
# Gaussian mixture of two components that have different means
sing.vals <- c(rnorm(10, mean = 1, sd = 1), rnorm(10, mean = 3, sd = 1))
dim.chosen <- dim_select(sing.vals)
dim.chosen

# Sample random vectors with multivariate normal distribution
# and normalize to unit length
lpvs <- matrix(rnorm(200), 10, 20)
lpvs <- apply(lpvs, 2, function(x) {
  (abs(x) / sqrt(sum(x^2)))
})
RDP.graph <- sample_dot_product(lpvs)
dim_select(embed_adjacency_matrix(RDP.graph, 10)$D)

# Sample random vectors with the Dirichlet distribution
lpvs.dir <- sample_dirichlet(n = 20, rep(1, 10))
RDP.graph.2 <- sample_dot_product(lpvs.dir)
dim_select(embed_adjacency_matrix(RDP.graph.2, 10)$D)

# Sample random vectors from hypersphere with radius 1.
lpvs.sph <- sample_sphere_surface(dim = 10, n = 20, radius = 1)
RDP.graph.3 <- sample_dot_product(lpvs.sph)
dim_select(embed_adjacency_matrix(RDP.graph.3, 10)$D)
```

`disjoint_union` *Disjoint union of graphs*

Description

The union of two or more graphs are created. The graphs are assumed to have disjoint vertex sets.

Usage

```
disjoint_union(...)

x %du% y
```

Arguments

`...`	Graph objects or lists of graph objects.
`x, y`	Graph objects.

Details

`disjoint_union()` creates a union of two or more disjoint graphs. Thus first the vertices in the second, third, etc. graphs are relabeled to have completely disjoint graphs. Then a simple union is created. This function can also be used via the `%du%` operator.

`disjoint_union()` handles graph, vertex and edge attributes. In particular, it merges vertex and edge attributes using the `vctrs::vec_c()` function. For graphs that lack some vertex/edge attribute, the corresponding values in the new graph are set to a missing value (`NA` for scalar attributes, `NULL` for list attributes). Graph attributes are simply copied to the result. If this would result a name clash, then they are renamed by adding suffixes: _1, _2, etc.

Note that if both graphs have vertex names (i.e. a `name` vertex attribute), then the concatenated vertex names might be non-unique in the result. A warning is given if this happens.

An error is generated if some input graphs are directed and others are undirected.

Value

A new graph object.

Related documentation in the C library

`vcount()`, `edges()`, `get_eids()`, `ecount()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Other functions for manipulating graph structure: +.igraph(), add_edges(), add_vertices(), complementer(), compose(), connect(), contract(), delete_edges(), delete_vertices(), difference(), difference.igraph(), edge(), igraph-minus, intersection(), intersection.igraph(), path(), permute(), rep.igraph(), reverse_edges(), simplify(), transitive_closure(), union(), union.igraph(), vertex()

Examples

```
## A star and a ring
g1 <- make_star(10, mode = "undirected")
V(g1)$name <- letters[1:10]
g2 <- make_ring(10)
V(g2)$name <- letters[11:20]
print_all(g1 %du% g2)
```

---

distance_table *Shortest (directed or undirected) paths between vertices*

Description

distances() calculates the length of all the shortest paths from or to the vertices in the network. shortest_paths() calculates one shortest path (the path itself, and not just its length) from or to the given vertex.

Usage

```
distance_table(graph, directed = TRUE)

mean_distance(
  graph,
  weights = NULL,
  directed = TRUE,
  unconnected = TRUE,
  details = FALSE
)

distances(
  graph,
  v = V(graph),
  to = V(graph),
  mode = c("all", "out", "in"),
  weights = NULL,
 algorithm = c("automatic", "unweighted", "dijkstra", "bellman-ford", "johnson",
    "floyd-warshall")
)
```

```
shortest_paths(
  graph,
  from,
  to = V(graph),
  mode = c("out", "all", "in"),
  weights = NULL,
  output = c("vpath", "epath", "both"),
  predecessors = FALSE,
  inbound.edges = FALSE,
  algorithm = c("automatic", "unweighted", "dijkstra", "bellman-ford")
)

all_shortest_paths(
  graph,
  from,
  to = V(graph),
  mode = c("out", "all", "in"),
  weights = NULL
)
```

Arguments

graph	The graph to work on.
directed	Whether to consider directed paths in directed graphs, this argument is ignored for undirected graphs.
weights	Possibly a numeric vector giving edge weights. If this is NULL and the graph has a weight edge attribute, then the attribute is used. If this is NA then no weights are used (even if the graph has a weight attribute). In a weighted graph, the length of a path is the sum of the weights of its constituent edges.
unconnected	What to do if the graph is unconnected (not strongly connected if directed paths are considered). If TRUE, only the lengths of the existing paths are considered and averaged; if FALSE, the length of the missing paths are considered as having infinite length, making the mean distance infinite as well.
details	Whether to provide additional details in the result. Functions accepting this argument (like mean_distance()) return additional information like the number of disconnected vertex pairs in the result when this parameter is set to TRUE.
v	Numeric vector, the vertices from which the shortest paths will be calculated.
to	Numeric vector, the vertices to which the shortest paths will be calculated. By default it includes all vertices. Note that for distances() every vertex must be included here at most once. (This is not required for shortest_paths().
mode	Character constant, gives whether the shortest paths to or from the given vertices should be calculated for directed graphs. If out then the shortest paths *from* the vertex, if in then *to* it will be considered. If all, the default, then the graph is treated as undirected, i.e. edge directions are not taken into account. This argument is ignored for undirected graphs.

`algorithm` Which algorithm to use for the calculation. By default igraph tries to select the fastest suitable algorithm. If there are no weights, then an unweighted breadth-first search is used, otherwise if all weights are positive, then Dijkstra's algorithm is used. If there are negative weights and we do the calculation for more than 100 sources, then Johnson's algorithm is used. Otherwise the Bellman-Ford algorithm is used. You can override igraph's choice by explicitly giving this parameter. Note that the igraph C core might still override your choice in obvious cases, i.e. if there are no edge weights, then the unweighted algorithm will be used, regardless of this argument.

`from` Numeric constant, the vertex from or to the shortest paths will be calculated. Note that right now this is not a vector of vertex ids, but only a single vertex.

`output` Character scalar, defines how to report the shortest paths. "vpath" means that the vertices along the paths are reported, this form was used prior to igraph version 0.6. "epath" means that the edges along the paths are reported. "both" means that both forms are returned, in a named list with components "vpath" and "epath".

`predecessors` Logical scalar, whether to return the predecessor vertex for each vertex. The predecessor of vertex `i` in the tree is the vertex from which vertex `i` was reached. The predecessor of the start vertex (in the `from` argument) is itself by definition. If the predecessor is zero, it means that the given vertex was not reached from the source during the search. Note that the search terminates if all the vertices in `to` are reached.

`inbound.edges` Logical scalar, whether to return the inbound edge for each vertex. The inbound edge of vertex `i` in the tree is the edge via which vertex `i` was reached. The start vertex and vertices that were not reached during the search will have zero in the corresponding entry of the vector. Note that the search terminates if all the vertices in `to` are reached.

Details

The shortest path, or geodesic between two pair of vertices is a path with the minimal number of vertices. The functions documented in this manual page all calculate shortest paths between vertex pairs.

`distances()` calculates the lengths of pairwise shortest paths from a set of vertices (`from`) to another set of vertices (`to`). It uses different algorithms, depending on the `algorithm` argument and the `weight` edge attribute of the graph. The implemented algorithms are breadth-first search ('`unweighted`'), this only works for unweighted graphs; the Dijkstra algorithm ('`dijkstra`'), this works for graphs with non-negative edge weights; the Bellman-Ford algorithm ('`bellman-ford`'); Johnson's algorithm ('`johnson`'); and a faster version of the Floyd-Warshall algorithm with expected quadratic running time ('`floyd-warshall`'). The latter three algorithms work with arbitrary edge weights, but (naturally) only for graphs that don't have a negative cycle. Note that a negative-weight edge in an undirected graph implies such a cycle. Johnson's algorithm performs better than the Bellman-Ford one when many source (and target) vertices are given, with all-pairs shortest path length calculations being the typical use case.

igraph can choose automatically between algorithms, and chooses the most efficient one that is appropriate for the supplied weights (if any). For automatic algorithm selection, supply '`automatic`' as the `algorithm` argument. (This is also the default.)

`shortest_paths()` calculates a single shortest path (i.e. the path itself, not just its length) between the source vertex given in `from`, to the target vertices given in `to`. `shortest_paths()` uses breadth-first search for unweighted graphs and Dijkstra's algorithm for weighted graphs. The latter only works if the edge weights are non-negative.

`all_shortest_paths()` calculates *all* shortest paths between pairs of vertices, including several shortest paths of the same length. More precisely, it computerd all shortest path starting at `from`, and ending at any vertex given in `to`. It uses a breadth-first search for unweighted graphs and Dijkstra's algorithm for weighted ones. The latter only supports non-negative edge weights. Caution: in multigraphs, the result size is exponentially large in the number of vertex pairs with multiple edges between them.

`mean_distance()` calculates the average path length in a graph, by calculating the shortest paths between all pairs of vertices (both ways for directed graphs). It uses a breadth-first search for unweighted graphs and Dijkstra's algorithm for weighted ones. The latter only supports non-negative edge weights.

`distance_table()` calculates a histogram, by calculating the shortest path length between each pair of vertices. For directed graphs both directions are considered, so every pair of vertices appears twice in the histogram.

Value

For `distances()` a numeric matrix with `length(to)` columns and `length(v)` rows. The shortest path length from a vertex to itself is always zero. For unreachable vertices `Inf` is included.

For `shortest_paths()` a named list with four entries is returned:

`vpath` This itself is a list, of length `length(to)`; list element `i` contains the vertex ids on the path from vertex `from` to vertex `to[i]` (or the other way for directed graphs depending on the `mode` argument). The vector also contains `from` and `i` as the first and last elements. If `from` is the same as `i` then it is only included once. If there is no path between two vertices then a numeric vector of length zero is returned as the list element. If this output is not requested in the `output` argument, then it will be `NULL`.

`epath` This is a list similar to `vpath`, but the vectors of the list contain the edge ids along the shortest paths, instead of the vertex ids. This entry is set to `NULL` if it is not requested in the `output` argument.

`predecessors` Numeric vector, the predecessor of each vertex in the `to` argument, or `NULL` if it was not requested.

`inbound_edges` Numeric vector, the inbound edge for each vertex, or `NULL`, if it was not requested.

For `all_shortest_paths()` a list is returned:

vpaths This is a list. Each list element contains the vertices of a shortest path from `from` to a vertex in `to`. The shortest paths to the same vertex are collected into consecutive elements of the list.

epaths This is a list similar to vpaths, but the vectors of the list contain the edge ids along the shortest paths, instead of the vertex ids.

nrgeo A vector in which each element is the number of shortest paths (geodesics) from `from` to the corresponding vertex in `to`.

res Deprecated

For `mean_distance()` a single number is returned if `details=FALSE`, or a named list with two entries:

`res` the mean distance as a numeric scalar

`unconnected` the number of unconnected vertex pairs, also as a numeric scalar.

`distance_table()` returns a named list with two entries:

`res` a numeric vector, the histogram of distances

`unconnected` a numeric scalar, the number of pairs for which the first vertex is not reachable from the second. In undirected and directed graphs, unorderde and ordered pairs are considered, respectively. Therefore the sum of the two entries is always $n(n-1)$ for directed graphs and $n(n-1)/2$ for undirected graphs.

Related documentation in the C library

`path_length_hist()`, `average_path_length_dijkstra()`, `edges()`, `get_eids()`, `vcount()`, `ecount()`, `is_directed()`, `get_all_shortest_paths()`, `get_all_shortest_paths_dijkstra()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

West, D.B. (1996). *Introduction to Graph Theory.* Upper Saddle River, N.J.: Prentice Hall.

See Also

Other structural.properties: `bfs()`, `component_distribution()`, `connect()`, `constraint()`, `coreness()`, `degree()`, `dfs()`, `edge_density()`, `feedback_arc_set()`, `feedback_vertex_set()`, `girth()`, `is_acyclic()`, `is_dag()`, `is_matching()`, `k_shortest_paths()`, `knn()`, `reciprocity()`, `subcomponent()`, `subgraph()`, `topo_sort()`, `transitivity()`, `unfold_tree()`, `which_multiple()`, `which_mutual()`

Other paths: `all_simple_paths()`, `diameter()`, `eccentricity()`, `graph_center()`, `radius()`

Examples

```
g <- make_ring(10)
distances(g)
shortest_paths(g, 5)
all_shortest_paths(g, 1, 6:8)
mean_distance(g)
## Weighted shortest paths
el <- matrix(
  ncol = 3, byrow = TRUE,
  c(
    1, 2, 0,
    1, 3, 2,
    1, 4, 1,
```

```
    2, 3, 0,
    2, 5, 5,
    2, 6, 2,
    3, 2, 1,
    3, 4, 1,
    3, 7, 1,
    4, 3, 0,
    4, 7, 2,
    5, 6, 2,
    5, 8, 8,
    6, 3, 2,
    6, 7, 1,
    6, 9, 1,
    6, 10, 3,
    8, 6, 1,
    8, 9, 1,
    9, 10, 4
  )
)
g2 <- add_edges(make_empty_graph(10), t(el[, 1:2]), weight = el[, 3])
distances(g2, mode = "out")
```

`diverging_pal` *Diverging palette*

Description

This is the 'PuOr' palette from https://colorbrewer2.org/. It has at most eleven colors.

Usage

```
diverging_pal(n)
```

Arguments

n	The number of colors in the palette. The maximum is eleven currently.

Details

This is similar to `sequential_pal()`, but it also puts emphasis on the mid-range values, plus the the two extreme ends. Use this palette, if you have such a quantity to mark with vertex colors.

Value

A character vector of RGB color codes.

See Also

Other palettes: `categorical_pal()`, `r_pal()`, `sequential_pal()`

Examples

```
library(igraphdata)
data(foodwebs)
fw <- foodwebs[[1]] %>%
  induced_subgraph(V(.)[ECO == 1]) %>%
  add_layout_(with_fr()) %>%
  set_vertex_attr("label", value = seq_len(gorder(.))) %>%
  set_vertex_attr("size", value = 10) %>%
  set_edge_attr("arrow.size", value = 0.3)

V(fw)$color <- scales::dscale(V(fw)$Biomass %>% cut(10), diverging_pal)
plot(fw)

data(karate)
karate <- karate %>%
  add_layout_(with_kk()) %>%
  set_vertex_attr("size", value = 10)

V(karate)$color <- scales::dscale(degree(karate) %>% cut(5), diverging_pal)
plot(karate)
```

---

diversity *Graph diversity*

Description

Calculates a measure of diversity for all vertices.

Usage

```
diversity(graph, weights = NULL, vids = V(graph))
```

Arguments

graph	The input graph. Edge directions are ignored.
weights	NULL, or the vector of edge weights to use for the computation. If NULL, then the 'weight' attibute is used. Note that this measure is not defined for unweighted graphs.
vids	The vertex ids for which to calculate the measure.

Details

The diversity of a vertex is defined as the (scaled) Shannon entropy of the weights of its incident edges:

$$D(i) = \frac{H(i)}{\log k_i}$$

and

$$H(i) = -\sum_{j=1}^{k_i} p_{ij} \log p_{ij},$$

where

$$p_{ij} = \frac{w_{ij}}{\sum_{l=1}^{k_i} V_{il}},$$

and k_i is the (total) degree of vertex i, w_{ij} is the weight of the edge(s) between vertices i and j.

For vertices with degree less than two the function returns `NaN`.

Value

A numeric vector, its length is the number of vertices.

Related documentation in the C library

`diversity()`, `vcount()`, `edges()`, `get_eids()`, `ecount()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Nathan Eagle, Michael Macy and Rob Claxton: Network Diversity and Economic Development, *Science* **328**, 1029–1031, 2010.

See Also

Centrality measures: `alpha_centrality()`, `authority_score()`, `betweenness()`, `closeness()`, `eigen_centrality()`, `harmonic_centrality()`, `hits_scores()`, `page_rank()`, `power_centrality()`, `spectrum()`, `strength()`, `subgraph_centrality()`

Examples

```
g1 <- sample_gnp(20, 2 / 20)
g2 <- sample_gnp(20, 2 / 20)
g3 <- sample_gnp(20, 5 / 20)
E(g1)$weight <- 1
E(g2)$weight <- runif(ecount(g2))
E(g3)$weight <- runif(ecount(g3))
diversity(g1)
diversity(g2)
diversity(g3)
```

`dominator_tree` *Dominator tree*

Description

Dominator tree of a directed graph.

Usage

```
dominator_tree(graph, root, mode = c("out", "in", "all", "total"))
```

Arguments

`graph`	A directed graph. If it is not a flowgraph, and it contains some vertices not reachable from the root vertex, then these vertices will be collected and returned as part of the result.
`root`	The id of the root (or source) vertex, this will be the root of the tree.
`mode`	Constant, must be '`in`' or '`out`'. If it is '`in`', then all directions are considered as opposite to the original one in the input graph.

Details

A flowgraph is a directed graph with a distinguished start (or root) vertex r, such that for any vertex v, there is a path from r to v. A vertex v dominates another vertex w (not equal to v), if every path from r to w contains v. Vertex v is the immediate dominator or w, $v = \text{idom}(w)$, if v dominates w and every other dominator of w dominates v. The edges $(\text{idom}(w), w)|w \neq r$ form a directed tree, rooted at r, called the dominator tree of the graph. Vertex v dominates vertex w if and only if v is an ancestor of w in the dominator tree.

This function implements the Lengauer-Tarjan algorithm to construct the dominator tree of a directed graph. For details see the reference below.

Value

A list with components:

dom A numeric vector giving the immediate dominators for each vertex. For vertices that are unreachable from the root, it contains `NaN`. For the root vertex itself it contains minus one.

domtree A graph object, the dominator tree. Its vertex ids are the as the vertex ids of the input graph. Isolate vertices are the ones that are unreachable from the root.

leftout A numeric vector containing the vertex ids that are unreachable from the root.

Related documentation in the C library

`dominator_tree()`, `vcount()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Thomas Lengauer, Robert Endre Tarjan: A fast algorithm for finding dominators in a flowgraph, *ACM Transactions on Programming Languages and Systems (TOPLAS)* I/1, 121–141, 1979.

See Also

Other flow: edge_connectivity(), is_min_separator(), is_separator(), max_flow(), min_cut(), min_separators(), min_st_separators(), st_cuts(), st_min_cuts(), vertex_connectivity()

Examples

```
## The example from the paper
g <- graph_from_literal(
  R -+ A:B:C, A -+ D, B -+ A:D:E, C -+ F:G, D -+ L,
  E -+ H, F -+ I, G -+ I:J, H -+ E:K, I -+ K, J -+ I,
  K -+ I:R, L -+ H
)
dtree <- dominator_tree(g, root = "R")
layout <- layout_as_tree(dtree$domtree, root = "R")
layout[, 2] <- -layout[, 2]
plot(dtree$domtree, layout = layout, vertex.label = V(dtree$domtree)$name)
```

---

dot-data .data *and* .env *pronouns*

Description

The .data and .env pronouns make it explicit where to look up attribute names when indexing V(g) or E(g), i.e. the vertex or edge sequence of a graph. These pronouns are inspired by .data and .env in rlang - thanks to Michał Bojanowski for bringing these to our attention.

The rules are simple:

- .data retrieves attributes from the graph whose vertex or edge sequence is being evaluated.
- .env retrieves variables from the calling environment.

Note that .data and .env are injected dynamically into the environment where the indexing expressions are evaluated; you cannot get access to these objects outside the context of an indexing expression. To avoid warnings printed by R CMD check when code containing .data and .env is checked, you can import .data and .env from igraph if needed. Alternatively, you can declare them explicitly with utils::globalVariables() to silence the warnings.

dyad_census *Dyad census of a graph*

Description

Classify dyads in a directed graphs. The relationship between each pair of vertices is measured. It can be in three states: mutual, asymmetric or non-existent.

Usage

```
dyad_census(graph)
```

Arguments

graph	The input graph. A warning is given if it is not directed.

Value

A named numeric vector with three elements:

mut The number of pairs with mutual connections.

asym The number of pairs with non-mutual connections.

null The number of pairs with no connection between them.

Related documentation in the C library

dyad_census(), is_directed()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Holland, P.W. and Leinhardt, S. A Method for Detecting Structure in Sociometric Data. *American Journal of Sociology*, 76, 492–513. 1970.

Wasserman, S., and Faust, K. *Social Network Analysis: Methods and Applications.* Cambridge: Cambridge University Press. 1994.

See Also

triad_census() for the same classification, but with triples.

Other graph motifs: count_motifs(), motifs(), sample_motifs()

Examples

```
g <- sample_pa(100)
dyad_census(g)
```

E *Edges of a graph*

Description

An edge sequence is a vector containing numeric edge ids, with a special class attribute that allows custom operations: selecting subsets of edges based on attributes, or graph structure, creating the intersection, union of edges, etc.

Usage

```
E(graph, P = NULL, path = NULL, directed = TRUE)
```

Arguments

graph	The graph.
P	A list of vertices to select edges via pairs of vertices. The first and second vertices select the first edge, the third and fourth the second, etc.
path	A list of vertices, to select edges along a path. Note that this only works reliable for simple graphs. If the graph has multiple edges, one of them will be chosen arbitrarily to be included in the edge sequence.
directed	Whether to consider edge directions in the P argument, for directed graphs.

Details

Edge sequences are usually used as igraph function arguments that refer to edges of a graph.

An edge sequence is tied to the graph it refers to: it really denoted the specific edges of that graph, and cannot be used together with another graph.

An edge sequence is most often created by the E() function. The result includes edges in increasing edge id order by default (if. none of the P and path arguments are used). An edge sequence can be indexed by a numeric vector, just like a regular R vector. See links to other edge sequence operations below.

Value

An edge sequence of the graph.

Indexing edge sequences

Edge sequences mostly behave like regular vectors, but there are some additional indexing operations that are specific for them; e.g. selecting edges based on graph structure, or based on edge attributes. See [.igraph.es for details.

Querying or setting attributes

Edge sequences can be used to query or set attributes for the edges in the sequence. See $.igraph.es() for details.

Related documentation in the C library

edges(), get_eids(), vcount(), ecount()

See Also

Other vertex and edge sequences: V(), as_ids(), igraph-es-attributes, igraph-es-indexing, igraph-es-indexing2, igraph-vs-attributes, igraph-vs-indexing, igraph-vs-indexing2, print.igraph.es(), print.igraph.vs()

Examples

```
# Edges of an unnamed graph
g <- make_ring(10)
E(g)

# Edges of a named graph
g2 <- make_ring(10) %>%
  set_vertex_attr("name", value = letters[1:10])
E(g2)
```

---

each_edge *Rewires the endpoints of the edges of a graph to a random vertex*

Description

This function can be used together with rewire(). This method rewires the endpoints of the edges with a constant probability uniformly randomly to a new vertex in a graph.

Usage

```
each_edge(
  prob,
  loops = FALSE,
  multiple = FALSE,
  mode = c("all", "out", "in", "total")
)
```

Arguments

prob	The rewiring probability, a real number between zero and one.
loops	Logical scalar, whether loop edges are allowed in the rewired graph.
multiple	Logical scalar, whether multiple edges are allowed in the generated graph.
mode	Character string, specifies which endpoint of the edges to rewire in directed graphs. 'all' rewires both endpoints, 'in' rewires the start (tail) of each directed edge, 'out' rewires the end (head) of each directed edge. Ignored for undirected graphs.

Details

Note that this method might create graphs with multiple and/or loop edges.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Other rewiring functions: keeping_degseq(), rewire()

Examples

```
# Some random shortcuts shorten the distances on a lattice
g <- make_lattice(length = 100, dim = 1, nei = 5)
mean_distance(g)
g <- rewire(g, each_edge(prob = 0.05))
mean_distance(g)

# Rewiring the start of each directed edge preserves the in-degree distribution
# but not the out-degree distribution
g <- sample_pa(1000)
g2 <- g %>% rewire(each_edge(mode = "in", multiple = TRUE, prob = 0.2))
degree(g, mode = "in") == degree(g2, mode = "in")
```

---

eccentricity *Eccentricity of the vertices in a graph*

Description

The eccentricity of a vertex is its shortest path distance from the farthest other node in the graph.

Usage

```
eccentricity(
  graph,
  vids = V(graph),
  ...,
  weights = NULL,
  mode = c("all", "out", "in", "total")
)
```

Arguments

graph	The input graph, it can be directed or undirected.
vids	The vertices for which the eccentricity is calculated.
...	These dots are for future extensions and must be empty.

`weights` Possibly a numeric vector giving edge weights. If this is `NULL` and the graph has a `weight` edge attribute, then the attribute is used. If this is `NA` then no weights are used (even if the graph has a `weight` attribute). In a weighted graph, the length of a path is the sum of the weights of its constituent edges.

`mode` Character constant, gives whether the shortest paths to or from the given vertices should be calculated for directed graphs. If `out` then the shortest paths *from* the vertex, if `in` then *to* it will be considered. If `all`, the default, then the graph is treated as undirected, i.e. edge directions are not taken into account. This argument is ignored for undirected graphs.

Details

The eccentricity of a vertex is calculated by measuring the shortest distance from (or to) the vertex, to (or from) all vertices in the graph, and taking the maximum.

This implementation ignores vertex pairs that are in different components. Isolate vertices have eccentricity zero.

Value

`eccentricity()` returns a numeric vector, containing the eccentricity score of each given vertex.

Related documentation in the C library

`eccentricity_dijkstra()`, `vcount()`, `edges()`, `get_eids()`, `ecount()`

References

Harary, F. Graph Theory. Reading, MA: Addison-Wesley, p. 35, 1994.

See Also

`radius()` for a related concept, `distances()` for general shortest path calculations.

Other paths: `all_simple_paths()`, `diameter()`, `distance_table()`, `graph_center()`, `radius()`

Examples

```
g <- make_star(10, mode = "undirected")
eccentricity(g)
```

edge *Helper function for adding and deleting edges*

Description

This is a helper function that simplifies adding and deleting edges to/from graphs.

Usage

```
edge(...)

edges(...)
```

Arguments

...	See details below.

Details

`edges()` is an alias for `edge()`.

When adding edges via `+`, all unnamed arguments of `edge()` (or `edges()`) are concatenated, and then passed to `add_edges()`. They are interpreted as pairs of vertex ids, and an edge will added between each pair. Named arguments will be used as edge attributes for the new edges.

When deleting edges via `-`, all arguments of `edge()` (or `edges()`) are concatenated via `c()` and passed to `delete_edges()`.

Value

A special object that can be used with together with igraph graphs and the plus and minus operators.

See Also

Other functions for manipulating graph structure: `+.igraph()`, `add_edges()`, `add_vertices()`, `complementer()`, `compose()`, `connect()`, `contract()`, `delete_edges()`, `delete_vertices()`, `difference()`, `difference.igraph()`, `disjoint_union()`, `igraph-minus`, `intersection()`, `intersection.igraph()`, `path()`, `permute()`, `rep.igraph()`, `reverse_edges()`, `simplify()`, `transitive_closure()`, `union()`, `union.igraph()`, `vertex()`

Examples

```
g <- make_ring(10) %>%
  set_edge_attr("color", value = "red")

g <- g + edge(1, 5, color = "green") +
  edge(2, 6, color = "blue") -
  edge("8|9")

E(g)[[]]
```

```
g %>%
  add_layout_(in_circle()) %>%
  plot()

g <- make_ring(10) + edges(1:10)
plot(g)
```

---

edge_attr *Query edge attributes of a graph*

Description

Query edge attributes of a graph

Usage

```
edge_attr(graph, name, index = E(graph))
```

Arguments

graph	The graph
name	The name of the attribute to query. If missing, then all edge attributes are returned in a list.
index	An optional edge sequence to query edge attributes for a subset of edges.

Value

The value of the edge attribute, or the list of all edge attributes if name is missing.

Related documentation in the C library

edges(), get_eids(), vcount(), ecount()

See Also

Vertex, edge and graph attributes: delete_edge_attr(), delete_graph_attr(), delete_vertex_attr(), edge_attr<-(), edge_attr_names(), graph_attr(), graph_attr<-(), graph_attr_names(), igraph-attribute-combination, igraph-dollar, igraph-vs-attributes, set_edge_attr(), set_graph_attr(), set_vertex_attr(), set_vertex_attrs(), vertex_attr(), vertex_attr<-(), vertex_attr_names()

Examples

```
g <- make_ring(10) %>%
  set_edge_attr("weight", value = 1:10) %>%
  set_edge_attr("color", value = "red")
g
plot(g, edge.width = E(g)$weight)
```

edge_attr_names *List names of edge attributes*

Description

List names of edge attributes

Usage

```
edge_attr_names(graph)
```

Arguments

graph	The graph.

Value

Character vector, the names of the edge attributes.

See Also

Vertex, edge and graph attributes: delete_edge_attr(), delete_graph_attr(), delete_vertex_attr(), edge_attr(), edge_attr<-(), graph_attr(), graph_attr<-(), graph_attr_names(), igraph-attribute-combinatio igraph-dollar, igraph-vs-attributes, set_edge_attr(), set_graph_attr(), set_vertex_attr(), set_vertex_attrs(), vertex_attr(), vertex_attr<-(), vertex_attr_names()

Examples

```
g <- make_ring(10) %>%
  set_edge_attr("label", value = letters[1:10])
edge_attr_names(g)
plot(g)
```

edge_attr<- *Set one or more edge attributes*

Description

Set one or more edge attributes

Usage

```
edge_attr(graph, name, index = E(graph)) <- value
```

Arguments

graph	The graph.
name	The name of the edge attribute to set. If missing, then value must be a named list, and its entries are set as edge attributes.
index	An optional edge sequence to set the attributes of a subset of edges.
value	The new value of the attribute(s) for all (or index) edges.

Value

The graph, with the edge attribute(s) added or set.

See Also

Vertex, edge and graph attributes: delete_edge_attr(), delete_graph_attr(), delete_vertex_attr(), edge_attr(), edge_attr_names(), graph_attr(), graph_attr<-(), graph_attr_names(), igraph-attribute-combir igraph-dollar, igraph-vs-attributes, set_edge_attr(), set_graph_attr(), set_vertex_attr(), set_vertex_attrs(), vertex_attr(), vertex_attr<-(), vertex_attr_names()

Examples

```
g <- make_ring(10)
edge_attr(g) <- list(
  name = LETTERS[1:10],
  color = rep("green", gsize(g))
)
edge_attr(g, "label") <- E(g)$name
g
plot(g)
```

---

edge_connectivity *Edge connectivity*

Description

The edge connectivity of a graph or two vertices, this is recently also called group adhesion.

Usage

```
edge_connectivity(graph, source = NULL, target = NULL, checks = TRUE)

edge_disjoint_paths(graph, source = NULL, target = NULL)

adhesion(graph, checks = TRUE)
```

Arguments

graph	The input graph.
source	The id of the source vertex, for edge_connectivity() it can be NULL, see details below.
target	The id of the target vertex, for edge_connectivity() it can be NULL, see details below.
checks	Logical constant. Whether to check that the graph is connected and also the degree of the vertices. If the graph is not (strongly) connected then the connectivity is obviously zero. Otherwise if the minimum degree is one then the edge connectivity is also one. It is a good idea to perform these checks, as they can be done quickly compared to the connectivity calculation itself. They were suggested by Peter McMahan, thanks Peter.

Value

A scalar real value.

edge_connectivity() Edge connectivity

The edge connectivity of a pair of vertices (source and target) is the minimum number of edges needed to remove to eliminate all (directed) paths from source to target. edge_connectivity() calculates this quantity if both the source and target arguments are given (and not NULL).

The edge connectivity of a graph is the minimum of the edge connectivity of every (ordered) pair of vertices in the graph. edge_connectivity() calculates this quantity if neither the source nor the target arguments are given (i.e. they are both NULL).

edge_disjoint_paths() The maximum number of edge-disjoint paths between two vertices

A set of paths between two vertices is called edge-disjoint if they do not share any edges. The maximum number of edge-disjoint paths are calculated by this function using maximum flow techniques. Directed paths are considered in directed graphs.

A set of edge disjoint paths between two vertices is a set of paths between them containing no common edges. The maximum number of edge disjoint paths between two vertices is the same as their edge connectivity.

When there are no direct edges between the source and the target, the number of vertex-disjoint paths is the same as the vertex connectivity of the two vertices. When some edges are present, each one of them contributes one extra path.

adhesion() Adhesion of a graph

The adhesion of a graph is the minimum number of edges needed to remove to obtain a graph which is not strongly connected. This is the same as the edge connectivity of the graph.

All three functions

The three functions documented on this page calculate similar properties, more precisely the most general is edge_connectivity(), the others are included only for having more descriptive function names.

Related documentation in the C library

st_edge_connectivity(), edge_connectivity(), vcount(), edge_disjoint_paths(), adhesion()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Douglas R. White and Frank Harary (2001): The cohesiveness of blocks in social networks: node connectivity and conditional density, Sociological Methodology, vol. 31, 2001, pp. 305–59.

See Also

Other flow: dominator_tree(), is_min_separator(), is_separator(), max_flow(), min_cut(), min_separators(), min_st_separators(), st_cuts(), st_min_cuts(), vertex_connectivity()

Examples

```
g <- sample_pa(100, m = 1)
g2 <- sample_pa(100, m = 5)
edge_connectivity(g, 100, 1)
edge_connectivity(g2, 100, 1)
edge_disjoint_paths(g2, 100, 1)

g <- sample_gnp(50, 5 / 50)
g <- as_directed(g)
g <- induced_subgraph(g, subcomponent(g, 1))
adhesion(g)
```

---

edge_density *Graph density*

Description

The density of a graph is the ratio of the actual number of edges and the largest possible number of edges in the graph, assuming that no multi-edges are present.

Usage

```
edge_density(graph, loops = FALSE)
```

Arguments

graph	The input graph.
loops	Logical constant, whether loop edges may exist in the graph. This affects the calculation of the largest possible number of edges in the graph. If this parameter is set to FALSE yet the graph contains self-loops, the result will not be meaningful.

Details

The concept of density is ill-defined for multigraphs. Note that this function does not check whether the graph has multi-edges and will return meaningless results for such graphs.

Value

A real constant. This function returns NaN (=0.0/0.0) for an empty graph with zero vertices.

Related documentation in the C library

density()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Wasserman, S., and Faust, K. (1994). Social Network Analysis: Methods and Applications. Cambridge: Cambridge University Press.

See Also

vcount(), ecount(), simplify() to get rid of the multiple and/or loop edges.

Other structural.properties: bfs(), component_distribution(), connect(), constraint(), coreness(), degree(), dfs(), distance_table(), feedback_arc_set(), feedback_vertex_set(), girth(), is_acyclic(), is_dag(), is_matching(), k_shortest_paths(), knn(), reciprocity(), subcomponent(), subgraph(), topo_sort(), transitivity(), unfold_tree(), which_multiple(), which_mutual()

Examples

```
edge_density(make_empty_graph(n = 10)) # empty graphs have density 0
edge_density(make_full_graph(n = 10)) # complete graphs have density 1
edge_density(sample_gnp(n = 100, p = 0.4)) # density will be close to p

# loop edges
g <- make_graph(c(1, 2, 2, 2, 2, 3)) # graph with a self-loop
edge_density(g, loops = FALSE) # this is wrong!!!
edge_density(g, loops = TRUE) # this is right!!!
edge_density(simplify(g), loops = FALSE) # this is also right, but different
```

`eigen_centrality` *Eigenvector centrality of vertices*

Description

`eigen_centrality()` takes a graph (`graph`) and returns the eigenvector centralities of the vertices `v` within it.

Usage

```
eigen_centrality(
  graph,
  directed = FALSE,
  scale = deprecated(),
  weights = NULL,
  options = arpack_defaults()
)
```

Arguments

`graph`	Graph to be analyzed.
`directed`	Logical scalar, whether to consider direction of the edges in directed graphs. It is ignored for undirected graphs.
`scale`	**[Deprecated]** Normalization will always take place.
`weights`	A numerical vector or `NULL`. This argument can be used to give edge weights for calculating the weighted eigenvector centrality of vertices. If this is `NULL` and the graph has a `weight` edge attribute then that is used. If `weights` is a numerical vector then it is used, even if the graph has a `weight` edge attribute. If this is `NA`, then no edge weights are used (even if the graph has a `weight` edge attribute). Note that if there are negative edge weights and the direction of the edges is considered, then the eigenvector might be complex. In this case only the real part is reported. This function interprets weights as connection strength. Higher weights spread the centrality better.
`options`	A named list, to override some ARPACK options. See `arpack()` for details.

Details

Eigenvector centrality scores correspond to the values of the principal eigenvector of the graph's adjacency matrix; these scores may, in turn, be interpreted as arising from a reciprocal process in which the centrality of each actor is proportional to the sum of the centralities of those actors to whom he or she is connected. In general, vertices with high eigenvector centralities are those which are connected to many other vertices which are, in turn, connected to many others (and so on). The perceptive may realize that this implies that the largest values will be obtained by individuals in large cliques (or high-density substructures). This is also intelligible from an algebraic point of view, with the first eigenvector being closely related to the best rank-1 approximation of the adjacency matrix

(a relationship which is easy to see in the special case of a diagonalizable symmetric real matrix via the $SLS^{-}1$ decomposition).

The adjacency matrix used in the eigenvector centrality calculation assumes that loop edges are counted *twice* in undirected graphs; this is because each loop edge has *two* endpoints that are both connected to the same vertex, and you could traverse the loop edge via either endpoint.

In the directed case, the left eigenvector of the adjacency matrix is calculated. In other words, the centrality of a vertex is proportional to the sum of centralities of vertices pointing to it.

Eigenvector centrality is meaningful only for (strongly) connected graphs. Undirected graphs that are not connected should be decomposed into connected components, and the eigenvector centrality calculated for each separately. This function does not verify that the graph is connected. If it is not, in the undirected case the scores of all but one component will be zeros.

Also note that the adjacency matrix of a directed acyclic graph or the adjacency matrix of an empty graph does not possess positive eigenvalues, therefore the eigenvector centrality is not defined for these graphs. igraph will return an eigenvalue of zero in such cases. The eigenvector centralities will all be equal for an empty graph and will all be zeros for a directed acyclic graph. Such pathological cases can be detected by checking whether the eigenvalue is very close to zero.

From igraph version 0.5 this function uses ARPACK for the underlying computation, see `arpack()` for more about ARPACK in igraph.

Value

A named list with components:

vector A vector containing the centrality scores.

value The eigenvalue corresponding to the calculated eigenvector, i.e. the centrality scores.

options A named list, information about the underlying ARPACK computation. See `arpack()` for the details.

Related documentation in the C library

`eigenvector_centrality()`, `vcount()`, `edges()`, `get_eids()`, `ecount()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com> and Carter T. Butts (https://www.faculty.uci.edu/profile.cfm?faculty_id=5057) for the manual page.

References

Bonacich, P. (1987). Power and Centrality: A Family of Measures. *American Journal of Sociology*, 92, 1170-1182.

See Also

Centrality measures: `alpha_centrality()`, `authority_score()`, `betweenness()`, `closeness()`, `diversity()`, `harmonic_centrality()`, `hits_scores()`, `page_rank()`, `power_centrality()`, `spectrum()`, `strength()`, `subgraph_centrality()`

Examples

```
# Generate some test data
g <- make_ring(10, directed = FALSE)
# Compute eigenvector centrality scores
eigen_centrality(g)
```

`embed_adjacency_matrix` *Spectral Embedding of Adjacency Matrices*

Description

Spectral decomposition of the adjacency matrices of graphs.

Usage

```
embed_adjacency_matrix(
  graph,
  no,
  weights = NULL,
  which = c("lm", "la", "sa"),
  scaled = TRUE,
  cvec = strength(graph, weights = weights)/(vcount(graph) - 1),
  options = arpack_defaults()
)
```

Arguments

`graph`	The input graph, directed or undirected.
`no`	An integer scalar. This value is the embedding dimension of the spectral embedding. Should be smaller than the number of vertices. The largest no-dimensional non-zero singular values are used for the spectral embedding.
`weights`	Optional positive weight vector for calculating a weighted embedding. If the graph has a `weight` edge attribute, then this is used by default. In a weighted embedding, the edge weights are used instead of the binary adjacencny matrix.
`which`	Which eigenvalues (or singular values, for directed graphs) to use. 'lm' means the ones with the largest magnitude, 'la' is the ones (algebraic) largest, and 'sa' is the (algebraic) smallest eigenvalues. The default is 'lm'. Note that for directed graphs 'la' and 'lm' are the equivalent, because the singular values are used for the ordering.
`scaled`	Logical scalar, if `FALSE`, then U and V are returned instead of X and Y.
`cvec`	A numeric vector, its length is the number vertices in the graph. This vector is added to the diagonal of the adjacency matrix.
`options`	A named list containing the parameters for the SVD computation algorithm in ARPACK. By default, the list of values is assigned the values given by `arpack_defaults()`.

Details

This function computes a no-dimensional Euclidean representation of the graph based on its adjacency matrix, A. This representation is computed via the singular value decomposition of the adjacency matrix, $A = UDV^T$.In the case, where the graph is a random dot product graph generated using latent position vectors in R^{no} for each vertex, the embedding will provide an estimate of these latent vectors.

For undirected graphs the latent positions are calculated as $X = U^{no}D^{1/2}$, where U^{no} equals to the first no columns of U, and $D^{1/2}$ is a diagonal matrix containing the top no singular values on the diagonal.

For directed graphs the embedding is defined as the pair $X = U^{no}D^{1/2}$ and $Y = V^{no}D^{1/2}$. (For undirected graphs $U = V$, so it is enough to keep one of them.)

Value

A list containing with entries:

X Estimated latent positions, an n times no matrix, n is the number of vertices.

Y NULL for undirected graphs, the second half of the latent positions for directed graphs, an n times no matrix, n is the number of vertices.

D The eigenvalues (for undirected graphs) or the singular values (for directed graphs) calculated by the algorithm.

options A named list, information about the underlying ARPACK computation. See arpack() for the details.

Related documentation in the C library

adjacency_spectral_embedding(), strength(), vcount(), edges(), get_eids(), ecount()

References

Sussman, D.L., Tang, M., Fishkind, D.E., Priebe, C.E. A Consistent Adjacency Spectral Embedding for Stochastic Blockmodel Graphs, *Journal of the American Statistical Association*, Vol. 107(499), 2012

See Also

sample_dot_product()

Other embedding: dim_select(), embed_laplacian_matrix()

Examples

```
## A small graph
lpvs <- matrix(rnorm(200), 20, 10)
lpvs <- apply(lpvs, 2, function(x) {
  return(abs(x) / sqrt(sum(x^2)))
})
RDP <- sample_dot_product(lpvs)
embed <- embed_adjacency_matrix(RDP, 5)
```

`embed_laplacian_matrix` *Spectral Embedding of the Laplacian of a Graph*

Description

Spectral decomposition of Laplacian matrices of graphs.

Usage

```
embed_laplacian_matrix(
  graph,
  no,
  weights = NULL,
  which = c("lm", "la", "sa"),
  type = c("default", "D-A", "DAD", "I-DAD", "OAP"),
  scaled = TRUE,
  options = arpack_defaults()
)
```

Arguments

`graph`	The input graph, directed or undirected.
`no`	An integer scalar. This value is the embedding dimension of the spectral embedding. Should be smaller than the number of vertices. The largest `no`-dimensional non-zero singular values are used for the spectral embedding.
`weights`	Optional positive weight vector for calculating a weighted embedding. If the graph has a `weight` edge attribute, then this is used by default. For weighted embedding, edge weights are used instead of the binary adjacency matrix, and vertex strength (see `strength()`) is used instead of the degrees.
`which`	Which eigenvalues (or singular values, for directed graphs) to use. 'lm' means the ones with the largest magnitude, 'la' is the ones (algebraic) largest, and 'sa' is the (algebraic) smallest eigenvalues. The default is 'lm'. Note that for directed graphs 'la' and 'lm' are the equivalent, because the singular values are used for the ordering.
`type`	The type of the Laplacian to use. Various definitions exist for the Laplacian of a graph, and one can choose between them with this argument. Possible values: `D-A` means $D - A$ where D is the degree matrix and A is the adjacency matrix; `DAD` means $D^{1/2}$ times A times $D^{1/2}D^1/2$, $D^{1/2}$ is the inverse of the square root of the degree matrix; `I-DAD` means $I - D^{1/2}$, where I is the identity matrix. `OAP` is $O^{1/2}AP^{1/2}$, where $O^{1/2}$ is the inverse of the square root of the out-degree matrix and $P^{1/2}$ is the same for the in-degree matrix. `OAP` is not defined for undirected graphs, and is the only defined type for directed graphs. The default (i.e. type `default`) is to use `D-A` for undirected graphs and `OAP` for directed graphs.

`scaled` Logical scalar, if `FALSE`, then U and V are returned instead of X and Y.

`options` A named list containing the parameters for the SVD computation algorithm in ARPACK. By default, the list of values is assigned the values given by `arpack_defaults()`.

Details

This function computes a no-dimensional Euclidean representation of the graph based on its Laplacian matrix, L. This representation is computed via the singular value decomposition of the Laplacian matrix.

They are essentially doing the same as `embed_adjacency_matrix()`, but work on the Laplacian matrix, instead of the adjacency matrix.

Value

A list containing with entries:

X Estimated latent positions, an `n` times `no` matrix, `n` is the number of vertices.

Y `NULL` for undirected graphs, the second half of the latent positions for directed graphs, an `n` times `no` matrix, `n` is the number of vertices.

D The eigenvalues (for undirected graphs) or the singular values (for directed graphs) calculated by the algorithm.

options A named list, information about the underlying ARPACK computation. See `arpack()` for the details.

Related documentation in the C library

`laplacian_spectral_embedding()`, `is_directed()`, `edges()`, `get_eids()`, `vcount()`, `ecount()`

Author(s)

Gabor Csardi `<csardi.gabor@gmail.com>`

References

Sussman, D.L., Tang, M., Fishkind, D.E., Priebe, C.E. A Consistent Adjacency Spectral Embedding for Stochastic Blockmodel Graphs, *Journal of the American Statistical Association*, Vol. 107(499), 2012

See Also

`embed_adjacency_matrix()`, `sample_dot_product()`

Other embedding: `dim_select()`, `embed_adjacency_matrix()`

Examples

```
## A small graph
lpvs <- matrix(rnorm(200), 20, 10)
lpvs <- apply(lpvs, 2, function(x) {
  return(abs(x) / sqrt(sum(x^2)))
})
RDP <- sample_dot_product(lpvs)
embed <- embed_laplacian_matrix(RDP, 5)
```

---

ends *Incident vertices of some graph edges*

Description

Incident vertices of some graph edges

Usage

```
ends(graph, es, names = TRUE)
```

Arguments

graph	The input graph
es	The sequence of edges to query
names	Whether to return vertex names or numeric vertex ids. By default vertex names are used.

Value

A two column matrix of vertex names or vertex ids.

Related documentation in the C library

edges(), get_eids(), vcount(), ecount()

See Also

Other structural queries: [.igraph(), [[.igraph(), adjacent_vertices(), are_adjacent(), get_edge_ids(), gorder(), gsize(), head_of(), incident(), incident_edges(), is_directed(), neighbors(), tail_of()

Examples

```
g <- make_ring(5)
ends(g, E(g))
```

feedback_arc_set *Finding a feedback arc set in a graph*

Description

A feedback arc set of a graph is a subset of edges whose removal breaks all cycles in the graph.

Usage

```
feedback_arc_set(graph, weights = NULL, algo = c("approx_eades", "exact_ip"))
```

Arguments

graph	The input graph
weights	Potential edge weights. If the graph has an edge attribute called 'weight', and this argument is NULL, then the edge attribute is used automatically. The goal of the feedback arc set problem is to find a feedback arc set with the smallest total weight.
algo	Specifies the algorithm to use. "exact_ip" solves the feedback arc set problem with an exact integer programming algorithm that guarantees that the total weight of the removed edges is as small as possible. "approx_eades" uses a fast (linear-time) approximation algorithm from Eades, Lin and Smyth. "exact" is an alias to "exact_ip" while "approx" is an alias to "approx_eades".

Details

Feedback arc sets are typically used in directed graphs. The removal of a feedback arc set of a directed graph ensures that the remaining graph is a directed acyclic graph (DAG). For undirected graphs, the removal of a feedback arc set ensures that the remaining graph is a forest (i.e. every connected component is a tree).

Value

An edge sequence (by default, but see the return.vs.es option of igraph_options()) containing the feedback arc set.

Related documentation in the C library

feedback_arc_set(), edges(), ecount(), get_eids(), vcount()

References

Peter Eades, Xuemin Lin and W.F.Smyth: A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters* 47:6, pp. 319-323, 1993

See Also

Other structural.properties: bfs(), component_distribution(), connect(), constraint(), coreness(), degree(), dfs(), distance_table(), edge_density(), feedback_vertex_set(), girth(), is_acyclic(), is_dag(), is_matching(), k_shortest_paths(), knn(), reciprocity(), subcomponent(), subgraph(), topo_sort(), transitivity(), unfold_tree(), which_multiple(), which_mutual()

Graph cycles: feedback_vertex_set(), find_cycle(), girth(), has_eulerian_path(), is_acyclic(), is_dag(), simple_cycles()

Examples

```
g <- sample_gnm(20, 40, directed = TRUE)
feedback_arc_set(g)
feedback_arc_set(g, algo = "approx_eades")
```

---

feedback_vertex_set *Finding a feedback vertex set in a graph*

Description

[Experimental]

A feedback vertex set of a graph is a subset of vertices whose removal breaks all cycles in the graph. Finding a *minimum* feedback vertex set is an NP-complete problem, both on directed and undirected graphs.

Usage

```
feedback_vertex_set(graph, weights = NULL, algo = c("exact_ip"))
```

Arguments

graph	The input graph
weights	Potential vertex weights. If the graph has a vertex attribute called 'weight', and this argument is NULL, then the vertex attribute is used automatically. The goal of the feedback vertex set problem is to find a feedback vertex set with the smallest total weight.
algo	Specifies the algorithm to use. Currently, "exact_ip", which solves the feedback vertex set problem with an exact integer programming approach, is the only option.

Value

A vertex sequence (by default, but see the return.vs.es option of igraph_options()) containing the feedback vertex set.

Related documentation in the C library

feedback_vertex_set(), vcount()

See Also

Other structural.properties: bfs(), component_distribution(), connect(), constraint(), coreness(), degree(), dfs(), distance_table(), edge_density(), feedback_arc_set(), girth(), is_acyclic(), is_dag(), is_matching(), k_shortest_paths(), knn(), reciprocity(), subcomponent(), subgraph(), topo_sort(), transitivity(), unfold_tree(), which_multiple(), which_mutual()

Graph cycles: feedback_arc_set(), find_cycle(), girth(), has_eulerian_path(), is_acyclic(), is_dag(), simple_cycles()

Examples

```
g <- make_lattice(c(3,3))
feedback_vertex_set(g)
```

---

find_cycle *Finds a cycle in a graph, if there is one*

Description

[Experimental]

This function returns a cycle of the graph, in terms of both its vertices and edges. If the graph is acyclic, it returns empty vertex and edge sequences.

Use is_acyclic() to determine if a graph has cycles, without returning a specific cycle.

Usage

```
find_cycle(graph, mode = c("out", "in", "all", "total"))
```

Arguments

graph	The input graph.
mode	Character constant specifying how to handle directed graphs. out follows edge directions, in follows edges in the reverse direction, and all ignores edge directions. Ignored in undirected graphs.

Value

A list of integer vectors, each integer vector is a path from the source vertex to one of the target vertices. A path is given by its vertex ids.

Related documentation in the C library

find_cycle(), ecount(), vcount(), edges(), get_eids()

See Also

Graph cycles: feedback_arc_set(), feedback_vertex_set(), girth(), has_eulerian_path(), is_acyclic(), is_dag(), simple_cycles()

Examples

```
g <- make_lattice(c(3, 3))
find_cycle(g)

# Empty results are returned for acyclic graphs
find_cycle(sample_tree(5))
```

`fit_hrg` *Fit a hierarchical random graph model*

Description

`fit_hrg()` fits a HRG to a given graph. It takes the specified `steps` number of MCMC steps to perform the fitting, or a convergence criteria if the specified number of steps is zero. `fit_hrg()` can start from a given HRG, if this is given in the `hrg()` argument and the `start` argument is `TRUE`. It can be converted to the `hclust` class using `as.hclust()` provided in this package.

Usage

```
fit_hrg(graph, hrg = NULL, start = FALSE, steps = 0)
```

Arguments

`graph`	The graph to fit the model to. Edge directions are ignored in directed graphs.
`hrg`	A hierarchical random graph model, in the form of an `igraphHRG` object. `fit_hrg()` allows this to be `NULL`, in which case a random starting point is used for the fitting.
`start`	Logical, whether to start the fitting/sampling from the supplied `igraphHRG` object, or from a random starting point.
`steps`	The number of MCMC steps to make. If this is zero, then the MCMC procedure is performed until convergence.

Value

`fit_hrg()` returns an `igraphHRG` object. This is a list with the following members:

left Vector that contains the left children of the internal tree vertices. The first vertex is always the root vertex, so the first element of the vector is the left child of the root vertex. Internal vertices are denoted with negative numbers, starting from -1 and going down, i.e. the root vertex is -1. Leaf vertices are denoted by non-negative number, starting from zero and up.

right Vector that contains the right children of the vertices, with the same encoding as the `left` vector.

prob The connection probabilities attached to the internal vertices, the first number belongs to the root vertex (i.e. internal vertex -1), the second to internal vertex -2, etc.

edges The number of edges in the subtree below the given internal vertex.

vertices The number of vertices in the subtree below the given internal vertex, including itself.

Related documentation in the C library

hrg_fit(), vcount()

References

A. Clauset, C. Moore, and M.E.J. Newman. Hierarchical structure and the prediction of missing links in networks. *Nature* 453, 98–101 (2008);

A. Clauset, C. Moore, and M.E.J. Newman. Structural Inference of Hierarchies in Networks. In E. M. Airoldi et al. (Eds.): ICML 2006 Ws, *Lecture Notes in Computer Science* 4503, 1–13. Springer-Verlag, Berlin Heidelberg (2007).

See Also

Other hierarchical random graph functions: consensus_tree(), hrg(), hrg-methods, hrg_tree(), predict_edges(), print.igraphHRG(), print.igraphHRGConsensus(), sample_hrg()

Examples

```
## A graph with two dense groups
g <- sample_gnp(10, p = 1 / 2) + sample_gnp(10, p = 1 / 2)
hrg <- fit_hrg(g)
hrg
summary(as.hclust(hrg))

## The consensus tree for it
consensus_tree(g, hrg = hrg, start = TRUE)

## Prediction of missing edges
g2 <- make_full_graph(4) + (make_full_graph(4) - path(1, 2))
predict_edges(g2)
```

---

fit_power_law *Fitting a power-law distribution function to discrete data*

Description

fit_power_law() fits a power-law distribution to a data set.

Usage

```
fit_power_law(
  x,
  xmin = NULL,
  start = 2,
  force.continuous = FALSE,
  implementation = c("plfit", "R.mle"),
```

```
  p.value = FALSE,
  p.precision = NULL,
  ...
)
```

Arguments

x	The data to fit, a numeric vector. For implementation 'R.mle' the data must be integer values. For the 'plfit' implementation non-integer values might be present and then a continuous power-law distribution is fitted.
xmin	Numeric scalar, or NULL. The lower bound for fitting the power-law. If NULL, the smallest value in x will be used for the 'R.mle' implementation, and its value will be automatically determined for the 'plfit' implementation. This argument makes it possible to fit only the tail of the distribution.
start	Numeric scalar. The initial value of the exponent for the minimizing function, for the 'R.mle' implementation. Usually it is safe to leave this untouched.
force.continuous	Logical scalar. Whether to force a continuous distribution for the 'plfit' implementation, even if the sample vector contains integer values only (by chance). If this argument is false, igraph will assume a continuous distribution if at least one sample is non-integer and assume a discrete distribution otherwise.
implementation	Character scalar. Which implementation to use. See details below.
p.value	**[Experimental]** Set to TRUE to compute the p-value with implementation = "plfit".
p.precision	**[Experimental]** The desired precision of the p-value calculation. The precision ultimately depends on the number of resampling attempts. The number of resampling trials is determined by 0.25 divided by the square of the required precision. For instance, a required precision of 0.01 means that 2500 samples will be drawn.
...	Additional arguments, passed to the maximum likelihood optimizing function, stats4::mle(), if the 'R.mle' implementation is chosen. It is ignored by the 'plfit' implementation.

Details

This function fits a power-law distribution to a vector containing samples from a distribution (that is assumed to follow a power-law of course). In a power-law distribution, it is generally assumed that $P(X = x)$ is proportional to $x^{-\alpha}$, where x is a positive number and α is greater than 1. In many real-world cases, the power-law behaviour kicks in only above a threshold value x_{min}. The goal of this function is to determine α if x_{min} is given, or to determine x_{min} and the corresponding value of α.

fit_power_law() provides two maximum likelihood implementations. If the implementation argument is 'R.mle', then the BFGS optimization (see stats4::mle()) algorithm is applied. The additional arguments are passed to the mle function, so it is possible to change the optimization method and/or its parameters. This implementation can *not* to fit the x_{min} argument, so use the 'plfit' implementation if you want to do that.

The 'plfit' implementation also uses the maximum likelihood principle to determine α for a given $x_{\min}$; When $x_{\min}$ is not given in advance, the algorithm will attempt to find its optimal value for which the p-value of a Kolmogorov-Smirnov test between the fitted distribution and the original sample is the largest. The function uses the method of Clauset, Shalizi and Newman to calculate the parameters of the fitted distribution. See references below for the details.

[Experimental]

Pass `p.value = TRUE` to include the p-value in the output. This is not returned by default because the computation may be slow.

Value

Depends on the `implementation` argument. If it is 'R.mle', then an object with class 'mle'. It can be used to calculate confidence intervals and log-likelihood. See `stats4::mle-class()` for details.

If `implementation` is 'plfit', then the result is a named list with entries:

continuous Logical scalar, whether the fitted power-law distribution was continuous or discrete.

alpha Numeric scalar, the exponent of the fitted power-law distribution.

xmin Numeric scalar, the minimum value from which the power-law distribution was fitted. In other words, only the values larger than `xmin` were used from the input vector.

logLik Numeric scalar, the log-likelihood of the fitted parameters.

KS.stat Numeric scalar, the test statistic of a Kolmogorov-Smirnov test that compares the fitted distribution with the input vector. Smaller scores denote better fit.

KS.p Only for `p.value = TRUE`. Numeric scalar, the p-value of the Kolmogorov-Smirnov test. Small p-values (less than 0.05) indicate that the test rejected the hypothesis that the original data could have been drawn from the fitted power-law distribution.

Author(s)

Tamas Nepusz <ntamas@gmail.com> and Gabor Csardi <csardi.gabor@gmail.com>

References

Power laws, Pareto distributions and Zipf's law, M. E. J. Newman, *Contemporary Physics*, 46, 323-351, 2005.

Aaron Clauset, Cosma R .Shalizi and Mark E.J. Newman: Power-law distributions in empirical data. SIAM Review 51(4):661-703, 2009.

See Also

`stats4::mle()`

Examples

```
# This should approximately yield the correct exponent 3
g <- sample_pa(1000) # increase this number to have a better estimate
d <- degree(g, mode = "in")
fit1 <- fit_power_law(d + 1, 10)
fit2 <- fit_power_law(d + 1, 10, implementation = "R.mle")

fit1$alpha
stats4::coef(fit2)
fit1$logLik
stats4::logLik(fit2)
```

---

get_edge_ids *Find the edge ids based on the incident vertices of the edges*

Description

Find the edges in an igraph graph that have the specified end points. This function handles multi-graph (graphs with multiple edges) and can consider or ignore the edge directions in directed graphs.

Usage

```
get_edge_ids(graph, vp, directed = TRUE, error = FALSE)
```

Arguments

graph	The input graph.
vp	The incident vertices, given as a two-column data frame, two-column matrix, or vector of vertex ids or symbolic vertex names. For a vector, the values are interpreted pairwise, i.e. the first and second are used for the first edge, the third and fourth for the second, etc.
directed	Logical scalar, whether to consider edge directions in directed graphs. This argument is ignored for undirected graphs.
error	Logical scalar, whether to report an error if an edge is not found in the graph. If FALSE, then no error is reported, and zero is returned for the non-existant edge(s).

Details

igraph vertex ids are natural numbers, starting from one, up to the number of vertices in the graph. Similarly, edges are also numbered from one, up to the number of edges.

This function allows finding the edges of the graph, via their incident vertices.

Value

A numeric vector of edge ids, one for each pair of input vertices. If there is no edge in the input graph for a given pair of vertices, then zero is reported. (If the error argument is FALSE.)

Related documentation in the C library

get_eids(), ecount(), vcount(), edges()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Other structural queries: [.igraph(), [[.igraph(), adjacent_vertices(), are_adjacent(), ends(), gorder(), gsize(), head_of(), incident(), incident_edges(), is_directed(), neighbors(), tail_of()

Examples

```
g <- make_ring(10)
ei <- get_edge_ids(g, c(1, 2, 4, 5))
E(g)[ei]

## non-existant edge
get_edge_ids(g, c(2, 1, 1, 4, 5, 4))

## For multiple edges, a single edge id is returned,
## as many times as corresponding pairs in the vertex series.
g <- make_graph(rep(c(1, 2), 5))
eis <- get_edge_ids(g, c(1, 2, 1, 2))
eis
E(g)[eis]
```

---

girth *Girth of a graph*

Description

The girth of a graph is the length of the shortest circle in it.

Usage

```
girth(graph, circle = TRUE)
```

Arguments

graph	The input graph. It may be directed, but the algorithm searches for undirected circles anyway.
circle	Logical scalar, whether to return the shortest circle itself.

Details

The current implementation works for undirected graphs only, directed graphs are treated as undirected graphs. Loop edges and multiple edges are ignored. If the graph is a forest (i.e. acyclic), then `Inf` is returned.

This implementation is based on Alon Itai and Michael Rodeh: Finding a minimum circuit in a graph *Proceedings of the ninth annual ACM symposium on Theory of computing*, 1-10, 1977. The first implementation of this function was done by Keith Briggs, thanks Keith.

Value

A named list with two components:

girth Integer constant, the girth of the graph, or `Inf` if the graph is acyclic.

circle Numeric vector with the vertex ids in the shortest circle.

Related documentation in the C library

`vcount()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Alon Itai and Michael Rodeh: Finding a minimum circuit in a graph *Proceedings of the ninth annual ACM symposium on Theory of computing*, 1-10, 1977

See Also

Other structural.properties: `bfs()`, `component_distribution()`, `connect()`, `constraint()`, `coreness()`, `degree()`, `dfs()`, `distance_table()`, `edge_density()`, `feedback_arc_set()`, `feedback_vertex_set()`, `is_acyclic()`, `is_dag()`, `is_matching()`, `k_shortest_paths()`, `knn()`, `reciprocity()`, `subcomponent()`, `subgraph()`, `topo_sort()`, `transitivity()`, `unfold_tree()`, `which_multiple()`, `which_mutual()`

Graph cycles: `feedback_arc_set()`, `feedback_vertex_set()`, `find_cycle()`, `has_eulerian_path()`, `is_acyclic()`, `is_dag()`, `simple_cycles()`

Examples

```
# No circle in a tree
g <- make_tree(1000, 3)
girth(g)

# The worst case running time is for a ring
g <- make_ring(100)
girth(g)

# What about a random graph?
g <- sample_gnp(1000, 1 / 1000)
girth(g)
```

global_efficiency *Efficiency of a graph*

Description

These functions calculate the global or average local efficiency of a network, or the local efficiency of every vertex in the network. See below for definitions.

Usage

```
global_efficiency(graph, weights = NULL, directed = TRUE)

local_efficiency(
  graph,
  vids = V(graph),
  weights = NULL,
  directed = TRUE,
  mode = c("all", "out", "in", "total")
)

average_local_efficiency(
  graph,
  weights = NULL,
  directed = TRUE,
  mode = c("all", "out", "in", "total")
)
```

Arguments

graph	The graph to analyze.
weights	The edge weights. All edge weights must be non-negative; additionally, no edge weight may be NaN. If it is NULL (the default) and the graph has a weight edge attribute, then it is used automatically.
directed	Logical scalar, whether to consider directed paths. Ignored for undirected graphs.
vids	The vertex ids of the vertices for which the calculation will be done. Applies to the local efficiency calculation only.
mode	Specifies how to define the local neighborhood of a vertex in directed graphs. "out" considers out-neighbors only, "in" considers in-neighbors only, "all" considers both.

Value

For global_efficiency(), the global efficiency of the graph as a single number. For average_local_efficiency(), the average local efficiency of the graph as a single number. For local_efficiency(), the local efficiency of each vertex in a vector.

Global efficiency

The global efficiency of a network is defined as the average of inverse distances between all pairs of vertices.

More precisely:

$$E_g = \frac{1}{n(n-1)} \sum_{i \neq j} \frac{1}{d_{ij}}$$

where n is the number of vertices.

The inverse distance between pairs that are not reachable from each other is considered to be zero. For graphs with fewer than 2 vertices, NaN is returned.

Local efficiency

The local efficiency of a network around a vertex is defined as follows: We remove the vertex and compute the distances (shortest path lengths) between its neighbours through the rest of the network. The local efficiency around the removed vertex is the average of the inverse of these distances.

The inverse distance between two vertices which are not reachable from each other is considered to be zero. The local efficiency around a vertex with fewer than two neighbours is taken to be zero by convention.

Average local efficiency

The average local efficiency of a network is simply the arithmetic mean of the local efficiencies of all the vertices; see the definition for local efficiency above.

Related documentation in the C library

`global_efficiency()`, `edges()`, `get_eids()`, `vcount()`, `ecount()`, `local_efficiency()`, `average_local_efficiency`

References

V. Latora and M. Marchiori: Efficient Behavior of Small-World Networks, Phys. Rev. Lett. 87, 198701 (2001).

I. Vragović, E. Louis, and A. Díaz-Guilera, Efficiency of informational transfer in regular and complex networks, Phys. Rev. E 71, 1 (2005).

Examples

```
g <- make_graph("zachary")
global_efficiency(g)
average_local_efficiency(g)
```

gorder *Order (number of vertices) of a graph*

Description

vcount() and gorder() are aliases.

Usage

```
vcount(graph)

gorder(graph)
```

Arguments

graph	The graph

Value

Number of vertices, numeric scalar.

Related documentation in the C library

vcount()

See Also

Other structural queries: [.igraph(), [[.igraph(), adjacent_vertices(), are_adjacent(), ends(), get_edge_ids(), gsize(), head_of(), incident(), incident_edges(), is_directed(), neighbors(), tail_of()

Examples

```
g <- make_ring(10)
gorder(g)
vcount(g)
```

graph_ *Convert object to a graph*

Description

This is a generic function to convert R objects to igraph graphs.

Usage

```
graph_(...)
```

Arguments

...	Parameters, see details below.

Details

TODO

Related documentation in the C library

simplify(), vcount(), edges(), get_eids(), ecount()

Examples

```
## These are equivalent
graph_(cbind(1:5, 2:6), from_edgelist(directed = FALSE))
graph_(cbind(1:5, 2:6), from_edgelist(), directed = FALSE)
```

graph_attr *Graph attributes of a graph*

Description

Graph attributes of a graph

Usage

```
graph_attr(graph, name)
```

Arguments

graph	Input graph.
name	The name of attribute to query. If missing, then all attributes are returned in a list.

Value

A list of graph attributes, or a single graph attribute.

See Also

Vertex, edge and graph attributes: delete_edge_attr(), delete_graph_attr(), delete_vertex_attr(), edge_attr(), edge_attr<-(), edge_attr_names(), graph_attr<-(), graph_attr_names(), igraph-attribute-combination, igraph-dollar, igraph-vs-attributes, set_edge_attr(), set_graph_attr(), set_vertex_attr(), set_vertex_attrs(), vertex_attr(), vertex_attr<-(), vertex_attr_names()

Examples

```
g <- make_ring(10)
graph_attr(g)
graph_attr(g, "name")
```

---

graph_attr_names *List names of graph attributes*

Description

List names of graph attributes

Usage

```
graph_attr_names(graph)
```

Arguments

graph	The graph.

Value

Character vector, the names of the graph attributes.

See Also

Vertex, edge and graph attributes: delete_edge_attr(), delete_graph_attr(), delete_vertex_attr(), edge_attr(), edge_attr<-(), edge_attr_names(), graph_attr(), graph_attr<-(), igraph-attribute-combination, igraph-dollar, igraph-vs-attributes, set_edge_attr(), set_graph_attr(), set_vertex_attr(), set_vertex_attrs(), vertex_attr(), vertex_attr<-(), vertex_attr_names()

Examples

```
g <- make_ring(10)
graph_attr_names(g)
```

graph_attr<- *Set all or some graph attributes*

Description

Set all or some graph attributes

Usage

```
graph_attr(graph, name) <- value
```

Arguments

graph	The graph.
name	The name of the attribute to set. If missing, then value should be a named list, and all list members are set as attributes.
value	The value of the attribute to set

Value

The graph, with the attribute(s) added.

See Also

Vertex, edge and graph attributes: delete_edge_attr(), delete_graph_attr(), delete_vertex_attr(), edge_attr(), edge_attr<-(), edge_attr_names(), graph_attr(), graph_attr_names(), igraph-attribute-combina igraph-dollar, igraph-vs-attributes, set_edge_attr(), set_graph_attr(), set_vertex_attr(), set_vertex_attrs(), vertex_attr(), vertex_attr<-(), vertex_attr_names()

Examples

```
g <- make_graph(~ A - B:C:D)
graph_attr(g, "name") <- "4-star"
g

graph_attr(g) <- list(
  layout = layout_with_fr(g),
  name = "4-star layed out"
)
plot(g)
```

graph_center *Central vertices of a graph*

Description

[Experimental]

The center of a graph is the set of its vertices with minimal eccentricity.

Usage

```
graph_center(graph, ..., weights = NULL, mode = c("all", "out", "in", "total"))
```

Arguments

graph	The input graph, it can be directed or undirected.
...	These dots are for future extensions and must be empty.
weights	Possibly a numeric vector giving edge weights. If this is NULL and the graph has a weight edge attribute, then the attribute is used. If this is NA then no weights are used (even if the graph has a weight attribute). In a weighted graph, the length of a path is the sum of the weights of its constituent edges.
mode	Character constant, gives whether the shortest paths to or from the given vertices should be calculated for directed graphs. If out then the shortest paths *from* the vertex, if in then *to* it will be considered. If all, the default, then the graph is treated as undirected, i.e. edge directions are not taken into account. This argument is ignored for undirected graphs.

Value

The vertex IDs of the central vertices.

Related documentation in the C library

graph_center_dijkstra(), edges(), vcount(), get_eids(), ecount()

See Also

eccentricity(), radius()

Other paths: all_simple_paths(), diameter(), distance_table(), eccentricity(), radius()

Examples

```
tree <- make_tree(100, 7)
graph_center(tree)
graph_center(tree, mode = "in")
graph_center(tree, mode = "out")

# Without and with weights
```

```
ring <- make_ring(10)
graph_center(ring)
# Add weights
E(ring)$weight <- seq_len(ecount(ring))
graph_center(ring)
```

`graph_from_adj_list` *Create graphs from adjacency lists*

Description

An adjacency list is a list of numeric vectors, containing the neighbor vertices for each vertex. This function creates an igraph graph object from such a list.

Usage

```
graph_from_adj_list(
  adjlist,
  mode = c("out", "in", "all", "total"),
  duplicate = TRUE
)
```

Arguments

`adjlist`	The adjacency list. It should be consistent, i.e. the maximum throughout all vectors in the list must be less than the number of vectors (=the number of vertices in the graph).
`mode`	Character scalar, it specifies whether the graph to create is undirected ('all' or 'total') or directed; and in the latter case, whether it contains the outgoing ('out') or the incoming ('in') neighbors of the vertices.
`duplicate`	Logical scalar. For undirected graphs it gives whether edges are included in the list twice. E.g. if it is `TRUE` then for an undirected `{A,B}` edge `graph_from_adj_list()` expects `A` included in the neighbors of `B` and `B` to be included in the neighbors of `A`. This argument is ignored if `mode` is `out` or `in`.

Details

Adjacency lists are handy if you intend to do many (small) modifications to a graph. In this case adjacency lists are more efficient than igraph graphs.

The idea is that you convert your graph to an adjacency list by `as_adj_list()`, do your modifications to the graphs and finally create again an igraph graph by calling `graph_from_adj_list()`.

Value

An igraph graph object.

Related documentation in the C library

adjlist()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

as_edgelist()

Other conversion: as.matrix.igraph(), as_adj_list(), as_adjacency_matrix(), as_biadjacency_matrix(), as_data_frame(), as_directed(), as_edgelist(), as_graphnel(), as_long_data_frame(), graph_from_graphnel()

Examples

```
## Directed
g <- make_ring(10, directed = TRUE)
al <- as_adj_list(g, mode = "out")
g2 <- graph_from_adj_list(al)
isomorphic(g, g2)

## Undirected
g <- make_ring(10)
al <- as_adj_list(g)
g2 <- graph_from_adj_list(al, mode = "all")
isomorphic(g, g2)
ecount(g2)
g3 <- graph_from_adj_list(al, mode = "all", duplicate = FALSE)
ecount(g3)
which_multiple(g3)
```

---

graph_from_adjacency_matrix

Create graphs from adjacency matrices

Description

graph_from_adjacency_matrix() is a flexible function for creating igraph graphs from adjacency matrices.

Usage

```
graph_from_adjacency_matrix(
  adjmatrix,
  mode = c("directed", "undirected", "max", "min", "upper", "lower", "plus"),
  weighted = NULL,
  diag = TRUE,
```

```
  add.colnames = NULL,
  add.rownames = NA
)

from_adjacency(
  adjmatrix,
  mode = c("directed", "undirected", "max", "min", "upper", "lower", "plus"),
  weighted = NULL,
  diag = TRUE,
  add.colnames = NULL,
  add.rownames = NA
)
```

Arguments

adjmatrix	A square adjacency matrix. From igraph version 0.5.1 this can be a sparse matrix created with the Matrix package.
mode	Character scalar, specifies how igraph should interpret the supplied matrix. See also the weighted argument, the interpretation depends on that too. Possible values are: directed, undirected, upper, lower, max, min, plus. See details below.
weighted	This argument specifies whether to create a weighted graph from an adjacency matrix. If it is NULL then an unweighted graph is created and the elements of the adjacency matrix gives the number of edges between the vertices. If it is a character constant then for every non-zero matrix entry an edge is created and the value of the entry is added as an edge attribute named by the weighted argument. If it is TRUE then a weighted graph is created and the name of the edge attribute will be weight. See also details below.
diag	Logical scalar, whether to include the diagonal of the matrix in the calculation. If this is FALSE then the diagonal is zerod out first.
add.colnames	Character scalar, whether to add the column names as vertex attributes. If it is NULL (the default) then, if present, column names are added as vertex attribute 'name'. If NA or FALSE then they will not be added. If a character constant, then it gives the name of the vertex attribute to add.
add.rownames	Character scalar, whether to add the row names as vertex attributes. Possible values the same as the previous argument. By default row names are not added. If 'add.rownames' and 'add.colnames' specify the same vertex attribute, then the former is ignored.

Details

The order of the vertices are preserved, i.e. the vertex corresponding to the first row will be vertex 0 in the graph, etc.

graph_from_adjacency_matrix() operates in two main modes, depending on the weighted argument.

If this argument is NULL then an unweighted graph is created and an element of the adjacency matrix gives the number of edges to create between the two corresponding vertices. The details depend on the value of the mode argument:

"directed" The graph will be directed and a matrix element gives the number of edges between two vertices.

"undirected" This is exactly the same as `max`, for convenience. Note that it is *not* checked whether the matrix is symmetric.

"max" An undirected graph will be created and `max(A(i,j), A(j,i))` gives the number of edges.

"upper" An undirected graph will be created, only the upper right triangle (including the diagonal) is used for the number of edges.

"lower" An undirected graph will be created, only the lower left triangle (including the diagonal) is used for creating the edges.

"min" An undirected graph will be created with `min(A(i,j), A(j,i))` edges between vertex `i` and `j`.

"plus" An undirected graph will be created with `A(i,j)+A(j,i)` edges between vertex `i` and `j`.

If the `weighted` argument is not `NULL` then the elements of the matrix give the weights of the edges (if they are not zero). The details depend on the value of the `mode` argument:

"directed" The graph will be directed and a matrix element gives the edge weights.

"undirected" First we check that the matrix is symmetric. It is an error if not. Then only the upper triangle is used to create a weighted undirected graph.

"max" An undirected graph will be created and `max(A(i,j), A(j,i))` gives the edge weights.

"upper" An undirected graph will be created, only the upper right triangle (including the diagonal) is used (for the edge weights).

"lower" An undirected graph will be created, only the lower left triangle (including the diagonal) is used for creating the edges.

"min" An undirected graph will be created, `min(A(i,j), A(j,i))` gives the edge weights.

"plus" An undirected graph will be created, `A(i,j)+A(j,i)` gives the edge weights.

Value

An igraph graph object.

Related documentation in the C library

`adjacency()`, `weighted_adjacency()`, `create()`, `empty()`, `famous()`, `vcount()`, `edges()`, `simplify()`, `get_eids()`, `ecount()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

`make_graph()` and `graph_from_literal()` for other ways to create graphs.

Examples

```
g1 <- sample(
  x = 0:1, size = 100, replace = TRUE,
  prob = c(0.9, 0.1)
) %>%
  matrix(ncol = 10) %>%
  graph_from_adjacency_matrix()

g2 <- sample(
  x = 0:5, size = 100, replace = TRUE,
  prob = c(0.9, 0.02, 0.02, 0.02, 0.02, 0.02)
) %>%
  matrix(ncol = 10) %>%
  graph_from_adjacency_matrix(weighted = TRUE)
E(g2)$weight

## various modes for weighted graphs, with some tests
non_zero_sort <- function(x) sort(x[x != 0])
adj_matrix <- matrix(runif(100), 10)
adj_matrix[adj_matrix < 0.5] <- 0
g3 <- graph_from_adjacency_matrix(
  (adj_matrix + t(adj_matrix)) / 2,
  weighted = TRUE,
  mode = "undirected"
)

g4 <- graph_from_adjacency_matrix(
  adj_matrix,
  weighted = TRUE,
  mode = "max"
)
expected_g4_weights <- non_zero_sort(
  pmax(adj_matrix, t(adj_matrix))[upper.tri(adj_matrix, diag = TRUE)]
)
actual_g4_weights <- sort(E(g4)$weight)
all(expected_g4_weights == actual_g4_weights)

g5 <- graph_from_adjacency_matrix(
  adj_matrix,
  weighted = TRUE,
  mode = "min"
)
expected_g5_weights <- non_zero_sort(
  pmin(adj_matrix, t(adj_matrix))[upper.tri(adj_matrix, diag = TRUE)]
)
actual_g5_weights <- sort(E(g5)$weight)
all(expected_g5_weights == actual_g5_weights)

g6 <- graph_from_adjacency_matrix(
  adj_matrix,
  weighted = TRUE,
  mode = "upper"
```

```
)
expected_g6_weights <- non_zero_sort(adj_matrix[upper.tri(adj_matrix, diag = TRUE)])
actual_g6_weights <- sort(E(g6)$weight)
all(expected_g6_weights == actual_g6_weights)

g7 <- graph_from_adjacency_matrix(
  adj_matrix,
  weighted = TRUE,
  mode = "lower"
)
expected_g7_weights <- non_zero_sort(adj_matrix[lower.tri(adj_matrix, diag = TRUE)])
actual_g7_weights <- sort(E(g7)$weight)
all(expected_g7_weights == actual_g7_weights)

g8 <- graph_from_adjacency_matrix(
  adj_matrix,
  weighted = TRUE,
  mode = "plus"
)
halve_diag <- function(x) {
  diag(x) <- diag(x) / 2
  x
}
expected_g8_weights <- non_zero_sort(
  halve_diag(adj_matrix + t(adj_matrix))[lower.tri(adj_matrix, diag = TRUE)]
)
actual_g8_weights <- sort(E(g8)$weight)
all(expected_g8_weights == actual_g8_weights)

g9 <- graph_from_adjacency_matrix(
  adj_matrix,
  weighted = TRUE,
  mode = "plus",
  diag = FALSE
)
zero_diag <- function(x) {
  diag(x) <- 0
}
expected_g9_weights <- non_zero_sort((zero_diag(adj_matrix + t(adj_matrix)))[lower.tri(adj_matrix)])
actual_g9_weights <- sort(E(g9)$weight)
all(expected_g9_weights == actual_g9_weights)

## row/column names
rownames(adj_matrix) <- sample(letters, nrow(adj_matrix))
colnames(adj_matrix) <- seq(ncol(adj_matrix))
g10 <- graph_from_adjacency_matrix(
  adj_matrix,
  weighted = TRUE,
  add.rownames = "code"
)
summary(g10)
```

graph_from_atlas *Create a graph from the Graph Atlas*

Description

graph_from_atlas() creates graphs from the book 'An Atlas of Graphs' by Roland C. Read and Robin J. Wilson. The atlas contains all undirected graphs with up to seven vertices, numbered from 0 up to 1252. The graphs are listed:

1. in increasing order of number of nodes;
2. for a fixed number of nodes, in increasing order of the number of edges;
3. for fixed numbers of nodes and edges, in increasing order of the degree sequence, for example 111223 < 112222;
4. for fixed degree sequence, in increasing number of automorphisms.

Usage

```
graph_from_atlas(n)

atlas(n)
```

Arguments

n	The id of the graph to create.

Value

An igraph graph.

Related documentation in the C library

atlas()

See Also

Other deterministic constructors: graph_from_edgelist(), graph_from_literal(), make_(), make_chordal_ring(), make_circulant(), make_empty_graph(), make_full_citation_graph(), make_full_graph(), make_full_multipartite(), make_graph(), make_lattice(), make_ring(), make_star(), make_tree(), make_turan(), make_wheel()

Examples

```
## Some randomly picked graphs from the atlas
graph_from_atlas(sample(0:1252, 1))
graph_from_atlas(sample(0:1252, 1))
```

graph_from_biadjacency_matrix *Create graphs from a bipartite adjacency matrix*

Description

`graph_from_biadjacency_matrix()` creates a bipartite igraph graph from an incidence matrix.

Usage

```
graph_from_biadjacency_matrix(
  incidence,
  directed = FALSE,
  mode = c("all", "out", "in", "total"),
  multiple = FALSE,
  weighted = NULL,
  add.names = NULL
)
```

Arguments

`incidence`	The input bipartite adjacency matrix. It can also be a sparse matrix from the `Matrix` package.
`directed`	Logical scalar, whether to create a directed graph.
`mode`	A character constant, defines the direction of the edges in directed graphs, ignored for undirected graphs. If '`out`', then edges go from vertices of the first kind (corresponding to rows in the bipartite adjacency matrix) to vertices of the second kind (columns in the incidence matrix). If '`in`', then the opposite direction is used. If '`all`' or '`total`', then mutual edges are created.
`multiple`	Logical scalar, specifies how to interpret the matrix elements. See details below.
`weighted`	This argument specifies whether to create a weighted graph from the bipartite adjacency matrix. If it is `NULL` then an unweighted graph is created and the `multiple` argument is used to determine the edges of the graph. If it is a character constant then for every non-zero matrix entry an edge is created and the value of the entry is added as an edge attribute named by the `weighted` argument. If it is `TRUE` then a weighted graph is created and the name of the edge attribute will be '`weight`'.
`add.names`	A character constant, `NA` or `NULL`. `graph_from_biadjacency_matrix()` can add the row and column names of the incidence matrix as vertex attributes. If this argument is `NULL` (the default) and the bipartite adjacency matrix has both row and column names, then these are added as the '`name`' vertex attribute. If you want a different vertex attribute for this, then give the name of the attributes as a character string. If this argument is `NA`, then no vertex attributes (other than type) will be added.

Details

Bipartite graphs have a 'type' vertex attribute in igraph, this is boolean and FALSE for the vertices of the first kind and TRUE for vertices of the second kind.

graph_from_biadjacency_matrix() can operate in two modes, depending on the multiple argument. If it is FALSE then a single edge is created for every non-zero element in the bipartite adjacency matrix. If multiple is TRUE, then the matrix elements are rounded up to the closest non-negative integer to get the number of edges to create between a pair of vertices.

Some authors refer to the bipartite adjacency matrix as the "bipartite incidence matrix". igraph 1.6.0 and later does not use this naming to avoid confusion with the edge-vertex incidence matrix.

Value

A bipartite igraph graph. In other words, an igraph graph that has a vertex attribute type.

Related documentation in the C library

biadjacency(), create(), empty(), vcount(), famous(), simplify(), edges(), get_eids(), ecount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

make_bipartite_graph() for another way to create bipartite graphs

Other biadjacency: as_data_frame()

Examples

```
inc <- matrix(sample(0:1, 15, repl = TRUE), 3, 5)
colnames(inc) <- letters[1:5]
rownames(inc) <- LETTERS[1:3]
graph_from_biadjacency_matrix(inc)
```

---

graph_from_edgelist *Create a graph from an edge list matrix*

Description

graph_from_edgelist() creates a graph from an edge list. Its argument is a two-column matrix, each row defines one edge. If it is a numeric matrix then its elements are interpreted as vertex ids. If it is a character matrix then it is interpreted as symbolic vertex names and a vertex id will be assigned to each name, and also a name vertex attribute will be added.

Usage

```
graph_from_edgelist(el, directed = TRUE)

from_edgelist(...)
```

Arguments

el	The edge list, a two column matrix, character or numeric.
directed	Whether to create a directed graph.
...	Passed to graph_from_edgelist().

Value

An igraph graph.

Related documentation in the C library

create(), empty(), vcount(), famous(), simplify()

See Also

Other deterministic constructors: graph_from_atlas(), graph_from_literal(), make_(), make_chordal_ring(), make_circulant(), make_empty_graph(), make_full_citation_graph(), make_full_graph(), make_full_multipartite(), make_graph(), make_lattice(), make_ring(), make_star(), make_tree(), make_turan(), make_wheel()

Examples

```
el <- matrix(c("foo", "bar", "bar", "foobar"), nc = 2, byrow = TRUE)
graph_from_edgelist(el)

# Create a ring by hand
graph_from_edgelist(cbind(1:10, c(2:10, 1)))
```

---

graph_from_graphdb *Load a graph from the graph database for testing graph isomorphism.*

Description

This function downloads a graph from a database created for the evaluation of graph isomorphism testing algorithms.

Usage

```
graph_from_graphdb(
  url = NULL,
  prefix = "iso",
  type = "r001",
  nodes = NULL,
  pair = "A",
  which = 0,
  base = "https://github.com/igraph/graphsdb/raw/refs/heads/main",
  compressed = TRUE,
  directed = TRUE
)
```

Arguments

url	If not NULL it is a complete URL with the file to import.
prefix	Gives the prefix. See details below. Possible values: iso, i2, si4, si6, mcs10, mcs30, mcs50, mcs70, mcs90.
type	Gives the graph type identifier. See details below. Possible values: r001, r005, r01, r02, m2D, m2Dr2, m2Dr4, m2Dr6 m3D, m3Dr2, m3Dr4, m3Dr6, m4D, m4Dr2, m4Dr4, m4Dr6, b03, b03m, b06, b06m, b09, b09m.
nodes	The number of vertices in the graph.
pair	Specifies which graph of the pair to read. Possible values: A and B.
which	Gives the number of the graph to read. For every graph type there are a number of actual graphs in the database. This argument specifies which one to read.
base	The base address of the database. See details below.
compressed	Logical constant, if TRUE than the file is expected to be compressed by gzip. If url is NULL then a '.gz' suffix is added to the filename.
directed	Logical constant, whether to create a directed graph.

Details

graph_from_graphdb() reads a graph from the graph database from an FTP or HTTP server or from a local copy. It has two modes of operation:

If the url argument is specified then it should the complete path to a local or remote graph database file. In this case we simply call read_graph() with the proper arguments to read the file.

If url is NULL, and this is the default, then the filename is assembled from the base, prefix, type, nodes, pair and which arguments.

Unfortunately the original graph database homepage is now defunct, but see its old version at https://web.archive.org/web/20090215182331/http://amalfi.dis.unina.it/graph/db/doc/graphdbat.html for the actual format of a graph database file and other information.

Value

A new graph object.

Related documentation in the C library

read_graph_graphdb()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

M. De Santo, P. Foggia, C. Sansone, M. Vento: A large database of graphs and its use for benchmarking graph isomorphism algorithms, *Pattern Recognition Letters*, Volume 24, Issue 8 (May 2003)

See Also

read_graph(), isomorphic()

Foreign format readers: read_graph(), write_graph()

---

graph_from_graphnel *Convert graphNEL objects from the graph package to igraph*

Description

The graphNEL class is defined in the graph package, it is another way to represent graphs. graph_from_graphnel() takes a graphNEL graph and converts it to an igraph graph. It handles all graph/vertex/edge attributes. If the graphNEL graph has a vertex attribute called 'name' it will be used as igraph vertex attribute 'name' and the graphNEL vertex names will be ignored.

Usage

```
graph_from_graphnel(graphNEL, name = TRUE, weight = TRUE, unlist.attrs = TRUE)
```

Arguments

graphNEL	The graphNEL graph.
name	Logical scalar, whether to add graphNEL vertex names as an igraph vertex attribute called 'name'.
weight	Logical scalar, whether to add graphNEL edge weights as an igraph edge attribute called 'weight'. (graphNEL graphs are always weighted.)
unlist.attrs	Logical scalar. graphNEL attribute query functions return the values of the attributes in R lists, if this argument is TRUE (the default) these will be converted to atomic vectors, whenever possible, before adding them to the igraph graph.

Details

Because graphNEL graphs poorly support multiple edges, the edge attributes of the multiple edges are lost: they are all replaced by the attributes of the first of the multiple edges.

Value

graph_from_graphnel() returns an igraph graph object.

Related documentation in the C library

get_edgelist(), adjlist(), vcount(), edges(), get_eids(), ecount()

See Also

as_graphnel() for the other direction, as_adjacency_matrix(), graph_from_adjacency_matrix(), as_adj_list() and graph_from_adj_list() for other graph representations.

Other conversion: as.matrix.igraph(), as_adj_list(), as_adjacency_matrix(), as_biadjacency_matrix(), as_data_frame(), as_directed(), as_edgelist(), as_graphnel(), as_long_data_frame(), graph_from_adj_list()

Examples

```
## Undirected
g <- make_ring(10)
V(g)$name <- letters[1:10]
GNEL <- as_graphnel(g)
g2 <- graph_from_graphnel(GNEL)
g2

## Directed
g3 <- make_star(10, mode = "in")
V(g3)$name <- letters[1:10]
GNEL2 <- as_graphnel(g3)
g4 <- graph_from_graphnel(GNEL2)
g4
```

---

graph_from_isomorphism_class

Create a graph from an isomorphism class

Description

The isomorphism class is a non-negative integer number. Graphs (with the same number of vertices) having the same isomorphism class are isomorphic and isomorphic graphs always have the same isomorphism class. Currently it can handle directed graphs with 3 or 4 vertices and undirected graphd with 3 to 6 vertices.

Usage

```
graph_from_isomorphism_class(size, number, directed = TRUE)
```

Arguments

size	The number of vertices in the graph.
number	The isomorphism class.
directed	Whether to create a directed graph (the default).

Value

An igraph object, the graph of the given size, directedness and isomorphism class.

Related documentation in the C library

isoclass_create()

See Also

Other graph isomorphism: canonical_permutation(), count_isomorphisms(), count_subgraph_isomorphisms(), isomorphic(), isomorphism_class(), isomorphisms(), subgraph_isomorphic(), subgraph_isomorphisms()

---

graph_from_lcf *Creating a graph from LCF notation*

Description

LCF is short for Lederberg-Coxeter-Frucht, it is a concise notation for 3-regular Hamiltonian graphs. It constists of three parameters, the number of vertices in the graph, a list of shifts giving additional edges to a cycle backbone and another integer giving how many times the shifts should be performed. See https://mathworld.wolfram.com/LCFNotation.html for details.

Usage

```
graph_from_lcf(shifts, ..., n = NULL, repeats = 1L)
```

Arguments

shifts	Integer vector, the shifts.
...	These dots are for future extensions and must be empty.
n	Integer, the number of vertices in the graph. If NULL (default), it is set to len(shifts) * repeats.
repeats	Integer constant, how many times to repeat the shifts.

Value

A graph object.

Related documentation in the C library

lcf_vector()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

make_graph() can create arbitrary graphs, see also the other functions on the its manual page for creating special graphs.

Examples

```
# This is the Franklin graph:
g1 <- graph_from_lcf(shifts = c(5L, -5L), n = 12L, repeats = 6L)
g2 <- make_graph("Franklin")
isomorphic(g1, g2)
```

---

graph_from_literal *Creating (small) graphs via a simple interface*

Description

This function is useful if you want to create a small (named) graph quickly, it works for both directed and undirected graphs.

Usage

```
graph_from_literal(..., simplify = TRUE)

from_literal(...)
```

Arguments

...	For graph_from_literal() the formulae giving the structure of the graph, see details below. For from_literal() all arguments are passed to graph_from_literal().
simplify	Logical scalar, whether to call simplify() on the created graph. By default the graph is simplified, loop and multiple edges are removed.

Details

graph_from_literal() is very handy for creating small graphs quickly. You need to supply one or more R expressions giving the structure of the graph. The expressions consist of vertex names and edge operators. An edge operator is a sequence of '-' and '+' characters, the former is for the edges and the latter is used for arrow heads. The edges can be arbitrarily long, i.e. you may use as many '-' characters to "draw" them as you like.

If all edge operators consist of only '-' characters then the graph will be undirected, whereas a single '+' character implies a directed graph.

Let us see some simple examples. Without arguments the function creates an empty graph:

```
graph_from_literal()
```

A simple undirected graph with two vertices called 'A' and 'B' and one edge only:

```
graph_from_literal(A-B)
```

Remember that the length of the edges does not matter, so we could have written the following, this creates the same graph:

```
graph_from_literal( A-----B )
```

If you have many disconnected components in the graph, separate them with commas. You can also give isolate vertices.

```
graph_from_literal( A--B, C--D, E--F, G--H, I, J, K )
```

The ':' operator can be used to define vertex sets. If an edge operator connects two vertex sets then every vertex from the first set will be connected to every vertex in the second set. The following form creates a full graph, including loop edges:

```
graph_from_literal( A:B:C:D -- A:B:C:D )
```

In directed graphs, edges will be created only if the edge operator includes a arrow head ('+') *at the end* of the edge:

```
graph_from_literal( A -+ B -+ C )
graph_from_literal( A +- B -+ C )
graph_from_literal( A +- B -- C )
```

Thus in the third example no edge is created between vertices `B` and `C`.

Mutual edges can be also created with a simple edge operator:

```
graph_from_literal( A +-+ B +---+ C ++ D + E)
```

Note again that the length of the edge operators is arbitrary, '+', '++' and '+------+' have exactly the same meaning.

If the vertex names include spaces or other special characters then you need to quote them:

```
graph_from_literal( "this is" +- "a silly" -+ "graph here" )
```

You can include any character in the vertex names this way, even '+' and '-' characters.

See more examples below.

Value

An igraph graph

Related documentation in the C library

`create()`, `simplify()`, `famous()`, `vcount()`, `empty()`

See Also

Other deterministic constructors: graph_from_atlas(), graph_from_edgelist(), make_(), make_chordal_ring(), make_circulant(), make_empty_graph(), make_full_citation_graph(), make_full_graph(), make_full_multipartite(), make_graph(), make_lattice(), make_ring(), make_star(), make_tree(), make_turan(), make_wheel()

Examples

```
# A simple undirected graph
g <- graph_from_literal(
  Alice - Bob - Cecil - Alice,
  Daniel - Cecil - Eugene,
  Cecil - Gordon
)
g

# Another undirected graph, ":" notation
g2 <- graph_from_literal(Alice - Bob:Cecil:Daniel, Cecil:Daniel - Eugene:Gordon)
g2

# A directed graph
g3 <- graph_from_literal(
  Alice +-+ Bob --+ Cecil +-- Daniel,
  Eugene --+ Gordon:Helen
)
g3

# A graph with isolate vertices
g4 <- graph_from_literal(Alice -- Bob -- Daniel, Cecil:Gordon, Helen)
g4
V(g4)$name

# "Arrows" can be arbitrarily long
g5 <- graph_from_literal(Alice +---------+ Bob)
g5

# Special vertex names
g6 <- graph_from_literal("+" -- "-", "*" -- "/", "%%" -- "%/%")
g6
```

---

graph_id *Get the id of a graph*

Description

Graph ids are used to check that a vertex or edge sequence belongs to a graph. If you create a new graph by changing the structure of a graph, the new graph will have a new id. Changing the attributes will not change the id.

Usage

```
graph_id(x, ...)
```

Arguments

x	A graph or a vertex sequence or an edge sequence.
...	Not used currently.

Value

The id of the graph, a character scalar. For vertex and edge sequences the id of the graph they were created from.

Examples

```
g <- make_ring(10)
graph_id(g)
graph_id(V(g))
graph_id(E(g))

g2 <- g + 1
graph_id(g2)
```

---

graph_version *igraph data structure versions*

Description

igraph's internal data representation changes sometimes between versions. This means that it is not always possible to use igraph objects that were created (and possibly saved to a file) with an older igraph version.

Usage

```
graph_version(graph)
```

Arguments

graph	The input graph. If it is missing, then the version number of the current data format is returned.

Details

`graph_version()` queries the current data format, or the data format of a possibly older igraph graph.

`upgrade_graph()` can convert an older data format to the current one.

Value

An integer scalar.

See Also

upgrade_graph to convert the data format of a graph.

Other versions: `upgrade_graph()`

`graphlet_basis` *Graphlet decomposition of a graph*

Description

Graphlet decomposition models a weighted undirected graph via the union of potentially overlapping dense social groups. This is done by a two-step algorithm. In the first step a candidate set of groups (a candidate basis) is created by finding cliques if the thresholded input graph. In the second step these the graph is projected on the candidate basis, resulting a weight coefficient for each clique in the candidate basis.

Usage

```
graphlet_basis(graph, weights = NULL)

graphlet_proj(
  graph,
  weights = NULL,
  cliques,
  niter = 1000,
  Mu = rep(1, length(cliques))
)

graphlets(graph, weights = NULL, niter = 1000)
```

Arguments

`graph`	The input graph, edge directions are ignored. Only simple graph (i.e. graphs without self-loops and multiple edges) are supported.
`weights`	Edge weights. If the graph has a `weight` edge attribute and this argument is `NULL` (the default), then the `weight` edge attribute is used.
`cliques`	A list of vertex ids, the graphlet basis to use for the projection.
`niter`	Integer scalar, the number of iterations to perform.
`Mu`	Starting weights for the projection.

Details

igraph contains three functions for performing the graph decomponsition of a graph. The first is `graphlets()`, which performed both steps on the method and returns a list of subgraphs, with their corresponding weights. The second and third functions correspond to the first and second steps of the algorithm, and they are useful if the user wishes to perform them individually: `graphlet_basis()` and `graphlet_proj()`.

Value

`graphlets()` returns a list with two members:

cliques A list of subgraphs, the candidate graphlet basis. Each subgraph is give by a vector of vertex ids.

Mu The weights of the subgraphs in graphlet basis.

`graphlet_basis()` returns a list of two elements:

cliques A list of subgraphs, the candidate graphlet basis. Each subgraph is give by a vector of vertex ids.

thresholds The weight thresholds used for finding the subgraphs.

`graphlet_proj()` return a numeric vector, the weights of the graphlet basis subgraphs.

Related documentation in the C library

`graphlets_candidate_basis()`, `vcount()`, `edges()`, `get_eids()`, `ecount()`, `graphlets_project()`, `graphlets()`

Examples

```
## Create an example graph first
D1 <- matrix(0, 5, 5)
D2 <- matrix(0, 5, 5)
D3 <- matrix(0, 5, 5)
D1[1:3, 1:3] <- 2
D2[3:5, 3:5] <- 3
D3[2:5, 2:5] <- 1

g <- simplify(graph_from_adjacency_matrix(D1 + D2 + D3,
  mode = "undirected", weighted = TRUE
))
V(g)$color <- "white"
E(g)$label <- E(g)$weight
E(g)$label.cex <- 2
E(g)$color <- "black"
layout(matrix(1:6, nrow = 2, byrow = TRUE))
co <- layout_with_kk(g)
par(mar = c(1, 1, 1, 1))
plot(g, layout = co)

## Calculate graphlets
```

```
gl <- graphlets(g, niter = 1000)

## Plot graphlets
for (i in 1:length(gl$cliques)) {
  sel <- gl$cliques[[i]]
  V(g)$color <- "white"
  V(g)[sel]$color <- "#E495A5"
  E(g)$width <- 1
  E(g)[V(g)[sel] %--% V(g)[sel]]$width <- 2
  E(g)$label <- ""
  E(g)[width == 2]$label <- round(gl$Mu[i], 2)
  E(g)$color <- "black"
  E(g)[width == 2]$color <- "#E495A5"
  plot(g, layout = co)
}
```

`greedy_vertex_coloring` *Greedy vertex coloring*

Description

`greedy_vertex_coloring()` finds a coloring for the vertices of a graph based on a simple greedy algorithm.

Usage

```
greedy_vertex_coloring(graph, heuristic = c("colored_neighbors", "dsatur"))
```

Arguments

`graph`	The graph object to color.
`heuristic`	The selection heuristic for the next vertex to consider. Possible values are: "colored_neighbors" selects the vertex with the largest number of already colored neighbors. "dsatur" selects the vertex with the largest number of unique colors in its neighborhood, i.e. its "saturation degree"; when there are several maximum saturation degree vertices, the one with the most uncolored neighbors will be selected.

Details

The goal of vertex coloring is to assign a "color" (represented as a positive integer) to each vertex of the graph such that neighboring vertices never have the same color. This function solves the problem by considering the vertices one by one according to a heuristic, always choosing the smallest color that differs from that of already colored neighbors. The coloring obtained this way is not necessarily minimum but it can be calculated in linear time.

Value

A numeric vector where item `i` contains the color index associated to vertex `i`.

Related documentation in the C library

`vertex_coloring_greedy()`, `vcount()`

Examples

```
g <- make_graph("petersen")
col <- greedy_vertex_coloring(g)
plot(g, vertex.color = col)
```

---

groups *Groups of a vertex partitioning*

Description

Create a list of vertex groups from some graph clustering or community structure.

Usage

```
groups(x)
```

Arguments

`x`	Some object that represents a grouping of the vertices. See details below.

Details

Currently two methods are defined for this function. The default method works on the output of `components()`. (In fact it works on any object that is a list with an entry called `membership`.)

The second method works on `communities()` objects.

Value

A named list of numeric or character vectors. The names are just numbers that refer to the groups. The vectors themselves are numeric or symbolic vertex ids.

See Also

`components()` and the various community finding functions.

Community detection: `as_membership()`, `cluster_edge_betweenness()`, `cluster_fast_greedy()`, `cluster_fluid_communities()`, `cluster_infomap()`, `cluster_label_prop()`, `cluster_leading_eigen()`, `cluster_leiden()`, `cluster_louvain()`, `cluster_optimal()`, `cluster_spinglass()`, `cluster_walktrap()`, `compare()`, `make_clusters()`, `membership()`, `modularity.igraph()`, `plot_dendrogram()`, `split_join_distance()`, `voronoi_cells()`

Examples

```
g <- make_graph("Zachary")
fgc <- cluster_fast_greedy(g)
groups(fgc)

g2 <- make_ring(10) + make_full_graph(5)
groups(components(g2))
```

---

gsize *The size of the graph (number of edges)*

Description

ecount() and gsize() are aliases.

Usage

```
gsize(graph)

ecount(graph)
```

Arguments

graph	The graph.

Value

Numeric scalar, the number of edges.

Related documentation in the C library

ecount()

See Also

Other structural queries: [.igraph(), [[.igraph(), adjacent_vertices(), are_adjacent(), ends(), get_edge_ids(), gorder(), head_of(), incident(), incident_edges(), is_directed(), neighbors(), tail_of()

Examples

```
g <- sample_gnp(100, 2 / 100)
gsize(g)
ecount(g)

# Number of edges in a G(n,p) graph
replicate(100, sample_gnp(10, 1 / 2), simplify = FALSE) %>%
  vapply(gsize, 0) %>%
  hist()
```

`harmonic_centrality` *Harmonic centrality of vertices*

Description

The harmonic centrality of a vertex is the mean inverse distance to all other vertices. The inverse distance to an unreachable vertex is considered to be zero.

Usage

```
harmonic_centrality(
  graph,
  vids = V(graph),
  mode = c("out", "in", "all", "total"),
  weights = NULL,
  normalized = FALSE,
  cutoff = -1
)
```

Arguments

`graph`	The graph to analyze.
`vids`	The vertices for which harmonic centrality will be calculated.
`mode`	Character string, defining the types of the paths used for measuring the distance in directed graphs. "out" follows paths along the edge directions only, "in" traverses the edges in reverse, while "all" ignores edge directions. This argument is ignored for undirected graphs.
`weights`	Optional positive weight vector for calculating weighted harmonic centrality. If the graph has a `weight` edge attribute, then this is used by default. Weights are used for calculating weighted shortest paths, so they are interpreted as distances.
`normalized`	Logical scalar, whether to calculate the normalized harmonic centrality. If true, the result is the mean inverse path length to other vertices, i.e. it is normalized by the number of vertices minus one. If false, the result is the sum of inverse path lengths to other vertices.
`cutoff`	The maximum path length to consider when calculating the harmonic centrality. There is no such limit when the cutoff is negative. Note that zero cutoff means that only paths of at most length 0 are considered.

Details

The `cutoff` argument can be used to restrict the calculation to paths of length `cutoff` or smaller only; this can be used for larger graphs to speed up the calculation. If `cutoff` is negative (which is the default), then the function calculates the exact harmonic centrality scores.

Value

Numeric vector with the harmonic centrality scores of all the vertices in `v`.

Related documentation in the C library

harmonic_centrality_cutoff(), vcount(), edges(), get_eids(), ecount()

References

M. Marchiori and V. Latora, Harmony in the small-world, *Physica A* 285, pp. 539-546 (2000).

See Also

betweenness(), closeness()

Centrality measures: alpha_centrality(), authority_score(), betweenness(), closeness(), diversity(), eigen_centrality(), hits_scores(), page_rank(), power_centrality(), spectrum(), strength(), subgraph_centrality()

Examples

```
g <- make_ring(10)
g2 <- make_star(10)
harmonic_centrality(g)
harmonic_centrality(g2, mode = "in")
harmonic_centrality(g2, mode = "out")
harmonic_centrality(g %du% make_full_graph(5), mode = "all")
```

---

has_eulerian_path *Find Eulerian paths or cycles in a graph*

Description

has_eulerian_path() and has_eulerian_cycle() checks whether there is an Eulerian path or cycle in the input graph. eulerian_path() and eulerian_cycle() return such a path or cycle if it exists, and throws an error otherwise.

Usage

```
has_eulerian_path(graph)

has_eulerian_cycle(graph)

eulerian_path(graph)

eulerian_cycle(graph)
```

Arguments

graph	An igraph graph object

Details

has_eulerian_path() decides whether the input graph has an Eulerian *path*, i.e. a path that passes through every edge of the graph exactly once, and returns a logical value as a result. eulerian_path() returns a possible Eulerian path, described with its edge and vertex sequence, or throws an error if no such path exists.

has_eulerian_cycle() decides whether the input graph has an Eulerian *cycle*, i.e. a path that passes through every edge of the graph exactly once and that returns to its starting point, and returns a logical value as a result. eulerian_cycle() returns a possible Eulerian cycle, described with its edge and vertex sequence, or throws an error if no such cycle exists.

Value

For has_eulerian_path() and has_eulerian_cycle(), a logical value that indicates whether the graph contains an Eulerian path or cycle. For eulerian_path() and eulerian_cycle(), a named list with two entries:

epath A vector containing the edge ids along the Eulerian path or cycle.

vpath A vector containing the vertex ids along the Eulerian path or cycle.

Related documentation in the C library

is_eulerian(), eulerian_path(), ecount(), vcount(), edges(), get_eids(), eulerian_cycle()

See Also

Graph cycles: feedback_arc_set(), feedback_vertex_set(), find_cycle(), girth(), is_acyclic(), is_dag(), simple_cycles()

Examples

```
g <- make_graph(~ A - B - C - D - E - A - F - D - B - F - E)

has_eulerian_path(g)
eulerian_path(g)

has_eulerian_cycle(g)
try(eulerian_cycle(g))
```

---

head_of *Head of the edge(s) in a graph*

Description

For undirected graphs, head and tail is not defined. In this case head_of() returns vertices incident to the supplied edges, and tail_of() returns the other end(s) of the edge(s).

Usage

```
head_of(graph, es)
```

Arguments

graph	The input graph.
es	The edges to query.

Value

A vertex sequence with the head(s) of the edge(s).

Related documentation in the C library

edges(), vcount(), get_eids(), ecount()

See Also

Other structural queries: [.igraph(), [[.igraph(), adjacent_vertices(), are_adjacent(), ends(), get_edge_ids(), gorder(), gsize(), incident(), incident_edges(), is_directed(), neighbors(), tail_of()

---

head_print *Print the only the head of an R object*

Description

Print the only the head of an R object

Usage

```
head_print(
  x,
  max_lines = 20,
  header = "",
  footer = "",
  omitted_footer = "",
  ...
)
```

Arguments

x	The object to print, or a callback function. See printer_callback() for details.
max_lines	Maximum number of lines to print, *not* including the header and the footer.
header	The header, if a function, then it will be called, otherwise printed using cat.
footer	The footer, if a function, then it will be called, otherwise printed using cat.

omitted_footer	Footer that is only printed if anything is omitted from the printout. If a function, then it will be called, otherwise printed using cat.
...	Extra arguments to pass to print().

Value

x, invisibly.

---

hits_scores *Kleinberg's hub and authority centrality scores.*

Description

The hub scores of the vertices are defined as the principal eigenvector of AA^T, where A is the adjacency matrix of the graph.

Usage

```
hits_scores(
  graph,
  ...,
  scale = TRUE,
  weights = NULL,
  options = arpack_defaults()
)
```

Arguments

graph	The input graph.
...	These dots are for future extensions and must be empty.
scale	Logical scalar, whether to scale the result to have a maximum score of one. If no scaling is used then the result vector has unit length in the Euclidean norm.
weights	Optional positive weight vector for calculating weighted scores. If the graph has a weight edge attribute, then this is used by default. Pass NA to ignore the weight attribute. This function interprets edge weights as connection strengths. The weights of parallel edges are effectively added up.
options	A named list, to override some ARPACK options. See arpack() for details.

Details

Similarly, the authority scores of the vertices are defined as the principal eigenvector of $A^T A$, where A is the adjacency matrix of the graph.

For undirected matrices the adjacency matrix is symmetric and the hub scores are the same as authority scores.

Value

A named list with members:

hub The hub score of the vertices.

authority The authority score of the vertices.

value The corresponding eigenvalue of the calculated principal eigenvector.

options Some information about the ARPACK computation, it has the same members as the `options` member returned by `arpack()`, see that for documentation.

Related documentation in the C library

`hub_and_authority_scores()`, `vcount()`, `edges()`, `get_eids()`, `ecount()`

References

J. Kleinberg. Authoritative sources in a hyperlinked environment. *Proc. 9th ACM-SIAM Symposium on Discrete Algorithms*, 1998. Extended version in *Journal of the ACM* 46(1999). Also appears as IBM Research Report RJ 10076, May 1997.

See Also

`eigen_centrality()` for eigenvector centrality, `page_rank()` for the Page Rank scores. `arpack()` for the underlining machinery of the computation.

Centrality measures: `alpha_centrality()`, `authority_score()`, `betweenness()`, `closeness()`, `diversity()`, `eigen_centrality()`, `harmonic_centrality()`, `page_rank()`, `power_centrality()`, `spectrum()`, `strength()`, `subgraph_centrality()`

Examples

```
## An in-star
g <- make_star(10)
hits_scores(g)

## A ring
g2 <- make_ring(10)
hits_scores(g2)
```

---

hrg *Create a hierarchical random graph from an igraph graph*

Description

`hrg()` creates a HRG from an igraph graph. The igraph graph must be a directed binary tree, with $n-1$ internal and n leaf vertices. The `prob` argument contains the HRG probability labels for each vertex; these are ignored for leaf vertices.

Usage

```
hrg(graph, prob)
```

Arguments

graph	The igraph graph to create the HRG from.
prob	A vector of probabilities, one for each vertex, in the order of vertex ids.

Value

`hrg()` returns an `igraphHRG` object.

Related documentation in the C library

`hrg_create()`

See Also

Other hierarchical random graph functions: `consensus_tree()`, `fit_hrg()`, `hrg-methods`, `hrg_tree()`, `predict_edges()`, `print.igraphHRG()`, `print.igraphHRGConsensus()`, `sample_hrg()`

---

hrg-methods *Hierarchical random graphs*

Description

Fitting and sampling hierarchical random graph models.

Details

A hierarchical random graph is an ensemble of undirected graphs with n vertices. It is defined via a binary tree with n leaf and $n - 1$ internal vertices, where the internal vertices are labeled with probabilities. The probability that two vertices are connected in the random graph is given by the probability label at their closest common ancestor.

Please see references below for more about hierarchical random graphs.

igraph contains functions for fitting HRG models to a given network (`fit_hrg()`, for generating networks from a given HRG ensemble (`sample_hrg()`), converting an igraph graph to a HRG and back (`hrg()`, `hrg_tree()`), for calculating a consensus tree from a set of sampled HRGs (`consensus_tree()`) and for predicting missing edges in a network based on its HRG models (`predict_edges()`).

The igraph HRG implementation is heavily based on the code published by Aaron Clauset, at his website (not functional any more).

See Also

Other hierarchical random graph functions: `consensus_tree()`, `fit_hrg()`, `hrg()`, `hrg_tree()`, `predict_edges()`, `print.igraphHRG()`, `print.igraphHRGConsensus()`, `sample_hrg()`

hrg_tree *Create an igraph graph from a hierarchical random graph model*

Description

`hrg_tree()` creates the corresponsing igraph tree of a hierarchical random graph model.

Usage

```
hrg_tree(hrg)
```

Arguments

`hrg`	A hierarchical random graph model.

Value

An igraph graph with a vertex attribute called `"probability"`.

Related documentation in the C library

`from_hrg_dendrogram()`, `vcount()`

See Also

Other hierarchical random graph functions: `consensus_tree()`, `fit_hrg()`, `hrg()`, `hrg-methods`, `predict_edges()`, `print.igraphHRG()`, `print.igraphHRGConsensus()`, `sample_hrg()`

identical_graphs *Decide if two graphs are identical*

Description

Two graphs are considered identical by this function if and only if they are represented in exactly the same way in the internal R representation. This means that the two graphs must have the same list of vertices and edges, in exactly the same order, with same directedness, and the two graphs must also have identical graph, vertex and edge attributes.

Usage

```
identical_graphs(g1, g2, attrs = TRUE)
```

Arguments

`g1`, `g2`	The two graphs
`attrs`	Whether to compare the attributes of the graphs

Details

This is similar to `identical` in the `base` package, but it ignores the mutable piece of igraph objects; those might be different even if the two graphs are identical.

Attribute comparison can be turned off with the `attrs` parameter if the attributes of the two graphs are allowed to be different.

Value

Logical scalar

`igraph-attribute-combination`

How igraph functions handle attributes when the graph changes

---

Description

Many times, when the structure of a graph is modified, vertices/edges map of the original graph map to vertices/edges in the newly created (modified) graph. For example `simplify()` maps multiple edges to single edges. igraph provides a flexible mechanism to specify what to do with the vertex/edge attributes in these cases.

Details

The functions that support the combination of attributes have one or two extra arguments called `vertex.attr.comb` and/or `edge.attr.comb` that specify how to perform the mapping of the attributes. E.g. `contract()` contracts many vertices into a single one, the attributes of the vertices can be combined and stores as the vertex attributes of the new graph.

The specification of the combination of (vertex or edge) attributes can be given as

1. a character scalar,
2. a function object or
3. a list of character scalars and/or function objects.

If it is a character scalar, then it refers to one of the predefined combinations, see their list below.

If it is a function, then the given function is expected to perform the combination. It will be called once for each new vertex/edge in the graph, with a single argument: the attribute values of the vertices that map to that single vertex.

The third option, a list can be used to specify different combination methods for different attributes. A named entry of the list corresponds to the attribute with the same name. An unnamed entry (i.e. if the name is the empty string) of the list specifies the default combination method. I.e.

```
list(weight="sum", "ignore")
```

specifies that the weight of the new edge should be sum of the weights of the corresponding edges in the old graph; and that the rest of the attributes should be ignored (=dropped).

Predefined combination functions

The following combination behaviors are predefined:

"ignore" The attribute is ignored and dropped.

"sum" The sum of the attributes is calculated. This does not work for character attributes and works for complex attributes only if they have a `sum` generic defined. (E.g. it works for sparse matrices from the `Matrix` package, because they have a `sum` method.)

"prod" The product of the attributes is calculated. This does not work for character attributes and works for complex attributes only if they have a `prod` function defined.

"min" The minimum of the attributes is calculated and returned. For character and complex attributes the standard R `min` function is used.

"max" The maximum of the attributes is calculated and returned. For character and complex attributes the standard R `max` function is used.

"random" Chooses one of the supplied attribute values, uniformly randomly. For character and complex attributes this is implemented by calling `sample`.

"first" Always chooses the first attribute value. It is implemented by calling the `head()` function.

"last" Always chooses the last attribute value. It is implemented by calling the `tail()` function.

"mean" The mean of the attributes is calculated and returned. For character and complex attributes this simply calls the `mean()` function.

"median" The median of the attributes is selected. Calls the R `median()` function for all attribute types.

"concat" Concatenate the attributes, using the `c()` function. This results almost always a complex attribute.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

`graph_attr()`, `vertex_attr()`, `edge_attr()` on how to use graph/vertex/edge attributes in general. `igraph_options()` on igraph parameters.

Vertex, edge and graph attributes: `delete_edge_attr()`, `delete_graph_attr()`, `delete_vertex_attr()`, `edge_attr()`, `edge_attr<-()`, `edge_attr_names()`, `graph_attr()`, `graph_attr<-()`, `graph_attr_names()`, `igraph-dollar`, `igraph-vs-attributes`, `set_edge_attr()`, `set_graph_attr()`, `set_vertex_attr()`, `set_vertex_attrs()`, `vertex_attr()`, `vertex_attr<-()`, `vertex_attr_names()`

Examples

```
g <- make_graph(c(1, 2, 1, 2, 1, 2, 2, 3, 3, 4))
E(g)$weight <- 1:5

## print attribute values with the graph
igraph_options(print.graph.attributes = TRUE)
igraph_options(print.vertex.attributes = TRUE)
igraph_options(print.edge.attributes = TRUE)
```

```
## new attribute is the sum of the old ones
simplify(g, edge.attr.comb = "sum")

## collect attributes into a string
simplify(g, edge.attr.comb = toString)

## concatenate them into a vector, this creates a complex
## attribute
simplify(g, edge.attr.comb = "concat")

E(g)$name <- letters[seq_len(ecount(g))]

## both attributes are collected into strings
simplify(g, edge.attr.comb = toString)

## harmonic average of weights, names are dropped
simplify(g, edge.attr.comb = list(
  weight = function(x) length(x) / sum(1 / x),
  name = "ignore"
))
```

`igraph-dollar` *Getting and setting graph attributes, shortcut*

Description

The `$` operator is a shortcut to get and and set graph attributes. It is shorter and just as readable as `graph_attr()` and `set_graph_attr()`.

Usage

```
## S3 method for class 'igraph'
x$name

## S3 replacement method for class 'igraph'
x$name <- value
```

Arguments

`x`	An igraph graph
`name`	Name of the attribute to get/set.
`value`	New value of the graph attribute.

See Also

Vertex, edge and graph attributes: `delete_edge_attr()`, `delete_graph_attr()`, `delete_vertex_attr()`, `edge_attr()`, `edge_attr<-()`, `edge_attr_names()`, `graph_attr()`, `graph_attr<-()`, `graph_attr_names()`, `igraph-attribute-combination`, `igraph-vs-attributes`, `set_edge_attr()`, `set_graph_attr()`, `set_vertex_attr()`, `set_vertex_attrs()`, `vertex_attr()`, `vertex_attr<-()`, `vertex_attr_names()`

Examples

```
g <- make_ring(10)
g$name
g$name <- "10-ring"
g$name
```

`igraph-es-attributes` *Query or set attributes of the edges in an edge sequence*

Description

The `$` operator is a syntactic sugar to query and set edge attributes, for edges in an edge sequence.

Usage

```
## S3 replacement method for class 'igraph.es'
x[[i]] <- value

## S3 replacement method for class 'igraph.es'
x[i] <- value

## S3 method for class 'igraph.es'
x$name

## S3 replacement method for class 'igraph.es'
x$name <- value

E(x, path = NULL, P = NULL, directed = NULL) <- value
```

Arguments

`x`	An edge sequence. For `E<-` it is a graph.
`i`	Index.
`value`	New value of the attribute, for the edges in the edge sequence.
`name`	Name of the edge attribute to query or set.
`path`	Select edges along a path, given by a vertex sequence See `E()`.
`P`	Select edges via pairs of vertices. See `E()`.
`directed`	Whether to use edge directions for the `path` or `P` arguments.

Details

The query form of `$` is a shortcut for `edge_attr()`, e.g. `E(g)[idx]$attr` is equivalent to `edge_attr(g, attr, E(g)[idx])`.

The assignment form of `$` is a shortcut for `set_edge_attr()`, e.g. `E(g)[idx]$attr <- value` is equivalent to `g <- set_edge_attr(g, attr, E(g)[idx], value)`.

Value

A vector or list, containing the values of the attribute `name` for the edges in the sequence. For numeric, character or logical attributes, it is a vector of the appropriate type, otherwise it is a list.

See Also

Other vertex and edge sequences: `E()`, `V()`, `as_ids()`, `igraph-es-indexing`, `igraph-es-indexing2`, `igraph-vs-attributes`, `igraph-vs-indexing`, `igraph-vs-indexing2`, `print.igraph.es()`, `print.igraph.vs()`

Examples

```
# color edges of the largest component
largest_comp <- function(graph) {
  cl <- components(graph)
  V(graph)[which.max(cl$csize) == cl$membership]
}
g <- sample_(
  gnp(100, 1 / 100),
  with_vertex_(size = 3, label = ""),
  with_graph_(layout = layout_with_fr)
)
giant_v <- largest_comp(g)
E(g)$color <- "orange"
E(g)[giant_v %--% giant_v]$color <- "blue"
plot(g)
```

---

igraph-es-indexing *Indexing edge sequences*

Description

Edge sequences can be indexed very much like a plain numeric R vector, with some extras.

Usage

```
## S3 method for class 'igraph.es'
x[...]
```

Arguments

`x`	An edge sequence
`...`	Indices, see details below.

Value

Another edge sequence, referring to the same graph.

Multiple indices

When using multiple indices within the bracket, all of them are evaluated independently, and then the results are concatenated using the `c()` function. E.g. `E(g)[1, 2, .inc(1)]` is equivalent to `c(E(g)[1], E(g)[2], E(g)[.inc(1)])`.

Index types

Edge sequences can be indexed with positive numeric vectors, negative numeric vectors, logical vectors, character vectors:

- When indexed with positive numeric vectors, the edges at the given positions in the sequence are selected. This is the same as indexing a regular R atomic vector with positive numeric vectors.
- When indexed with negative numeric vectors, the edges at the given positions in the sequence are omitted. Again, this is the same as indexing a regular R atomic vector.
- When indexed with a logical vector, the lengths of the edge sequence and the index must match, and the edges for which the index is `TRUE` are selected.
- Named graphs can be indexed with character vectors, to select edges with the given names. Note that a graph may have edge names and vertex names, and both can be used to select edges. Edge names are simply used as names of the numeric edge id vector. Vertex names effectively only work in graphs without multiple edges, and must be separated with a `|` bar character to select an edges that incident to the two given vertices. See examples below.

Edge attributes

When indexing edge sequences, edge attributes can be referred to simply by using their names. E.g. if a graph has a `weight` edge attribute, then `E(G)[weight > 1]` selects all edges with a weight larger than one. See more examples below. Note that attribute names mask the names of variables present in the calling environment; if you need to look up a variable and you do not want a similarly named edge attribute to mask it, use the `.env` pronoun to perform the name lookup in the calling environment. In other words, use `E(g)[.env$weight > 1]` to make sure that `weight` is looked up from the calling environment even if there is an edge attribute with the same name. Similarly, you can use `.data` to match attribute names only.

Special functions

There are some special igraph functions that can be used only in expressions indexing edge sequences:

`.inc` takes a vertex sequence, and selects all edges that have at least one incident vertex in the vertex sequence.

`.from` similar to `.inc()`, but only the tails of the edges are considered.

`.to` is similar to `.inc()`, but only the heads of the edges are considered.

`\%-\%` a special operator that can be used to select all edges between two sets of vertices. It ignores the edge directions in directed graphs.

`\%->\%` similar to `\%--\%`, but edges *from* the left hand side argument, pointing *to* the right hand side argument, are selected, in directed graphs.

`\%<-\%` similar to `\%--\%`, but edges *to* the left hand side argument, pointing *from* the right hand side argument, are selected, in directed graphs.

Note that multiple special functions can be used together, or with regular indices, and then their results are concatenated. See more examples below.

See Also

Other vertex and edge sequences: `E()`, `V()`, `as_ids()`, `igraph-es-attributes`, `igraph-es-indexing2`, `igraph-vs-attributes`, `igraph-vs-indexing`, `igraph-vs-indexing2`, `print.igraph.es()`, `print.igraph.vs()`

Other vertex and edge sequence operations: `c.igraph.es()`, `c.igraph.vs()`, `difference.igraph.es()`, `difference.igraph.vs()`, `igraph-es-indexing2`, `igraph-vs-indexing`, `igraph-vs-indexing2`, `intersection.igraph.es()`, `intersection.igraph.vs()`, `rev.igraph.es()`, `rev.igraph.vs()`, `union.igraph.es()`, `union.igraph.vs()`, `unique.igraph.es()`, `unique.igraph.vs()`

Examples

```
# -----------------------------------------------------------------
# Special operators for indexing based on graph structure
g <- sample_pa(100, power = 0.3)
E(g)[1:3 %--% 2:6]
E(g)[1:5 %->% 1:6]
E(g)[1:3 %<-% 2:6]

# -----------------------------------------------------------------
# The edges along the diameter
g <- sample_pa(100, directed = FALSE)
d <- get_diameter(g)
E(g, path = d)

# -----------------------------------------------------------------
# Select edges based on attributes
g <- sample_gnp(20, 3 / 20) %>%
  set_edge_attr("weight", value = rnorm(gsize(.)))
E(g)[[weight < 0]]

# Indexing with a variable whose name matches the name of an attribute
# may fail; use .env to force the name lookup in the parent environment
E(g)$x <- E(g)$weight
x <- 2
E(g)[.env$x]
```

---

igraph-es-indexing2 *Select edges and show their metadata*

Description

The double bracket operator can be used on edge sequences, to print the meta-data (edge attributes) of the edges in the sequence.

Usage

```
## S3 method for class 'igraph.es'
x[[...]]
```

Arguments

x	An edge sequence.
...	Additional arguments, passed to [.

Details

Technically, when used with edge sequences, the double bracket operator does exactly the same as the single bracket operator, but the resulting edge sequence is printed differently: all attributes of the edges in the sequence are printed as well.

See [.igraph.es for more about indexing edge sequences.

Value

Another edge sequence, with metadata printing turned on. See details below.

See Also

Other vertex and edge sequences: E(), V(), as_ids(), igraph-es-attributes, igraph-es-indexing, igraph-vs-attributes, igraph-vs-indexing, igraph-vs-indexing2, print.igraph.es(), print.igraph.vs()

Other vertex and edge sequence operations: c.igraph.es(), c.igraph.vs(), difference.igraph.es(), difference.igraph.vs(), igraph-es-indexing, igraph-vs-indexing, igraph-vs-indexing2, intersection.igraph.es(), intersection.igraph.vs(), rev.igraph.es(), rev.igraph.vs(), union.igraph.es(), union.igraph.vs(), unique.igraph.es(), unique.igraph.vs()

Examples

```
g <- make_(
  ring(10),
  with_vertex_(name = LETTERS[1:10]),
  with_edge_(weight = 1:10, color = "green")
)
E(g)
E(g)[[]]
E(g)[[.inc("A")]]
```

`igraph-minus` *Delete vertices or edges from a graph*

Description

Delete vertices or edges from a graph

Usage

```
## S3 method for class 'igraph'
e1 - e2
```

Arguments

`e1`	Left argument, see details below.
`e2`	Right argument, see details below.

Details

The minus operator (‘`-`’) can be used to remove vertices or edges from the graph. The operation performed is selected based on the type of the right hand side argument:

- If it is an igraph graph object, then the difference of the two graphs is calculated, see `difference()`.
- If it is a numeric or character vector, then it is interpreted as a vector of vertex ids and the specified vertices will be deleted from the graph. Example:

```
  g <- make_ring(10)
V(g)$name <- letters[1:10]
g <- g - c("a", "b")
```

- If `e2` is a vertex sequence (e.g. created by the `V()` function), then these vertices will be deleted from the graph.
- If it is an edge sequence (e.g. created by the `E()` function), then these edges will be deleted from the graph.
- If it is an object created with the `vertex()` (or the `vertices()`) function, then all arguments of `vertices()` are concatenated and the result is interpreted as a vector of vertex ids. These vertices will be removed from the graph.
- If it is an object created with the `edge()` (or the `edges()`) function, then all arguments of `edges()` are concatenated and then interpreted as edges to be removed from the graph. Example:

```
  g <- make_ring(10)
V(g)$name <- letters[1:10]
E(g)$name <- LETTERS[1:10]
g <- g - edge("e|f")
g <- g - edge("H")
```

- If it is an object created with the path() function, then all path() arguments are concatenated and then interpreted as a path along which edges will be removed from the graph. Example:

```
  g <- make_ring(10)
V(g)$name <- letters[1:10]
g <- g - path("a", "b", "c", "d")
```

Value

An igraph graph.

See Also

Other functions for manipulating graph structure: +.igraph(), add_edges(), add_vertices(), complementer(), compose(), connect(), contract(), delete_edges(), delete_vertices(), difference(), difference.igraph(), disjoint_union(), edge(), intersection(), intersection.igraph(), path(), permute(), rep.igraph(), reverse_edges(), simplify(), transitive_closure(), union(), union.igraph(), vertex()

---

igraph-vs-attributes *Query or set attributes of the vertices in a vertex sequence*

Description

The $ operator is a syntactic sugar to query and set the attributes of the vertices in a vertex sequence.

Usage

```
## S3 replacement method for class 'igraph.vs'
x[[i]] <- value

## S3 replacement method for class 'igraph.vs'
x[i] <- value

## S3 method for class 'igraph.vs'
x$name

## S3 replacement method for class 'igraph.vs'
x$name <- value

V(x) <- value
```

Arguments

x	A vertex sequence. For V<- it is a graph.
i	Index.
value	New value of the attribute, for the vertices in the vertex sequence.
name	Name of the vertex attribute to query or set.

Details

The query form of `$` is a shortcut for `vertex_attr()`, e.g. `V(g)[idx]$attr` is equivalent to `vertex_attr(g, attr, V(g)[idx])`.

The assignment form of `$` is a shortcut for `set_vertex_attr()`, e.g. `V(g)[idx]$attr <- value` is equivalent to `g <- set_vertex_attr(g, attr, V(g)[idx], value)`.

Value

A vector or list, containing the values of attribute `name` for the vertices in the vertex sequence. For numeric, character or logical attributes, it is a vector of the appropriate type, otherwise it is a list.

See Also

Other vertex and edge sequences: `E()`, `V()`, `as_ids()`, `igraph-es-attributes`, `igraph-es-indexing`, `igraph-es-indexing2`, `igraph-vs-indexing`, `igraph-vs-indexing2`, `print.igraph.es()`, `print.igraph.vs()`

Vertex, edge and graph attributes: `delete_edge_attr()`, `delete_graph_attr()`, `delete_vertex_attr()`, `edge_attr()`, `edge_attr<-()`, `edge_attr_names()`, `graph_attr()`, `graph_attr<-()`, `graph_attr_names()`, `igraph-attribute-combination`, `igraph-dollar`, `set_edge_attr()`, `set_graph_attr()`, `set_vertex_attr()`, `set_vertex_attrs()`, `vertex_attr()`, `vertex_attr<-()`, `vertex_attr_names()`

Examples

```
g <- make_(
  ring(10),
  with_vertex_(
    name = LETTERS[1:10],
    color = sample(1:2, 10, replace = TRUE)
  )
)
V(g)$name
V(g)$color
V(g)$frame.color <- V(g)$color

# color vertices of the largest component
largest_comp <- function(graph) {
  cl <- components(graph)
  V(graph)[which.max(cl$csize) == cl$membership]
}
g <- sample_(
  gnp(100, 2 / 100),
  with_vertex_(size = 3, label = ""),
  with_graph_(layout = layout_with_fr)
)
giant_v <- largest_comp(g)
V(g)$color <- "blue"
V(g)[giant_v]$color <- "orange"
plot(g)
```

igraph-vs-indexing *Indexing vertex sequences*

Description

Vertex sequences can be indexed very much like a plain numeric R vector, with some extras.

Usage

```
## S3 method for class 'igraph.vs'
x[..., na_ok = FALSE]
```

Arguments

x	A vertex sequence.
...	Indices, see details below.
na_ok	Whether it is OK to have NAs in the vertex sequence.

Details

Vertex sequences can be indexed using both the single bracket and the double bracket operators, and they both work the same way. The only difference between them is that the double bracket operator marks the result for printing vertex attributes.

Value

Another vertex sequence, referring to the same graph.

Multiple indices

When using multiple indices within the bracket, all of them are evaluated independently, and then the results are concatenated using the `c()` function (except for the `na_ok` argument, which is special an must be named. E.g. `V(g)[1, 2, .nei(1)]` is equivalent to `c(V(g)[1], V(g)[2], V(g)[.nei(1)])`.

Index types

Vertex sequences can be indexed with positive numeric vectors, negative numeric vectors, logical vectors, character vectors:

- When indexed with positive numeric vectors, the vertices at the given positions in the sequence are selected. This is the same as indexing a regular R atomic vector with positive numeric vectors.
- When indexed with negative numeric vectors, the vertices at the given positions in the sequence are omitted. Again, this is the same as indexing a regular R atomic vector.
- When indexed with a logical vector, the lengths of the vertex sequence and the index must match, and the vertices for which the index is `TRUE` are selected.
- Named graphs can be indexed with character vectors, to select vertices with the given names.

Vertex attributes

When indexing vertex sequences, vertex attributes can be referred to simply by using their names. E.g. if a graph has a `name` vertex attribute, then `V(g)[name == "foo"]` is equivalent to `V(g)[V(g)$name == "foo"]`. See more examples below. Note that attribute names mask the names of variables present in the calling environment; if you need to look up a variable and you do not want a similarly named vertex attribute to mask it, use the `.env` pronoun to perform the name lookup in the calling environment. In other words, use `V(g)[.env$name == "foo"]` to make sure that `name` is looked up from the calling environment even if there is a vertex attribute with the same name. Similarly, you can use `.data` to match attribute names only.

Special functions

There are some special igraph functions that can be used only in expressions indexing vertex sequences:

`.nei` takes a vertex sequence as its argument and selects neighbors of these vertices. An optional `mode` argument can be used to select successors (`mode="out"`), or predecessors (`mode="in"`) in directed graphs.

`.inc` Takes an edge sequence as an argument, and selects vertices that have at least one incident edge in this edge sequence.

`.from` Similar to `.inc`, but only considers the tails of the edges.

`.to` Similar to `.inc`, but only considers the heads of the edges.

`.innei`, `.outnei` `.innei(v)` is a shorthand for `.nei(v, mode = "in")`, and `.outnei(v)` is a shorthand for `.nei(v, mode = "out")`.

Note that multiple special functions can be used together, or with regular indices, and then their results are concatenated. See more examples below.

See Also

Other vertex and edge sequences: `E()`, `V()`, `as_ids()`, `igraph-es-attributes`, `igraph-es-indexing`, `igraph-es-indexing2`, `igraph-vs-attributes`, `igraph-vs-indexing2`, `print.igraph.es()`, `print.igraph.vs()`

Other vertex and edge sequence operations: `c.igraph.es()`, `c.igraph.vs()`, `difference.igraph.es()`, `difference.igraph.vs()`, `igraph-es-indexing`, `igraph-es-indexing2`, `igraph-vs-indexing2`, `intersection.igraph.es()`, `intersection.igraph.vs()`, `rev.igraph.es()`, `rev.igraph.vs()`, `union.igraph.es()`, `union.igraph.vs()`, `unique.igraph.es()`, `unique.igraph.vs()`

Examples

```
# ---------------------------------------------------------------
# Setting attributes for subsets of vertices
largest_comp <- function(graph) {
  cl <- components(graph)
  V(graph)[which.max(cl$csize) == cl$membership]
}
g <- sample_(
  gnp(100, 2 / 100),
  with_vertex_(size = 3, label = ""),
```

```
  with_graph_(layout = layout_with_fr)
)
giant_v <- largest_comp(g)
V(g)$color <- "green"
V(g)[giant_v]$color <- "red"
plot(g)

# ----------------------------------------------------------------
# nei() special function
g <- make_graph(c(1, 2, 2, 3, 2, 4, 4, 2))
V(g)[.nei(c(2, 4))]
V(g)[.nei(c(2, 4), "in")]
V(g)[.nei(c(2, 4), "out")]

# ----------------------------------------------------------------
# The same with vertex names
g <- make_graph(~ A -+ B, B -+ C:D, D -+ B)
V(g)[.nei(c("B", "D"))]
V(g)[.nei(c("B", "D"), "in")]
V(g)[.nei(c("B", "D"), "out")]

# ----------------------------------------------------------------
# Resolving attributes
g <- make_graph(~ A -+ B, B -+ C:D, D -+ B)
V(g)$color <- c("red", "red", "green", "green")
V(g)[color == "red"]

# Indexing with a variable whose name matches the name of an attribute
# may fail; use .env to force the name lookup in the parent environment
V(g)$x <- 10:13
x <- 2
V(g)[.env$x]
```

`igraph-vs-indexing2` *Select vertices and show their metadata*

Description

The double bracket operator can be used on vertex sequences, to print the meta-data (vertex attributes) of the vertices in the sequence.

Usage

```
## S3 method for class 'igraph.vs'
x[[...]]
```

Arguments

`x`	A vertex sequence.
`...`	Additional arguments, passed to `[`.

Details

Technically, when used with vertex sequences, the double bracket operator does exactly the same as the single bracket operator, but the resulting vertex sequence is printed differently: all attributes of the vertices in the sequence are printed as well.

See `[.igraph.vs` for more about indexing vertex sequences.

Value

The double bracket operator returns another vertex sequence, with meta-data (attribute) printing turned on. See details below.

See Also

Other vertex and edge sequences: `E()`, `V()`, `as_ids()`, `igraph-es-attributes`, `igraph-es-indexing`, `igraph-es-indexing2`, `igraph-vs-attributes`, `igraph-vs-indexing`, `print.igraph.es()`, `print.igraph.vs()`

Other vertex and edge sequence operations: `c.igraph.es()`, `c.igraph.vs()`, `difference.igraph.es()`, `difference.igraph.vs()`, `igraph-es-indexing`, `igraph-es-indexing2`, `igraph-vs-indexing`, `intersection.igraph.es()`, `intersection.igraph.vs()`, `rev.igraph.es()`, `rev.igraph.vs()`, `union.igraph.es()`, `union.igraph.vs()`, `unique.igraph.es()`, `unique.igraph.vs()`

Examples

```
g <- make_ring(10) %>%
  set_vertex_attr("color", value = "red") %>%
  set_vertex_attr("name", value = LETTERS[1:10])
V(g)
V(g)[[]]
V(g)[1:5]
V(g)[[1:5]]
```

---

igraph_options *Parameters for the igraph package*

Description

igraph has some parameters which (usually) affect the behavior of many functions. These can be set for the whole session via `igraph_options()`.

Usage

```
igraph_options(...)

igraph_opt(x, default = NULL)
```

Arguments

...	A list may be given as the only argument, or any number of arguments may be in the name=value form, or no argument at all may be given. See the Value and Details sections for explanation.
x	A character string holding an option name.
default	If the specified option is not set in the options list, this value is returned. This facilitates retrieving an option and checking whether it is set and setting it separately if not.

Details

The parameter values set via a call to the igraph_options() function will remain in effect for the rest of the session, affecting the subsequent behaviour of the other functions of the igraph package for which the given parameters are relevant.

This offers the possibility of customizing the functioning of the igraph package, for instance by insertions of appropriate calls to igraph_options() in a load hook for package **igraph**.

The currently used parameters in alphabetical order:

add.params Logical scalar, whether to add model parameter to the graphs that are created by the various graph constructors. By default it is TRUE.

add.vertex.names Logical scalar, whether to add vertex names to node level indices, like degree, betweenness scores, etc. By default it is TRUE.

annotate.plot Logical scalar, whether to annotate igraph plots with the graph's name (name graph attribute, if present) as main, and with the number of vertices and edges as xlab. Defaults to FALSE.

dend.plot.type The plotting function to use when plotting community structure dendrograms via plot_dendrogram(). Possible values are 'auto' (the default), 'phylo', 'hclust' and 'dendrogram'. See plot_dendrogram() for details.

edge.attr.comb Specifies what to do with the edge attributes if the graph is modified. The default value is list(weight="sum", name="concat", "ignore"). See attribute.combination() for details on this.

print.edge.attributes Logical constant, whether to print edge attributes when printing graphs. Defaults to FALSE.

print.full Logical scalar, whether print.igraph() should show the graph structure as well, or only a summary of the graph.

print.graph.attributes Logical constant, whether to print graph attributes when printing graphs. Defaults to FALSE.

print.vertex.attributes Logical constant, whether to print vertex attributes when printing graphs. Defaults to FALSE.

return.vs.es Whether functions that return a set or sequence of vertices/edges should return formal vertex/edge sequence objects. This option was introduced in igraph version 1.0.0 and defaults to TRUE. If your package requires the old behavior, you can set it to FALSE in the .onLoad function of your package, without affecting other packages.

sparsematrices Whether to use the Matrix package for (sparse) matrices. It is recommended, if the user works with larger graphs.

verbose Logical constant, whether igraph functions should talk more than minimal. E.g. if TRUE then some functions will use progress bars while computing. Defaults to FALSE.

vertex.attr.comb Specifies what to do with the vertex attributes if the graph is modified. The default value is list(name="concat", "ignore"). See attribute.combination() for details on this.

Value

igraph_options() returns a list with the old values of the updated parameters, invisibly. Without any arguments, it returns the values of all options.

For igraph_opt(), the current value set for option x, or NULL if the option is unset.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

igraph_options() is similar to options() and igraph_opt() is similar to getOption().

Other igraph options: with_igraph_opt()

Examples

```
oldval <- igraph_opt("verbose")
igraph_options(verbose = TRUE)
layout_with_kk(make_ring(10))
igraph_options(verbose = oldval)

oldval <- igraph_options(verbose = TRUE, sparsematrices = FALSE)
make_ring(10)[]
igraph_options(oldval)
igraph_opt("verbose")
```

---

incident *Incident edges of a vertex in a graph*

Description

Incident edges of a vertex in a graph

Usage

```
incident(graph, v, mode = c("all", "out", "in", "total"))
```

Arguments

graph	The input graph.
v	The vertex of which the incident edges are queried.
mode	Whether to query outgoing ('out'), incoming ('in') edges, or both types ('all'). This is ignored for undirected graphs.

Value

An edge sequence containing the incident edges of the input vertex.

Related documentation in the C library

incident(), is_directed(), ecount(), edges(), vcount(), get_eids()

See Also

Other structural queries: [.igraph(), [[.igraph(), adjacent_vertices(), are_adjacent(), ends(), get_edge_ids(), gorder(), gsize(), head_of(), incident_edges(), is_directed(), neighbors(), tail_of()

Examples

```
g <- make_graph("Zachary")
incident(g, 1)
incident(g, 34)
```

---

incident_edges *Incident edges of multiple vertices in a graph*

Description

This function is similar to incident(), but it queries multiple vertices at once.

Usage

```
incident_edges(graph, v, mode = c("out", "in", "all", "total"))
```

Arguments

graph	Input graph.
v	The vertices to query
mode	Whether to query outgoing ('out'), incoming ('in') edges, or both types ('all'). This is ignored for undirected graphs.

Value

A list of edge sequences.

Related documentation in the C library

vcount(), edges(), get_eids(), ecount()

See Also

Other structural queries: [.igraph(), [[.igraph(), adjacent_vertices(), are_adjacent(), ends(), get_edge_ids(), gorder(), gsize(), head_of(), incident(), is_directed(), neighbors(), tail_of()

Examples

```
g <- make_graph("Zachary")
incident_edges(g, c(1, 34))
```

---

indent_print *Indent a printout*

Description

Indent a printout

Usage

```
indent_print(..., .indent = " ", .printer = print)
```

Arguments

...	Passed to the printing function.
.indent	Character scalar, indent the printout with this.
.printer	The printing function, defaults to print.

Value

The first element in ..., invisibly.

intersection *Intersection of two or more sets*

Description

This is an S3 generic function. See `methods("intersection")` for the actual implementations for various S3 classes. Initially it is implemented for igraph graphs and igraph vertex and edge sequences. See `intersection.igraph()`, and `intersection.igraph.vs()`.

Usage

```
intersection(...)
```

Arguments

`...`	Arguments, their number and interpretation depends on the function that implements `intersection()`.

Value

Depends on the function that implements this method.

See Also

Other functions for manipulating graph structure: `+.igraph()`, `add_edges()`, `add_vertices()`, `complementer()`, `compose()`, `connect()`, `contract()`, `delete_edges()`, `delete_vertices()`, `difference()`, `difference.igraph()`, `disjoint_union()`, `edge()`, `igraph-minus`, `intersection.igraph()`, `path()`, `permute()`, `rep.igraph()`, `reverse_edges()`, `simplify()`, `transitive_closure()`, `union()`, `union.igraph()`, `vertex()`

intersection.igraph *Intersection of graphs*

Description

The intersection of two or more graphs are created. The graphs may have identical or overlapping vertex sets.

Usage

```
## S3 method for class 'igraph'
intersection(..., byname = "auto", keep.all.vertices = TRUE)
```

Arguments

...	Graph objects or lists of graph objects.
byname	A logical scalar, or the character scalar auto. Whether to perform the operation based on symbolic vertex names. If it is auto, that means TRUE if all graphs are named and FALSE otherwise. A warning is generated if auto and some (but not all) graphs are named.
keep.all.vertices	Logical scalar, whether to keep vertices that only appear in a subset of the input graphs.

Details

intersection() creates the intersection of two or more graphs: only edges present in all graphs will be included. The corresponding operator is %s%.

If the byname argument is TRUE (or auto and all graphs are named), then the operation is performed on symbolic vertex names instead of the internal numeric vertex ids.

intersection() keeps the attributes of all graphs. All graph, vertex and edge attributes are copied to the result. If an attribute is present in multiple graphs and would result a name clash, then this attribute is renamed by adding suffixes: _1, _2, etc.

The name vertex attribute is treated specially if the operation is performed based on symbolic vertex names. In this case name must be present in all graphs, and it is not renamed in the result graph.

An error is generated if some input graphs are directed and others are undirected.

Value

A new graph object.

Related documentation in the C library

vcount(), permute_vertices(), edges(), get_eids(), ecount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Other functions for manipulating graph structure: +.igraph(), add_edges(), add_vertices(), complementer(), compose(), connect(), contract(), delete_edges(), delete_vertices(), difference(), difference.igraph(), disjoint_union(), edge(), igraph-minus, intersection(), path(), permute(), rep.igraph(), reverse_edges(), simplify(), transitive_closure(), union(), union.igraph(), vertex()

Examples

```
## Common part of two social networks
net1 <- graph_from_literal(
  D - A:B:F:G, A - C - F - A, B - E - G - B, A - B, F - G,
  H - F:G, H - I - J
)
net2 <- graph_from_literal(D - A:F:Y, B - A - X - F - H - Z, F - Y)
print_all(net1 %s% net2)
```

---

intersection.igraph.es

Intersection of edge sequences

Description

Intersection of edge sequences

Usage

```
## S3 method for class 'igraph.es'
intersection(...)
```

Arguments

...	The edge sequences to take the intersection of.

Details

They must belong to the same graph. Note that this function has 'set' semantics and the multiplicity of edges is lost in the result.

Value

An edge sequence that contains edges that appear in all given sequences, each edge exactly once.

See Also

Other vertex and edge sequence operations: c.igraph.es(), c.igraph.vs(), difference.igraph.es(), difference.igraph.vs(), igraph-es-indexing, igraph-es-indexing2, igraph-vs-indexing, igraph-vs-indexing2, intersection.igraph.vs(), rev.igraph.es(), rev.igraph.vs(), union.igraph.es(), union.igraph.vs(), unique.igraph.es(), unique.igraph.vs()

Examples

```
g <- make_(ring(10), with_vertex_(name = LETTERS[1:10]))
intersection(E(g)[1:6], E(g)[5:9])
```

`intersection.igraph.vs` *Intersection of vertex sequences*

Description

Intersection of vertex sequences

Usage

```
## S3 method for class 'igraph.vs'
intersection(...)
```

Arguments

`...`	The vertex sequences to take the intersection of.

Details

They must belong to the same graph. Note that this function has 'set' semantics and the multiplicity of vertices is lost in the result.

Value

A vertex sequence that contains vertices that appear in all given sequences, each vertex exactly once.

See Also

Other vertex and edge sequence operations: `c.igraph.es()`, `c.igraph.vs()`, `difference.igraph.es()`, `difference.igraph.vs()`, `igraph-es-indexing`, `igraph-es-indexing2`, `igraph-vs-indexing`, `igraph-vs-indexing2`, `intersection.igraph.es()`, `rev.igraph.es()`, `rev.igraph.vs()`, `union.igraph.es()`, `union.igraph.vs()`, `unique.igraph.es()`, `unique.igraph.vs()`

Examples

```
g <- make_(ring(10), with_vertex_(name = LETTERS[1:10]))
intersection(E(g)[1:6], E(g)[5:9])
```

`invalidate_cache` *Invalidate the cache of a graph*

Description

igraph graphs cache some basic properties (such as whether the graph is a DAG or whether it is simple) in an internal data structure for faster repeated queries. This function invalidates the cache, forcing a recalculation of the cached properties the next time they are needed.

Usage

```
invalidate_cache(graph)
```

Arguments

`graph`	The graph whose cache is to be invalidated.

Details

You should not need to call this function during normal usage; however, it may be useful for debugging cache-related issues. A tell-tale sign of an invalid cache entry is when the result of a cached function (such as `is_dag()` or `is_simple()`) changes after calling this function.

Value

The graph with its cache invalidated. Since the graph is modified in place in R as well, you can also ignore the return value.

Related documentation in the C library

`invalidate_cache()`

Examples

```
g <- make_ring(10)
# Cache is populated when calling is_simple()
is_simple(g)
# Invalidate cache (for debugging purposes)
invalidate_cache(g)
# Result should be the same
is_simple(g)
```

is_acyclic *Acyclic graphs*

Description

This function tests whether the given graph is free of cycles.

Usage

```
is_acyclic(graph)
```

Arguments

graph	The input graph.

Details

This function looks for directed cycles in directed graphs and undirected cycles in undirected graphs. Use find_cycle() to return a specific cycle.

Value

A logical vector of length one.

Related documentation in the C library

is_acyclic()

See Also

is_forest() and is_dag() for functions specific to undirected and directed graphs.

Graph cycles: feedback_arc_set(), feedback_vertex_set(), find_cycle(), girth(), has_eulerian_path(), is_dag(), simple_cycles()

Other structural.properties: bfs(), component_distribution(), connect(), constraint(), coreness(), degree(), dfs(), distance_table(), edge_density(), feedback_arc_set(), feedback_vertex_set(), girth(), is_dag(), is_matching(), k_shortest_paths(), knn(), reciprocity(), subcomponent(), subgraph(), topo_sort(), transitivity(), unfold_tree(), which_multiple(), which_mutual()

Examples

```
g <- make_graph(c(1, 2, 1, 3, 2, 4, 3, 4), directed = TRUE)
is_acyclic(g)
is_acyclic(as_undirected(g))
```

`is_biconnected` *Check biconnectedness*

Description

[Experimental]

Tests whether a graph is biconnected.

Usage

```
is_biconnected(graph)
```

Arguments

`graph`	The input graph. Edge directions are ignored.

Details

A graph is biconnected if the removal of any single vertex (and its adjacent edges) does not disconnect it.

igraph does not consider single-vertex graphs biconnected.

Note that some authors do not consider the graph consisting of two connected vertices as biconnected, however, igraph does.

Value

Logical, `TRUE` if the graph is biconnected.

Related documentation in the C library

`is_biconnected()`

See Also

`articulation_points()`, `biconnected_components()`, `is_connected()`, `vertex_connectivity()`

Connected components: `articulation_points()`, `biconnected_components()`, `component_distribution()`, `count_reachable()`, `decompose()`

Examples

```
is_biconnected(make_graph("bull"))
is_biconnected(make_graph("dodecahedron"))
is_biconnected(make_full_graph(1))
is_biconnected(make_full_graph(2))
```

is_bipartite *Checks whether the graph has a vertex attribute called* type.

Description

It does not check whether the graph is bipartite in the mathematical sense. Use `bipartite_mapping()` for that.

Usage

```
is_bipartite(graph)
```

Arguments

graph	The input graph

See Also

Bipartite graphs: `bipartite_mapping()`, `bipartite_projection()`, `make_bipartite_graph()`

is_chordal *Chordality of a graph*

Description

A graph is chordal (or triangulated) if each of its cycles of four or more nodes has a chord, which is an edge joining two nodes that are not adjacent in the cycle. An equivalent definition is that any chordless cycles have at most three nodes.

Usage

```
is_chordal(
  graph,
  alpha = NULL,
  alpham1 = NULL,
  fillin = FALSE,
  newgraph = FALSE
)
```

Arguments

graph	The input graph. It may be directed, but edge directions are ignored, as the algorithm is defined for undirected graphs.
alpha	Numeric vector, the maximal chardinality ordering of the vertices. If it is NULL, then it is automatically calculated by calling max_cardinality(), or from alpham1 if that is given..
alpham1	Numeric vector, the inverse of alpha. If it is NULL, then it is automatically calculated by calling max_cardinality(), or from alpha.
fillin	Logical scalar, whether to calculate the fill-in edges.
newgraph	Logical scalar, whether to calculate the triangulated graph.

Details

The chordality of the graph is decided by first performing maximum cardinality search on it (if the alpha and alpham1 arguments are NULL), and then calculating the set of fill-in edges.

The set of fill-in edges is empty if and only if the graph is chordal.

It is also true that adding the fill-in edges to the graph makes it chordal.

Value

A list with three members:

chordal Logical scalar, it is TRUE iff the input graph is chordal.

fillin If requested, then a numeric vector giving the fill-in edges. NULL otherwise.

newgraph If requested, then the triangulated graph, an igraph object. NULL otherwise.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Robert E Tarjan and Mihalis Yannakakis. (1984). Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal of Computation* 13, 566–579.

See Also

max_cardinality()

Other chordal: max_cardinality()

Examples

```
## The examples from the Tarjan-Yannakakis paper
g1 <- graph_from_literal(
  A - B:C:I, B - A:C:D, C - A:B:E:H, D - B:E:F,
  E - C:D:F:H, F - D:E:G, G - F:H, H - C:E:G:I,
  I - A:H
)
max_cardinality(g1)
is_chordal(g1, fillin = TRUE)

g2 <- graph_from_literal(
  A - B:E, B - A:E:F:D, C - E:D:G, D - B:F:E:C:G,
  E - A:B:C:D:F, F - B:D:E, G - C:D:H:I, H - G:I:J,
  I - G:H:J, J - H:I
)
max_cardinality(g2)
is_chordal(g2, fillin = TRUE)
```

---

is_complete *Is this a complete graph?*

Description

A graph is considered complete if there is an edge between all distinct directed pairs of vertices. igraph considers both the singleton graph and the null graph complete.

Usage

```
is_complete(graph)
```

Arguments

graph	The input graph.

Value

True if the graph is complete.

Related documentation in the C library

is_complete()

See Also

make_full_graph()

Other cliques: cliques(), ivs(), weighted_cliques()

Examples

```
g <- make_full_graph(6, directed = TRUE)
is_complete(g)
g <- delete_edges(g, 1)
is_complete(g)
g <- as_undirected(g)
is_complete(g)
```

---

is_dag *Directed acyclic graphs*

Description

This function tests whether the given graph is a DAG, a directed acyclic graph.

Usage

```
is_dag(graph)
```

Arguments

graph	The input graph. It may be undirected, in which case FALSE is reported.

Details

is_dag() checks whether there is a directed cycle in the graph. If not, the graph is a DAG.

Value

A logical vector of length one.

Related documentation in the C library

is_dag()

Author(s)

Tamas Nepusz <ntamas@gmail.com> for the C code, Gabor Csardi <csardi.gabor@gmail.com> for the R interface.

See Also

Graph cycles: feedback_arc_set(), feedback_vertex_set(), find_cycle(), girth(), has_eulerian_path(), is_acyclic(), simple_cycles()

Other structural.properties: bfs(), component_distribution(), connect(), constraint(), coreness(), degree(), dfs(), distance_table(), edge_density(), feedback_arc_set(), feedback_vertex_set(), girth(), is_acyclic(), is_matching(), k_shortest_paths(), knn(), reciprocity(), subcomponent(), subgraph(), topo_sort(), transitivity(), unfold_tree(), which_multiple(), which_mutual()

Examples

```
g <- make_tree(10)
is_dag(g)
g2 <- g + edge(5, 1)
is_dag(g2)
```

`is_degseq` *Check if a degree sequence is valid for a multi-graph*

Description

`is_degseq()` checks whether the given vertex degrees (in- and out-degrees for directed graphs) can be realized by a graph. Note that the graph does not have to be simple, it may contain loop and multiple edges. For undirected graphs, it also checks whether the sum of degrees is even. For directed graphs, the function checks whether the lengths of the two degree vectors are equal and whether their sums are also equal. These are known sufficient and necessary conditions for a degree sequence to be valid.

Usage

```
is_degseq(out.deg, in.deg = NULL)
```

Arguments

`out.deg`	Integer vector, the degree sequence for undirected graphs, or the out-degree sequence for directed graphs.
`in.deg`	`NULL` or an integer vector. For undirected graphs, it should be `NULL`. For directed graphs it specifies the in-degrees.

Value

A logical scalar.

Related documentation in the C library

`is_graphical()`

Author(s)

Tamás Nepusz <ntamas@gmail.com> and Szabolcs Horvát <szhorvat@gmail.com>

References

Z Király, Recognizing graphic degree sequences and generating all realizations. TR-2011-11, Egerváry Research Group, H-1117, Budapest, Hungary. ISSN 1587-4451 (2012).

B. Cloteaux, Is This for Real? Fast Graphicality Testing, *Comput. Sci. Eng.* 17, 91 (2015).

A. Berger, A note on the characterization of digraphic sequences, *Discrete Math.* 314, 38 (2014).

G. Cairns and S. Mendan, Degree Sequence for Graphs with Loops (2013).

See Also

Other graphical degree sequences: is_graphical()

Examples

```
g <- sample_gnp(100, 2 / 100)
is_degseq(degree(g))
is_graphical(degree(g))
```

---

is_directed *Check whether a graph is directed*

Description

Check whether a graph is directed

Usage

```
is_directed(graph)
```

Arguments

graph	The input graph

Value

Logical scalar, whether the graph is directed.

Related documentation in the C library

is_directed()

See Also

Other structural queries: [.igraph(), [[.igraph(), adjacent_vertices(), are_adjacent(), ends(), get_edge_ids(), gorder(), gsize(), head_of(), incident(), incident_edges(), neighbors(), tail_of()

Examples

```
g <- make_ring(10)
is_directed(g)

g2 <- make_ring(10, directed = TRUE)
is_directed(g2)
```

`is_forest` *Decide whether a graph is a forest.*

Description

`is_forest()` decides whether a graph is a forest, and optionally returns a set of possible root vertices for its components.

Usage

```
is_forest(graph, mode = c("out", "in", "all", "total"), details = FALSE)
```

Arguments

`graph`	An igraph graph object
`mode`	Whether to consider edge directions in a directed graph. 'all' ignores edge directions; 'out' requires edges to be oriented outwards from the root, 'in' requires edges to be oriented towards the root.
`details`	Whether to return only whether the graph is a tree (`FALSE`) or also a possible root (`TRUE`)

Details

An undirected graph is a forest if it has no cycles. In the directed case, a possible additional requirement is that edges in each tree are oriented away from the root (out-trees or arborescences) or all edges are oriented towards the root (in-trees or anti-arborescences). This test can be controlled using the mode parameter.

By convention, the null graph (i.e. the graph with no vertices) is considered to be a forest.

Value

When `details` is `FALSE`, a logical value that indicates whether the graph is a tree. When `details` is `TRUE`, a named list with two entries:

res Logical value that indicates whether the graph is a tree.

root The root vertex of the tree; undefined if the graph is not a tree.

Related documentation in the C library

`is_forest()`, `vcount()`

See Also

Other trees: `is_tree()`, `make_from_prufer()`, `sample_spanning_tree()`, `to_prufer()`

Examples

```
g <- make_tree(3) + make_tree(5, 3)
is_forest(g)
is_forest(g, details = TRUE)
```

`is_graphical` *Is a degree sequence graphical?*

Description

Determine whether the given vertex degrees (in- and out-degrees for directed graphs) can be realized by a graph.

Usage

```
is_graphical(
  out.deg,
  in.deg = NULL,
  allowed.edge.types = c("simple", "loops", "multi", "all")
)
```

Arguments

`out.deg`	Integer vector, the degree sequence for undirected graphs, or the out-degree sequence for directed graphs.
`in.deg`	`NULL` or an integer vector. For undirected graphs, it should be `NULL`. For directed graphs it specifies the in-degrees.
`allowed.edge.types`	The allowed edge types in the graph. 'simple' means that neither loop nor multiple edges are allowed (i.e. the graph must be simple). 'loops' means that loop edges are allowed but mutiple edges are not. 'multi' means that multiple edges are allowed but loop edges are not. 'all' means that both loop edges and multiple edges are allowed.

Details

The classical concept of graphicality assumes simple graphs. This function can perform the check also when self-loops, multi-edges, or both are allowed in the graph.

Value

A logical scalar.

Related documentation in the C library

`is_graphical()`

Author(s)

Tamás Nepusz <ntamas@gmail.com>

References

Hakimi SL: On the realizability of a set of integers as degrees of the vertices of a simple graph. *J SIAM Appl Math* 10:496-506, 1962.

PL Erdős, I Miklós and Z Toroczkai: A simple Havel-Hakimi type algorithm to realize graphical degree sequences of directed graphs. *The Electronic Journal of Combinatorics* 17(1):R66, 2010.

See Also

Other graphical degree sequences: `is_degseq()`

Examples

```
g <- sample_gnp(100, 2 / 100)
is_degseq(degree(g))
is_graphical(degree(g))
```

---

is_igraph *Is this object an igraph graph?*

Description

Is this object an igraph graph?

Usage

```
is_igraph(graph)
```

Arguments

graph	An R object.

Value

A logical constant, TRUE if argument graph is a graph object.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

Examples

```
g <- make_ring(10)
is_igraph(g)
is_igraph(numeric(10))
```

is_matching *Matching*

Description

A matching in a graph means the selection of a set of edges that are pairwise non-adjacent, i.e. they have no common incident vertices. A matching is maximal if it is not a proper subset of any other matching.

Usage

```
is_matching(graph, matching, types = NULL)

is_max_matching(graph, matching, types = NULL)

max_bipartite_match(
  graph,
  types = NULL,
  weights = NULL,
  eps = .Machine$double.eps
)
```

Arguments

graph	The input graph. It might be directed, but edge directions will be ignored.
matching	A potential matching. An integer vector that gives the pair in the matching for each vertex. For vertices without a pair, supply NA here.
types	Vertex types, if the graph is bipartite. By default they are taken from the 'type' vertex attribute, if present.
weights	Potential edge weights. If the graph has an edge attribute called 'weight', and this argument is NULL, then the edge attribute is used automatically. In weighted matching, the weights of the edges must match as much as possible.
eps	A small real number used in equality tests in the weighted bipartite matching algorithm. Two real numbers are considered equal in the algorithm if their difference is smaller than eps. This is required to avoid the accumulation of numerical errors. By default it is set to the smallest x, such that $1 + x \neq 1$ holds. If you are running the algorithm with no weights, this argument is ignored.

Details

is_matching() checks a matching vector and verifies whether its length matches the number of vertices in the given graph, its values are between zero (inclusive) and the number of vertices (inclusive), and whether there exists a corresponding edge in the graph for every matched vertex pair. For bipartite graphs, it also verifies whether the matched vertices are in different parts of the graph.

is_max_matching() checks whether a matching is maximal. A matching is maximal if and only if there exists no unmatched vertex in a graph such that one of its neighbors is also unmatched.

max_bipartite_match() calculates a maximum matching in a bipartite graph. A matching in a bipartite graph is a partial assignment of vertices of the first kind to vertices of the second kind such that each vertex of the first kind is matched to at most one vertex of the second kind and vice versa, and matched vertices must be connected by an edge in the graph. The size (or cardinality) of a matching is the number of edges. A matching is a maximum matching if there exists no other matching with larger cardinality. For weighted graphs, a maximum matching is a matching whose edges have the largest possible total weight among all possible matchings.

Maximum matchings in bipartite graphs are found by the push-relabel algorithm with greedy initialization and a global relabeling after every $n/2$ steps where n is the number of vertices in the graph.

Value

is_matching() and is_max_matching() return a logical scalar.

max_bipartite_match() returns a list with components:

matching_size The size of the matching, i.e. the number of edges connecting the matched vertices.

matching_weight The weights of the matching, if the graph was weighted. For unweighted graphs this is the same as the size of the matching.

matching The matching itself. Numeric vertex id, or vertex names if the graph was named. Non-matched vertices are denoted by NA.

Related documentation in the C library

is_matching(), vcount(), maximum_bipartite_matching(), edges(), get_eids(), ecount(), is_maximal_matching()

Author(s)

Tamas Nepusz <ntamas@gmail.com>

See Also

Other structural.properties: bfs(), component_distribution(), connect(), constraint(), coreness(), degree(), dfs(), distance_table(), edge_density(), feedback_arc_set(), feedback_vertex_set(), girth(), is_acyclic(), is_dag(), k_shortest_paths(), knn(), reciprocity(), subcomponent(), subgraph(), topo_sort(), transitivity(), unfold_tree(), which_multiple(), which_mutual()

Examples

```
g <- graph_from_literal(a - b - c - d - e - f)
m1 <- c("b", "a", "d", "c", "f", "e") # maximal matching
m2 <- c("b", "a", "d", "c", NA, NA) # non-maximal matching
m3 <- c("b", "c", "d", "c", NA, NA) # not a matching
is_matching(g, m1)
is_matching(g, m2)
is_matching(g, m3)
is_max_matching(g, m1)
is_max_matching(g, m2)
```

```
is_max_matching(g, m3)

V(g)$type <- rep(c(FALSE, TRUE), 3)
print_all(g, v = TRUE)
max_bipartite_match(g)

g2 <- graph_from_literal(a - b - c - d - e - f - g)
V(g2)$type <- rep(c(FALSE, TRUE), length.out = vcount(g2))
print_all(g2, v = TRUE)
max_bipartite_match(g2)
#' @keywords graphs
```

`is_min_separator` *Minimal vertex separators*

Description

Check whether a given set of vertices is a minimal vertex separator.

Usage

```
is_min_separator(graph, candidate)
```

Arguments

`graph`	The input graph. It may be directed, but edge directions are ignored.
`candidate`	A numeric vector giving the vertex ids of the candidate separator.

Details

`is_min_separator()` decides whether the supplied vertex set is a minimal vertex separator. A minimal vertex separator is a vertex separator, such that none of its proper subsets are a vertex separator.

Value

A logical scalar, whether the supplied vertex set is a (minimal) vertex separator or not.

Related documentation in the C library

`is_minimal_separator()`, `vcount()`

See Also

Other flow: `dominator_tree()`, `edge_connectivity()`, `is_separator()`, `max_flow()`, `min_cut()`, `min_separators()`, `min_st_separators()`, `st_cuts()`, `st_min_cuts()`, `vertex_connectivity()`

Examples

```
# The graph from the Moody-White paper
mw <- graph_from_literal(
  1 - 2:3:4:5:6, 2 - 3:4:5:7, 3 - 4:6:7, 4 - 5:6:7,
  5 - 6:7:21, 6 - 7, 7 - 8:11:14:19, 8 - 9:11:14, 9 - 10,
  10 - 12:13, 11 - 12:14, 12 - 16, 13 - 16, 14 - 15, 15 - 16,
  17 - 18:19:20, 18 - 20:21, 19 - 20:22:23, 20 - 21,
  21 - 22:23, 22 - 23
)

# Cohesive subgraphs
mw1 <- induced_subgraph(mw, as.character(c(1:7, 17:23)))
mw2 <- induced_subgraph(mw, as.character(7:16))
mw3 <- induced_subgraph(mw, as.character(17:23))
mw4 <- induced_subgraph(mw, as.character(c(7, 8, 11, 14)))
mw5 <- induced_subgraph(mw, as.character(1:7))

check.sep <- function(G) {
  sep <- min_separators(G)
  sapply(sep, is_min_separator, graph = G)
}

check.sep(mw)
check.sep(mw1)
check.sep(mw2)
check.sep(mw3)
check.sep(mw4)
check.sep(mw5)
```

---

is_named *Named graphs*

Description

An igraph graph is named, if there is a symbolic name associated with its vertices.

Usage

```
is_named(graph)
```

Arguments

graph	The input graph.

Details

In igraph vertices can always be identified and specified via their numeric vertex ids. This is, however, not always convenient, and in many cases there exist symbolic ids that correspond to the vertices. To allow this more flexible identification of vertices, one can assign a vertex attribute called 'name' to an igraph graph. After doing this, the symbolic vertex names can be used in all igraph functions, instead of the numeric ids.

Note that the uniqueness of vertex names are currently not enforced in igraph, you have to check that for yourself, when assigning the vertex names.

Value

A logical scalar.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

Examples

```
g <- make_ring(10)
is_named(g)
V(g)$name <- letters[1:10]
is_named(g)
neighbors(g, "a")
```

---

is_printer_callback *Is this a printer callback?*

Description

Is this a printer callback?

Usage

```
is_printer_callback(x)
```

Arguments

x	An R object.

See Also

Other printer callbacks: printer_callback()

`is_separator` *Check whether removing this set of vertices would disconnect the graph.*

Description

`is_separator()` determines whether the supplied vertex set is a vertex separator: A vertex set S is a separator if there are vertices u and v in the graph such that all paths between u and v pass through some vertices in S.

Usage

```
is_separator(graph, candidate)
```

Arguments

`graph`	The input graph. It may be directed, but edge directions are ignored.
`candidate`	A numeric vector giving the vertex ids of the candidate separator.

Value

A logical scalar, whether the supplied vertex set is a (minimal) vertex separator or not. lists all vertex separator of minimum size.

Related documentation in the C library

`is_separator()`, `vcount()`

See Also

Other flow: `dominator_tree()`, `edge_connectivity()`, `is_min_separator()`, `max_flow()`, `min_cut()`, `min_separators()`, `min_st_separators()`, `st_cuts()`, `st_min_cuts()`, `vertex_connectivity()`

Examples

```
ring <- make_ring(4)
min_st_separators(ring)
is_separator(ring, 1)
is_separator(ring, c(1, 3))
is_separator(ring, c(2, 4))
is_separator(ring, c(2, 3))
```

is_tree *Decide whether a graph is a tree.*

Description

is_tree() decides whether a graph is a tree, and optionally returns a possible root vertex if the graph is a tree.

Usage

```
is_tree(graph, mode = c("out", "in", "all", "total"), details = FALSE)
```

Arguments

graph	An igraph graph object
mode	Whether to consider edge directions in a directed graph. 'all' ignores edge directions; 'out' requires edges to be oriented outwards from the root, 'in' requires edges to be oriented towards the root.
details	Whether to return only whether the graph is a tree (FALSE) or also a possible root (TRUE)

Details

An undirected graph is a tree if it is connected and has no cycles. In the directed case, a possible additional requirement is that all edges are oriented away from a root (out-tree or arborescence) or all edges are oriented towards a root (in-tree or anti-arborescence). This test can be controlled using the mode parameter.

By convention, the null graph (i.e. the graph with no vertices) is considered not to be a tree.

Value

When details is FALSE, a logical value that indicates whether the graph is a tree. When details is TRUE, a named list with two entries:

res Logical value that indicates whether the graph is a tree.

root The root vertex of the tree; undefined if the graph is not a tree.

Related documentation in the C library

is_tree(), vcount()

See Also

Other trees: is_forest(), make_from_prufer(), sample_spanning_tree(), to_prufer()

Examples

```
g <- make_tree(7, 2)
is_tree(g)
is_tree(g, details = TRUE)
```

`is_weighted` *Weighted graphs*

Description

In weighted graphs, a real number is assigned to each (directed or undirected) edge.

Usage

```
is_weighted(graph)
```

Arguments

`graph`	The input graph.

Details

In igraph edge weights are represented via an edge attribute, called 'weight'. The `is_weighted()` function only checks that such an attribute exists. (It does not even checks that it is a numeric edge attribute.)

Edge weights are used for different purposes by the different functions. E.g. shortest path functions use it as the cost of the path; community finding methods use it as the strength of the relationship between two vertices, etc. Check the manual pages of the functions working with weighted graphs for details.

Value

A logical scalar.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

Examples

```
g <- make_ring(10)
shortest_paths(g, 8, 2)
E(g)$weight <- seq_len(ecount(g))
shortest_paths(g, 8, 2)
```

`isomorphic` *Decide if two graphs are isomorphic*

Description

Decide if two graphs are isomorphic

Usage

```
isomorphic(graph1, graph2, method = c("auto", "direct", "vf2", "bliss"), ...)

is_isomorphic_to(
  graph1,
  graph2,
  method = c("auto", "direct", "vf2", "bliss"),
  ...
)
```

Arguments

`graph1`	The first graph.
`graph2`	The second graph.
`method`	The method to use. Possible values: 'auto', 'direct', 'vf2', 'bliss'. See their details below.
`...`	Additional arguments, passed to the various methods.

Value

Logical scalar, `TRUE` if the graphs are isomorphic.

'auto' method

It tries to select the appropriate method based on the two graphs. This is the algorithm it uses:

1. If the two graphs do not agree on their order and size (i.e. number of vertices and edges), then return `FALSE`.
2. If the graphs have three or four vertices, then the 'direct' method is used.
3. If the graphs are directed, then the 'vf2' method is used.
4. Otherwise the 'bliss' method is used.

'direct' method

This method only works on graphs with three or four vertices, and it is based on a pre-calculated and stored table. It does not have any extra arguments.

'vf2' method

This method uses the VF2 algorithm by Cordella, Foggia et al., see references below. It supports vertex and edge colors and have the following extra arguments:

vertex.color1, vertex.color2 Optional integer vectors giving the colors of the vertices for colored graph isomorphism. If they are not given, but the graph has a "color" vertex attribute, then it will be used. If you want to ignore these attributes, then supply `NULL` for both of these arguments. See also examples below.

edge.color1, edge.color2 Optional integer vectors giving the colors of the edges for edge-colored (sub)graph isomorphism. If they are not given, but the graph has a "color" edge attribute, then it will be used. If you want to ignore these attributes, then supply `NULL` for both of these arguments.

'bliss' method

Uses the BLISS algorithm by Junttila and Kaski, and it works for undirected graphs. For both graphs the `canonical_permutation()` and then the `permute()` function is called to transfer them into canonical form; finally the canonical forms are compared. Extra arguments:

sh Character constant, the heuristics to use in the BLISS algorithm for `graph1` and `graph2`. See the `sh` argument of `canonical_permutation()` for possible values.

`sh` defaults to 'fm'.

Related documentation in the C library

`isomorphic()`, `isomorphic_bliss()`, `isomorphic_vf2()`, `vcount()`, `edges()`, `get_eids()`, `ecount()`

References

Tommi Junttila and Petteri Kaski: Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs, *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics.* 2007.

LP Cordella, P Foggia, C Sansone, and M Vento: An improved algorithm for matching large graphs, *Proc. of the 3rd IAPR TC-15 Workshop on Graphbased Representations in Pattern Recognition*, 149–159, 2001.

See Also

Other graph isomorphism: `canonical_permutation()`, `count_isomorphisms()`, `count_subgraph_isomorphisms()`, `graph_from_isomorphism_class()`, `isomorphism_class()`, `isomorphisms()`, `subgraph_isomorphic()`, `subgraph_isomorphisms()`

Examples

```
# create some non-isomorphic graphs
g1 <- graph_from_isomorphism_class(3, 10)
g2 <- graph_from_isomorphism_class(3, 11)
isomorphic(g1, g2)
```

```
# create two isomorphic graphs, by permuting the vertices of the first
g1 <- sample_pa(30, m = 2, directed = FALSE)
g2 <- permute(g1, sample(vcount(g1)))
# should be TRUE
isomorphic(g1, g2)
isomorphic(g1, g2, method = "bliss")
isomorphic(g1, g2, method = "vf2")

# colored graph isomorphism
g1 <- make_ring(10)
g2 <- make_ring(10)
isomorphic(g1, g2)

V(g1)$color <- rep(1:2, length = vcount(g1))
V(g2)$color <- rep(2:1, length = vcount(g2))
# consider colors by default
count_isomorphisms(g1, g2)
# ignore colors
count_isomorphisms(g1, g2,
  vertex.color1 = NULL,
  vertex.color2 = NULL
)
```

`isomorphism_class` *Isomorphism class of a graph*

Description

The isomorphism class is a non-negative integer number. Graphs (with the same number of vertices) having the same isomorphism class are isomorphic and isomorphic graphs always have the same isomorphism class. Currently it can handle directed graphs with 3 or 4 vertices and undirected graphs with 3 to 6 vertices.

Usage

```
isomorphism_class(graph, v)
```

Arguments

`graph`	The input graph.
`v`	Optionally a vertex sequence. If not missing, then an induced subgraph of the input graph, consisting of this vertices, is used.

Value

An integer number.

Related documentation in the C library

`isoclass_subgraph()`, `isoclass()`, `vcount()`

See Also

Other graph isomorphism: canonical_permutation(), count_isomorphisms(), count_subgraph_isomorphisms(), graph_from_isomorphism_class(), isomorphic(), isomorphisms(), subgraph_isomorphic(), subgraph_isomorphisms()

Examples

```
# create some non-isomorphic graphs
g1 <- graph_from_isomorphism_class(3, 10)
g2 <- graph_from_isomorphism_class(3, 11)
isomorphism_class(g1)
isomorphism_class(g2)
isomorphic(g1, g2)
```

---

isomorphisms *Calculate all isomorphic mappings between the vertices of two graphs*

Description

Calculate all isomorphic mappings between the vertices of two graphs

Usage

```
isomorphisms(graph1, graph2, method = "vf2", ..., callback = NULL)
```

Arguments

graph1	The first graph.
graph2	The second graph.
method	Currently only 'vf2' is supported, see isomorphic() for details about it and extra arguments.
...	Extra arguments, passed to the various methods.
callback	Optional callback function to call for each isomorphism found. If provided, the function should accept two arguments: map12 (integer vector mapping vertex IDs from graph1 to graph2, 1-based indexing) and map21 (integer vector mapping vertex IDs from graph2 to graph1, 1-based indexing). The function should return FALSE to continue the search or TRUE to stop it. If NULL (the default), all isomorphisms are collected and returned as a list. Only supported for method = "vf2". **Important limitation:** Callback functions must NOT call any igraph functions (including simple queries like vcount() or ecount()). Doing so will cause R to crash due to reentrancy issues. Extract any needed graph information before calling the function with a callback, or use collector mode (the default) and process results afterward.

Value

If `callback` is `NULL`, returns a list of vertex sequences, corresponding to all mappings from the first graph to the second. If `callback` is provided, returns `NULL` invisibly.

Related documentation in the C library

`get_isomorphisms_vf2_callback()`, `get_isomorphisms_vf2()`, `vcount()`, `edges()`, `get_eids()`, `ecount()`

See Also

Other graph isomorphism: `canonical_permutation()`, `count_isomorphisms()`, `count_subgraph_isomorphisms()`, `graph_from_isomorphism_class()`, `isomorphic()`, `isomorphism_class()`, `subgraph_isomorphic()`, `subgraph_isomorphisms()`

---

ivs *Independent vertex sets*

Description

A vertex set is called independent if there no edges between any two vertices in it. These functions find independent vertex sets in undirected graphs

Usage

```
ivs(graph, min = NULL, max = NULL)

largest_ivs(graph)

max_ivs(graph)

ivs_size(graph)

independence_number(graph)

is_ivs(graph, candidate)
```

Arguments

`graph`	The input graph.
`min`	Numeric constant, limit for the minimum size of the independent vertex sets to find. `NULL` means no limit.
`max`	Numeric constant, limit for the maximum size of the independent vertex sets to find. `NULL` means no limit.
`candidate`	The vertex set to test for being an independent set.

Details

`ivs()` finds all independent vertex sets in the network, obeying the size limitations given in the `min` and `max` arguments.

`largest_ivs()` finds the largest independent vertex sets in the graph. An independent vertex set is largest if there is no independent vertex set with more vertices.

`max_ivs()` finds the maximal independent vertex sets in the graph. An independent vertex set is maximal if it cannot be extended to a larger independent vertex set. The largest independent vertex sets are maximal, but the opposite is not always true.

`ivs_size()` calculate the size of the largest independent vertex set(s).

`independence_number()` is an alias for `ivs_size()`.

These functions use the algorithm described by Tsukiyama et al., see reference below.

`is_ivs()` tests if no pairs within a vertex set are connected.

Value

`ivs()`, `largest_ivs()` and `max_ivs()` return a list containing numeric vertex ids, each list element is an independent vertex set.

`ivs_size()` returns an integer constant.

`is_ivs()` returns `TRUE` if the candidate vertex set forms an independent set.

Related documentation in the C library

`vcount()`, `largest_independent_vertex_sets()`, `maximal_independent_vertex_sets()`, `independence_number()`, `is_independent_vertex_set()`

Author(s)

Tamas Nepusz <ntamas@gmail.com> ported it from the Very Nauty Graph Library by Keith Briggs (https://keithbriggs.info/) and Gabor Csardi <csardi.gabor@gmail.com> wrote the R interface and this manual page.

References

S. Tsukiyama, M. Ide, H. Ariyoshi and I. Shirawaka. A new algorithm for generating all the maximal independent sets. *SIAM J Computing*, 6:505–517, 1977.

See Also

Other cliques: `cliques()`, `is_complete()`, `weighted_cliques()`

Examples

```
# Do not run, takes a couple of seconds

# A quite dense graph
set.seed(42)
g <- sample_gnp(100, 0.9)
```

```
ivs_size(g)
ivs(g, min = ivs_size(g))
largest_ivs(g)
# Empty graph
induced_subgraph(g, largest_ivs(g)[[1]])

length(max_ivs(g))
```

`k_shortest_paths` *Find the k shortest paths between two vertices*

Description

Finds the k shortest paths between the given source and target vertex in order of increasing length. Currently this function uses Yen's algorithm.

Usage

```
k_shortest_paths(
  graph,
  from,
  to,
  k,
  ...,
  weights = NULL,
  mode = c("out", "in", "all", "total")
)
```

Arguments

`graph`	The input graph.
`from`	The source vertex of the shortest paths.
`to`	The target vertex of the shortest paths.
`k`	The number of paths to find. They will be returned in order of increasing length.
`...`	These dots are for future extensions and must be empty.
`weights`	Possibly a numeric vector giving edge weights. If this is `NULL` and the graph has a `weight` edge attribute, then the attribute is used. If this is `NA` then no weights are used (even if the graph has a `weight` attribute). In a weighted graph, the length of a path is the sum of the weights of its constituent edges.
`mode`	Character constant, gives whether the shortest paths to or from the given vertices should be calculated for directed graphs. If `out` then the shortest paths *from* the vertex, if `in` then *to* it will be considered. If `all`, the default, then the graph is treated as undirected, i.e. edge directions are not taken into account. This argument is ignored for undirected graphs.

Value

A named list with two components is returned:

vpaths The list of k shortest paths in terms of vertices

epaths The list of k shortest paths in terms of edges

Related documentation in the C library

`get_k_shortest_paths()`, `vcount()`, `edges()`, `get_eids()`, `ecount()`

References

Yen, Jin Y.: An algorithm for finding shortest routes from all source nodes to a given destination in general networks. Quarterly of Applied Mathematics. 27 (4): 526–530. (1970) doi:10.1090/qam/253822

See Also

`shortest_paths()`, `all_shortest_paths()`

Other structural.properties: `bfs()`, `component_distribution()`, `connect()`, `constraint()`, `coreness()`, `degree()`, `dfs()`, `distance_table()`, `edge_density()`, `feedback_arc_set()`, `feedback_vertex_set()`, `girth()`, `is_acyclic()`, `is_dag()`, `is_matching()`, `knn()`, `reciprocity()`, `subcomponent()`, `subgraph()`, `topo_sort()`, `transitivity()`, `unfold_tree()`, `which_multiple()`, `which_mutual()`

---

keeping_degseq *Graph rewiring while preserving the degree distribution*

Description

This function can be used together with `rewire()` to randomly rewire the edges while preserving the original graph's degree distribution.

Usage

```
keeping_degseq(loops = FALSE, niter = 100)
```

Arguments

`loops`	Whether to allow destroying and creating loop edges.
`niter`	Number of rewiring trials to perform.

Details

The rewiring algorithm chooses two arbitrary edges in each step ((a,b) and (c,d)) and substitutes them with (a,d) and (c,b), if they not already exists in the graph. The algorithm does not create multiple edges.

Author(s)

Tamas Nepusz <ntamas@gmail.com> and Gabor Csardi <csardi.gabor@gmail.com>

See Also

sample_degseq()

Other rewiring functions: each_edge(), rewire()

Examples

```
g <- make_ring(10)
g %>%
  rewire(keeping_degseq(niter = 20)) %>%
  degree()
print_all(rewire(g, with = keeping_degseq(niter = vcount(g) * 10)))
```

---

knn *Average nearest neighbor degree*

Description

Calculate the average nearest neighbor degree of the given vertices and the same quantity in the function of vertex degree

Usage

```
knn(
  graph,
  vids = V(graph),
  mode = c("all", "out", "in", "total"),
  neighbor.degree.mode = c("all", "out", "in", "total"),
  weights = NULL
)
```

Arguments

graph	The input graph. It may be directed.
vids	The vertices for which the calculation is performed. Normally it includes all vertices. Note, that if not all vertices are given here, then both 'knn' and 'knnk' will be calculated based on the given vertices only.
mode	Character constant to indicate the type of neighbors to consider in directed graphs. out considers out-neighbors, in considers in-neighbors and all ignores edge directions.
neighbor.degree.mode	The type of degree to average in directed graphs. out averages out-degrees, in averages in-degrees and all ignores edge directions for the degree calculation.

`weights` Weight vector. If the graph has a `weight` edge attribute, then this is used by default. If this argument is given, then vertex strength (see `strength()`) is used instead of vertex degree. But note that `knnk` is still given in the function of the normal vertex degree. Weights are are used to calculate a weighted degree (also called `strength()`) instead of the degree.

Details

Note that for zero degree vertices the answer in 'knn' is `NaN` (zero divided by zero), the same is true for 'knnk' if a given degree never appears in the network.

The weighted version computes a weighted average of the neighbor degrees as

$$k_{nn,u} = \frac{1}{s_u} \sum_v w_{uv} k_v,$$

where $s_u = \sum_v w_{uv}$ is the sum of the incident edge weights of vertex `u`, i.e. its strength. The sum runs over the neighbors `v` of vertex `u` as indicated by `mode`. w_{uv} denotes the weighted adjacency matrix and k_v is the neighbors' degree, specified by `neighbor_degree_mode`.

Value

A list with two members:

knn A numeric vector giving the average nearest neighbor degree for all vertices in `vids`.

knnk A numeric vector, its length is the maximum (total) vertex degree in the graph. The first element is the average nearest neighbor degree of vertices with degree one, etc.

Related documentation in the C library

`avg_nearest_neighbor_degree()`, `vcount()`, `edges()`, `get_eids()`, `ecount()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Alain Barrat, Marc Barthelemy, Romualdo Pastor-Satorras, Alessandro Vespignani: The architecture of complex weighted networks, Proc. Natl. Acad. Sci. USA 101, 3747 (2004)

See Also

Other structural.properties: `bfs()`, `component_distribution()`, `connect()`, `constraint()`, `coreness()`, `degree()`, `dfs()`, `distance_table()`, `edge_density()`, `feedback_arc_set()`, `feedback_vertex_set()`, `girth()`, `is_acyclic()`, `is_dag()`, `is_matching()`, `k_shortest_paths()`, `reciprocity()`, `subcomponent()`, `subgraph()`, `topo_sort()`, `transitivity()`, `unfold_tree()`, `which_multiple()`, `which_mutual()`

Examples

```
# Some trivial ones
g <- make_ring(10)
knn(g)
g2 <- make_star(10)
knn(g2)

# A scale-free one, try to plot 'knnk'
g3 <- sample_pa(1000, m = 5)
knn(g3)

# A random graph
g4 <- sample_gnp(1000, p = 5 / 1000)
knn(g4)

# A weighted graph
g5 <- make_star(10)
E(g5)$weight <- seq(ecount(g5))
knn(g5)
```

`laplacian_matrix` *Graph Laplacian*

Description

The Laplacian of a graph.

Usage

```
laplacian_matrix(
  graph,
  weights = NULL,
  sparse = igraph_opt("sparsematrices"),
  normalization = c("unnormalized", "symmetric", "left", "right"),
  normalized
)
```

Arguments

`graph`	The input graph.
`weights`	An optional vector giving edge weights for weighted Laplacian matrix. If this is `NULL` and the graph has an edge attribute called `weight`, then it will be used automatically. Set this to `NA` if you want the unweighted Laplacian on a graph that has a `weight` edge attribute.
`sparse`	Logical scalar, whether to return the result as a sparse matrix. The `Matrix` package is required for sparse matrices.
`normalization`	The normalization method to use when calculating the Laplacian matrix. See the "Normalization methods" section on this page.
`normalized`	Deprecated, use `normalization` instead.

Details

The Laplacian Matrix of a graph is a symmetric matrix having the same number of rows and columns as the number of vertices in the graph and element (i,j) is d[i], the degree of vertex i if if i==j, -1 if i!=j and there is an edge between vertices i and j and 0 otherwise.

The Laplacian matrix can also be normalized, with several conventional normalization methods. See the "Normalization methods" section on this page.

The weighted version of the Laplacian simply works with the weighted degree instead of the plain degree. I.e. (i,j) is d[i], the weighted degree of vertex i if if i==j, -w if i!=j and there is an edge between vertices i and j with weight w, and 0 otherwise. The weighted degree of a vertex is the sum of the weights of its adjacent edges.

Value

A numeric matrix.

Normalization methods

The Laplacian matrix L is defined in terms of the adjacency matrix A and a diagonal matrix D containing the degrees as follows:

- "unnormalized": Unnormalized Laplacian, $L = D - A$.
- "symmetric": Symmetrically normalized Laplacian, $L = I - D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$.
- "left": Left-stochastic normalized Laplacian, $L = I - D^{-1}A$.
- "right": Right-stochastic normalized Laplacian, $L = I - AD^{-1}$.

Related documentation in the C library

get_laplacian(), get_laplacian_sparse(), is_directed(), vcount(), edges(), get_eids(), ecount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

Examples

```
g <- make_ring(10)
laplacian_matrix(g)
laplacian_matrix(g, normalization = "unnormalized")
laplacian_matrix(g, normalization = "unnormalized", sparse = FALSE)
```

layout_ *Graph layouts*

Description

This is a generic function to apply a layout function to a graph.

Usage

```
layout_(graph, layout, ...)

## S3 method for class 'igraph_layout_spec'
print(x, ...)

## S3 method for class 'igraph_layout_modifier'
print(x, ...)
```

Arguments

graph	The input graph.
layout	The layout specification. It must be a call to a layout specification function.
...	Further modifiers, see a complete list below. For the print() methods, it is ignored.
x	The layout specification

Details

There are two ways to calculate graph layouts in igraph. The first way is to call a layout function (they all have prefix layout_() on a graph, to get the vertex coordinates.

The second way (new in igraph 0.8.0), has two steps, and it is more flexible. First you call a layout specification function (the one without the layout_() prefix, and then layout_() (or add_layout_()) to perform the layouting.

The second way is preferred, as it is more flexible. It allows operations before and after the layouting. E.g. using the component_wise() argument, the layout can be calculated separately for each component, and then merged to get the final results.

Value

The return value of the layout function, usually a two column matrix. For 3D layouts a three column matrix.

Modifiers

Modifiers modify how a layout calculation is performed. Modifiers are applied in the order they are specified as arguments to `layout_()`.

There are two types of modifiers:

- **Pre-layout modifiers** affect how the layout is calculated. Only one pre-layout modifier can be used at a time.
- **Post-layout modifiers** transform the resulting coordinates. Multiple post-layout modifiers can be chained together.

Currently implemented modifiers:

- `component_wise()` (pre-layout) calculates the layout separately for each component of the graph, and then merges them.
- `normalize()` (post-layout) scales the layout to a square.

Custom modifiers can be created using the `layout_modifier()` function. A custom modifier must specify:

- `id`: A unique identifier string for the modifier
- `type`: Either `"pre"` for pre-layout or `"post"` for post-layout
- `args`: A list of arguments to pass to the apply function
- `apply`: A function with signature `function(graph, layout, modifier_args)` that performs the modification. For pre-layout modifiers, `layout` is the layout specification. For post-layout modifiers, `layout` is the coordinate matrix to transform.

See Also

`add_layout_()` to add the layout to the graph as an attribute.

Other graph layouts: `add_layout_()`, `component_wise()`, `layout_as_bipartite()`, `layout_as_star()`, `layout_as_tree()`, `layout_in_circle()`, `layout_nicely()`, `layout_on_grid()`, `layout_on_sphere()`, `layout_randomly()`, `layout_with_dh()`, `layout_with_fr()`, `layout_with_gem()`, `layout_with_graphopt()`, `layout_with_kk()`, `layout_with_lgl()`, `layout_with_mds()`, `layout_with_sugiyama()`, `merge_coords()`, `norm_coords()`, `normalize()`

Examples

```
g <- make_ring(10) + make_full_graph(5)
coords <- layout_(g, as_star())
plot(g, layout = coords)

# Using modifiers
g <- make_ring(10) + make_ring(5)
coords <- layout_(g, in_circle(), component_wise(), normalize())
plot(g, layout = coords)

# Creating a custom post-layout modifier
scale_by <- function(factor) {
  layout_modifier(
```

```
    id = "scale_by",
    type = "post",
    args = list(factor = factor),
    apply = function(graph, layout, modifier_args) {
      layout * modifier_args$factor
    }
  )
}
coords <- layout_(make_ring(10), in_circle(), scale_by(3))
plot(make_ring(10), layout = coords)
```

`layout_as_bipartite` *Simple two-row layout for bipartite graphs*

Description

Minimize edge-crossings in a simple two-row (or column) layout for bipartite graphs.

Usage

```
layout_as_bipartite(graph, types = NULL, hgap = 1, vgap = 1, maxiter = 100)

as_bipartite(...)
```

Arguments

`graph`	The bipartite input graph. It should have a logical 'type' vertex attribute, or the `types` argument must be given.
`types`	A logical vector, the vertex types. If this argument is `NULL` (the default), then the 'type' vertex attribute is used.
`hgap`	Real scalar, the minimum horizontal gap between vertices in the same layer.
`vgap`	Real scalar, the distance between the two layers.
`maxiter`	Integer scalar, the maximum number of iterations in the crossing minimization stage. 100 is a reasonable default; if you feel that you have too many edge crossings, increase this.
`...`	Arguments to pass to `layout_as_bipartite()`.

Details

The layout is created by first placing the vertices in two rows, according to their types. Then the positions within the rows are optimized to minimize edge crossings, using the Sugiyama algorithm (see `layout_with_sugiyama()`).

Value

A matrix with two columns and as many rows as the number of vertices in the input graph.

Related documentation in the C library

layout_bipartite(), vcount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

layout_with_sugiyama()

Other graph layouts: add_layout_(), component_wise(), layout_(), layout_as_star(), layout_as_tree(), layout_in_circle(), layout_nicely(), layout_on_grid(), layout_on_sphere(), layout_randomly(), layout_with_dh(), layout_with_fr(), layout_with_gem(), layout_with_graphopt(), layout_with_kk(), layout_with_lgl(), layout_with_mds(), layout_with_sugiyama(), merge_coords(), norm_coords(), normalize()

Examples

```
# Random bipartite graph
inc <- matrix(sample(0:1, 50, replace = TRUE, prob = c(2, 1)), 10, 5)
g <- graph_from_biadjacency_matrix(inc)
plot(g,
  layout = layout_as_bipartite,
  vertex.color = c("green", "cyan")[V(g)$type + 1]
)

# Two columns
g %>%
  add_layout_(as_bipartite()) %>%
  plot()
```

---

layout_as_star *Generate coordinates to place the vertices of a graph in a star-shape*

Description

A simple layout generator, that places one vertex in the center of a circle and the rest of the vertices equidistantly on the perimeter.

Usage

```
layout_as_star(graph, center = V(graph)[1], order = NULL)

as_star(...)
```

Arguments

graph	The graph to layout.
center	The id of the vertex to put in the center. By default it is the first vertex.
order	Numeric vector, the order of the vertices along the perimeter. The default ordering is given by the vertex ids.
...	Arguments to pass to layout_as_star().

Details

It is possible to choose the vertex that will be in the center, and the order of the vertices can be also given.

Value

A matrix with two columns and as many rows as the number of vertices in the input graph.

Related documentation in the C library

layout_star(), vcount(), layout_circle()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

layout() and layout_with_drl() for other layout algorithms, plot.igraph() and tkplot() on how to plot graphs and star() on how to create ring graphs.

Other graph layouts: add_layout_(), component_wise(), layout_(), layout_as_bipartite(), layout_as_tree(), layout_in_circle(), layout_nicely(), layout_on_grid(), layout_on_sphere(), layout_randomly(), layout_with_dh(), layout_with_fr(), layout_with_gem(), layout_with_graphopt(), layout_with_kk(), layout_with_lgl(), layout_with_mds(), layout_with_sugiyama(), merge_coords(), norm_coords(), normalize()

Examples

```
g <- make_star(10)
layout_as_star(g)

## Alternative form
layout_(g, as_star())
```

layout_as_tree *The Reingold-Tilford graph layout algorithm*

Description

A tree-like layout, it is perfect for trees, acceptable for graphs with not too many cycles.

Usage

```
layout_as_tree(
  graph,
  root = numeric(),
  circular = FALSE,
  rootlevel = numeric(),
  mode = c("out", "in", "all"),
  flip.y = TRUE
)

as_tree(...)
```

Arguments

graph	The input graph.
root	The index of the root vertex or root vertices. If this is a non-empty vector then the supplied vertex ids are used as the roots of the trees (or a single tree if the graph is connected). If it is an empty vector, then the root vertices are automatically calculated based on topological sorting, performed with the opposite mode than the mode argument. After the vertices have been sorted, one is selected from each component.
circular	Logical scalar, whether to plot the tree in a circular fashion. Defaults to FALSE, so the tree branches are going bottom-up (or top-down, see the flip.y argument.
rootlevel	This argument can be useful when drawing forests which are not trees (i.e. they are unconnected and have tree components). It specifies the level of the root vertices for every tree in the forest. It is only considered if the roots argument is not an empty vector.
mode	Specifies which edges to consider when building the tree. If it is 'out', then only the outgoing, if it is 'in', then only the incoming edges of a parent are considered. If it is 'all' then all edges are used (this was the behavior in igraph 0.5 and before). This parameter also influences how the root vertices are calculated, if they are not given. See the roots parameter.
flip.y	Logical scalar, whether to flip the 'y' coordinates. The default is flipping because that puts the root vertex on the top.
...	Passed to layout_as_tree().

Details

Arranges the nodes in a tree where the given node is used as the root. The tree is directed downwards and the parents are centered above its children. For the exact algorithm, the reference below.

If the given graph is not a tree, a breadth-first search is executed first to obtain a possible spanning tree.

Value

A numeric matrix with two columns, and one row for each vertex.

Related documentation in the C library

vcount()

Author(s)

Tamas Nepusz <ntamas@gmail.com> and Gabor Csardi <csardi.gabor@gmail.com>

References

Reingold, E and Tilford, J (1981). Tidier drawing of trees. *IEEE Trans. on Softw. Eng.*, SE-7(2):223–228.

See Also

Other graph layouts: add_layout_(), component_wise(), layout_(), layout_as_bipartite(), layout_as_star(), layout_in_circle(), layout_nicely(), layout_on_grid(), layout_on_sphere(), layout_randomly(), layout_with_dh(), layout_with_fr(), layout_with_gem(), layout_with_graphopt(), layout_with_kk(), layout_with_lgl(), layout_with_mds(), layout_with_sugiyama(), merge_coords(), norm_coords(), normalize()

Examples

```
tree <- make_tree(20, 3)
plot(tree, layout = layout_as_tree)
plot(tree, layout = layout_as_tree(tree, flip.y = FALSE))
plot(tree, layout = layout_as_tree(tree, circular = TRUE))

tree2 <- make_tree(10, 3) + make_tree(10, 2)
plot(tree2, layout = layout_as_tree)
plot(tree2, layout = layout_as_tree(tree2,
  root = c(1, 11),
  rootlevel = c(2, 1)
))
```

layout_in_circle *Graph layout with vertices on a circle.*

Description

Place vertices on a circle, in the order of their vertex ids.

Usage

```
layout_in_circle(graph, order = V(graph))

in_circle(...)
```

Arguments

graph	The input graph.
order	The vertices to place on the circle, in the order of their desired placement. Vertices that are not included here will be placed at (0,0).
...	Passed to layout_in_circle().

Details

If you want to order the vertices differently, then permute them using the permute() function.

Value

A numeric matrix with two columns, and one row for each vertex.

Related documentation in the C library

layout_circle(), vcount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Other graph layouts: add_layout_(), component_wise(), layout_(), layout_as_bipartite(), layout_as_star(), layout_as_tree(), layout_nicely(), layout_on_grid(), layout_on_sphere(), layout_randomly(), layout_with_dh(), layout_with_fr(), layout_with_gem(), layout_with_graphopt(), layout_with_kk(), layout_with_lgl(), layout_with_mds(), layout_with_sugiyama(), merge_coords(), norm_coords(), normalize()

Examples

```
## Place vertices on a circle, order them according to their
## community
library(igraphdata)
data(karate)
karate_groups <- cluster_optimal(karate)
coords <- layout_in_circle(karate,
  order =
    order(membership(karate_groups))
)
V(karate)$label <- sub("Actor ", "", V(karate)$name)
V(karate)$label.color <- membership(karate_groups)
V(karate)$shape <- "none"
plot(karate, layout = coords)
```

`layout_modifier` *Create a layout modifier*

Description

This is a constructor function for creating custom layout modifiers. Layout modifiers can be used with `layout_()` to modify how layouts are calculated or to transform the resulting coordinates.

Usage

```
layout_modifier(...)
```

Arguments

`...`	Named arguments that define the modifier. Must include: **id** A unique identifier string for the modifier **type** Either `"pre"` for pre-layout or `"post"` for post-layout **args** A list of arguments to pass to the apply function **apply** A function with signature `function(graph, layout, modifier_args)` that performs the modification

Value

An object of class `igraph_layout_modifier`.

See Also

`layout_()` for using modifiers, `component_wise()`, `normalize()` for examples of built-in modifiers.

Other layout modifiers: `component_wise()`, `normalize()`

Examples

```
# Create a custom post-layout modifier that scales coordinates
scale_by <- function(factor) {
  layout_modifier(
    id = "scale_by",
    type = "post",
    args = list(factor = factor),
    apply = function(graph, layout, modifier_args) {
      layout * modifier_args$factor
    }
  )
}

# Use the custom modifier
g <- make_ring(10)
coords <- layout_(g, in_circle(), scale_by(2))
plot(g, layout = coords)
```

`layout_nicely` *Choose an appropriate graph layout algorithm automatically*

Description

This function tries to choose an appropriate graph layout algorithm for the graph, automatically, based on a simple algorithm. See details below.

Usage

```
layout_nicely(graph, dim = 2, ...)

nicely(...)
```

Arguments

`graph`	The input graph
`dim`	Dimensions, should be 2 or 3.
`...`	For `layout_nicely()` the extra arguments are passed to the real layout function. For `nicely()` all argument are passed to `layout_nicely()`.

Details

`layout_nicely()` tries to choose an appropriate layout function for the supplied graph, and uses that to generate the layout. The current implementation works like this:

1. If the graph has a graph attribute called 'layout', then this is used. If this attribute is an R function, then it is called, with the graph and any other extra arguments.
2. Otherwise, if the graph has vertex attributes called 'x' and 'y', then these are used as coordinates. If the graph has an additional 'z' vertex attribute, that is also used.

3. Otherwise, if the graph is a forest and has less than 30 vertices, `layout_as_tree()` is used.
4. Otherwise, if the graph is connected and has less than 1000 vertices, the Fruchterman-Reingold layout is used, by calling `layout_with_fr()`.
5. Otherwise the DrL layout is used, `layout_with_drl()` is called.

In layout algorithm implementations, an argument named 'weights' is typically used to specify the weights of the edges if the layout algorithm supports them. In this case, omitting 'weights' or setting it to `NULL` will make igraph use the 'weight' edge attribute from the graph if it is present. However, most layout algorithms do not support non-positive weights, so `layout_nicely()` would fail if you simply called it on your graph without specifying explicit weights and the weights happened to include non-positive numbers. We strive to ensure that `layout_nicely()` works out-of-the-box for most graphs, so the rule is that if you omit 'weights' or set it to `NULL` and `layout_nicely()` would end up calling `layout_with_fr()` or `layout_with_drl()`, we do not forward the weights to these functions and issue a warning about this. You can use `weights = NA` to silence the warning.

Value

A numeric matrix with two or three columns.

Related documentation in the C library

`vcount()`, `layout_align()`, `is_forest()`, `edges()`, `get_eids()`, `ecount()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

`plot.igraph()`

Other graph layouts: `add_layout_()`, `component_wise()`, `layout_()`, `layout_as_bipartite()`, `layout_as_star()`, `layout_as_tree()`, `layout_in_circle()`, `layout_on_grid()`, `layout_on_sphere()`, `layout_randomly()`, `layout_with_dh()`, `layout_with_fr()`, `layout_with_gem()`, `layout_with_graphopt()`, `layout_with_kk()`, `layout_with_lgl()`, `layout_with_mds()`, `layout_with_sugiyama()`, `merge_coords()`, `norm_coords()`, `normalize()`

---

layout_on_grid *Simple grid layout*

Description

This layout places vertices on a rectangular grid, in two or three dimensions.

Usage

```
layout_on_grid(graph, width = 0, height = 0, dim = 2)

on_grid(...)
```

Arguments

graph	The input graph.
width	The number of vertices in a single row of the grid. If this is zero or negative, then for 2d layouts the width of the grid will be the square root of the number of vertices in the graph, rounded up to the next integer. Similarly, it will be the cube root for 3d layouts.
height	The number of vertices in a single column of the grid, for three dimensional layouts. If this is zero or negative, then it is determinted automatically.
dim	Two or three. Whether to make 2d or a 3d layout.
...	Passed to layout_on_grid().

Details

The function places the vertices on a simple rectangular grid, one after the other. If you want to change the order of the vertices, then see the permute() function.

Value

A two-column or three-column matrix.

Related documentation in the C library

layout_grid(), layout_grid_3d()

Author(s)

Tamas Nepusz <ntamas@gmail.com>

See Also

layout() for other layout generators

Other graph layouts: add_layout_(), component_wise(), layout_(), layout_as_bipartite(), layout_as_star(), layout_as_tree(), layout_in_circle(), layout_nicely(), layout_on_sphere(), layout_randomly(), layout_with_dh(), layout_with_fr(), layout_with_gem(), layout_with_graphopt(), layout_with_kk(), layout_with_lgl(), layout_with_mds(), layout_with_sugiyama(), merge_coords(), norm_coords(), normalize()

Examples

```
g <- make_lattice(c(3, 3))
layout_on_grid(g)

g2 <- make_lattice(c(3, 3, 3))
layout_on_grid(g2, dim = 3)

plot(g, layout = layout_on_grid)
if (interactive() && requireNamespace("rgl", quietly = TRUE)) {
  rglplot(g, layout = layout_on_grid(g, dim = 3))
}
```

layout_on_sphere *Graph layout with vertices on the surface of a sphere*

Description

Place vertices on a sphere, approximately uniformly, in the order of their vertex ids.

Usage

```
layout_on_sphere(graph)

on_sphere(...)
```

Arguments

graph	The input graph.
...	Passed to layout_on_sphere().

Details

layout_on_sphere() places the vertices (approximately) uniformly on the surface of a sphere, this is thus a 3d layout. It is not clear however what "uniformly on a sphere" means.

If you want to order the vertices differently, then permute them using the permute() function.

Value

A numeric matrix with three columns, and one row for each vertex.

Related documentation in the C library

layout_sphere()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Other graph layouts: add_layout_(), component_wise(), layout_(), layout_as_bipartite(), layout_as_star(), layout_as_tree(), layout_in_circle(), layout_nicely(), layout_on_grid(), layout_randomly(), layout_with_dh(), layout_with_fr(), layout_with_gem(), layout_with_graphopt(), layout_with_kk(), layout_with_lgl(), layout_with_mds(), layout_with_sugiyama(), merge_coords(), norm_coords(), normalize()

`layout_randomly` *Randomly place vertices on a plane or in 3d space*

Description

This function uniformly randomly places the vertices of the graph in two or three dimensions.

Usage

```
layout_randomly(graph, dim = c(2, 3))

randomly(...)
```

Arguments

`graph`	The input graph.
`dim`	Integer scalar, the dimension of the space to use. It must be 2 or 3.
`...`	Parameters to pass to `layout_randomly()`.

Details

Randomly places vertices on a [-1,1] square (in 2d) or in a cube (in 3d). It is probably a useless layout, but it can use as a starting point for other layout generators.

Value

A numeric matrix with two or three columns.

Related documentation in the C library

`layout_random()`, `layout_random_3d()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Other graph layouts: `add_layout_()`, `component_wise()`, `layout_()`, `layout_as_bipartite()`, `layout_as_star()`, `layout_as_tree()`, `layout_in_circle()`, `layout_nicely()`, `layout_on_grid()`, `layout_on_sphere()`, `layout_with_dh()`, `layout_with_fr()`, `layout_with_gem()`, `layout_with_graphopt()`, `layout_with_kk()`, `layout_with_lgl()`, `layout_with_mds()`, `layout_with_sugiyama()`, `merge_coords()`, `norm_coords()`, `normalize()`

layout_with_dh *The Davidson-Harel layout algorithm*

Description

Place vertices of a graph on the plane, according to the simulated annealing algorithm by Davidson and Harel.

Usage

```
layout_with_dh(
  graph,
  coords = NULL,
  maxiter = 10,
  fineiter = max(10, log2(vcount(graph))),
  cool.fact = 0.75,
  weight.node.dist = 1,
  weight.border = 0,
  weight.edge.lengths = edge_density(graph)/10,
  weight.edge.crossings = 1 - sqrt(edge_density(graph)),
  weight.node.edge.dist = 0.2 * (1 - edge_density(graph))
)

with_dh(...)
```

Arguments

graph	The graph to lay out. Edge directions are ignored.
coords	Optional starting positions for the vertices. If this argument is not NULL then it should be an appropriate matrix of starting coordinates.
maxiter	Number of iterations to perform in the first phase.
fineiter	Number of iterations in the fine tuning phase.
cool.fact	Cooling factor.
weight.node.dist	Weight for the node-node distances component of the energy function.
weight.border	Weight for the distance from the border component of the energy function. It can be set to zero, if vertices are allowed to sit on the border.
weight.edge.lengths	Weight for the edge length component of the energy function.
weight.edge.crossings	Weight for the edge crossing component of the energy function.
weight.node.edge.dist	Weight for the node-edge distance component of the energy function.
...	Passed to layout_with_dh().

Details

This function implements the algorithm by Davidson and Harel, see Ron Davidson, David Harel: Drawing Graphs Nicely Using Simulated Annealing. ACM Transactions on Graphics 15(4), pp. 301-331, 1996.

The algorithm uses simulated annealing and a sophisticated energy function, which is unfortunately hard to parameterize for different graphs. The original publication did not disclose any parameter values, and the ones below were determined by experimentation.

The algorithm consists of two phases, an annealing phase, and a fine-tuning phase. There is no simulated annealing in the second phase.

Our implementation tries to follow the original publication, as much as possible. The only major difference is that coordinates are explicitly kept within the bounds of the rectangle of the layout.

Value

A matrix with two columns, containing the x and y coordinates of the vertices:

x The x-coordinate of the vertex.

y The y-coordinate of the vertex.

Related documentation in the C library

`layout_davidson_harel()`, `vcount()`, `density()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Ron Davidson, David Harel: Drawing Graphs Nicely Using Simulated Annealing. *ACM Transactions on Graphics* 15(4), pp. 301-331, 1996.

See Also

`layout_with_fr()`, `layout_with_kk()` for other layout algorithms.

Other graph layouts: `add_layout_()`, `component_wise()`, `layout_()`, `layout_as_bipartite()`, `layout_as_star()`, `layout_as_tree()`, `layout_in_circle()`, `layout_nicely()`, `layout_on_grid()`, `layout_on_sphere()`, `layout_randomly()`, `layout_with_fr()`, `layout_with_gem()`, `layout_with_graphopt()`, `layout_with_kk()`, `layout_with_lgl()`, `layout_with_mds()`, `layout_with_sugiyama()`, `merge_coords()`, `norm_coords()`, `normalize()`

Examples

```
set.seed(42)
## Figures from the paper
g_1b <- make_star(19, mode = "undirected") + path(c(2:19, 2)) +
  path(c(seq(2, 18, by = 2), 2))
plot(g_1b, layout = layout_with_dh)
```

```
g_2 <- make_lattice(c(8, 3)) + edges(1, 8, 9, 16, 17, 24)
plot(g_2, layout = layout_with_dh)

g_3 <- make_empty_graph(n = 70)
plot(g_3, layout = layout_with_dh)

g_4 <- make_empty_graph(n = 70, directed = FALSE) + edges(1:70)
plot(g_4, layout = layout_with_dh, vertex.size = 5, vertex.label = NA)

g_5a <- make_ring(24)
plot(g_5a, layout = layout_with_dh, vertex.size = 5, vertex.label = NA)

g_5b <- make_ring(40)
plot(g_5b, layout = layout_with_dh, vertex.size = 5, vertex.label = NA)

g_6 <- make_lattice(c(2, 2, 2))
plot(g_6, layout = layout_with_dh)

g_7 <- graph_from_literal(1:3:5 - -2:4:6)
plot(g_7, layout = layout_with_dh, vertex.label = V(g_7)$name)

g_8 <- make_ring(5) + make_ring(10) + make_ring(5) +
  edges(
    1, 6, 2, 8, 3, 10, 4, 12, 5, 14,
    7, 16, 9, 17, 11, 18, 13, 19, 15, 20
  )
plot(g_8, layout = layout_with_dh, vertex.size = 5, vertex.label = NA)

g_9 <- make_lattice(c(3, 2, 2))
plot(g_9, layout = layout_with_dh, vertex.size = 5, vertex.label = NA)

g_10 <- make_lattice(c(6, 6))
plot(g_10, layout = layout_with_dh, vertex.size = 5, vertex.label = NA)

g_11a <- make_tree(31, 2, mode = "undirected")
plot(g_11a, layout = layout_with_dh, vertex.size = 5, vertex.label = NA)

g_11b <- make_tree(21, 4, mode = "undirected")
plot(g_11b, layout = layout_with_dh, vertex.size = 5, vertex.label = NA)

g_12 <- make_empty_graph(n = 37, directed = FALSE) +
  path(1:5, 10, 22, 31, 37:33, 27, 16, 6, 1) + path(6, 7, 11, 9, 10) + path(16:22) +
  path(27:31) + path(2, 7, 18, 28, 34) + path(3, 8, 11, 19, 29, 32, 35) +
  path(4, 9, 20, 30, 36) + path(1, 7, 12, 14, 19, 24, 26, 30, 37) +
  path(5, 9, 13, 15, 19, 23, 25, 28, 33) + path(3, 12, 16, 25, 35, 26, 22, 13, 3)
plot(g_12, layout = layout_with_dh, vertex.size = 5, vertex.label = NA)
```

`layout_with_drl` *The DrL graph layout generator*

Description

DrL is a force-directed graph layout toolbox focused on real-world large-scale graphs, developed by Shawn Martin and colleagues at Sandia National Laboratories.

Usage

```
layout_with_drl(
  graph,
  use.seed = FALSE,
  seed = matrix(runif(vcount(graph) * 2), ncol = 2),
  options = drl_defaults$default,
  weights = NULL,
  dim = c(2, 3)
)

with_drl(...)
```

Arguments

graph	The input graph, in can be directed or undirected.
use.seed	Logical scalar, whether to use the coordinates given in the seed argument as a starting point.
seed	A matrix with two columns, the starting coordinates for the vertices is use.seed is TRUE. It is ignored otherwise.
options	Options for the layout generator, a named list. See details below.
weights	The weights of the edges. It must be a positive numeric vector, NULL or NA. If it is NULL and the input graph has a 'weight' edge attribute, then that attribute will be used. If NULL and no such attribute is present, then the edges will have equal weights. Set this to NA if the graph was a 'weight' edge attribute, but you don't want to use it for the layout. Larger edge weights correspond to stronger connections.
dim	Either '2' or '3', it specifies whether we want a two dimensional or a three dimensional layout. Note that because of the nature of the DrL algorithm, the three dimensional layout takes significantly longer to compute.
...	Passed to layout_with_drl().

Details

This function implements the force-directed DrL layout generator.

The generator has the following parameters:

edge.cut Edge cutting is done in the late stages of the algorithm in order to achieve less dense layouts. Edges are cut if there is a lot of stress on them (a large value in the objective function sum). The edge cutting parameter is a value between 0 and 1 with 0 representing no edge cutting and 1 representing maximal edge cutting.

init.iterations Number of iterations in the first phase.

init.temperature Start temperature, first phase.

init.attraction Attraction, first phase.

init.damping.mult Damping, first phase.

liquid.iterations Number of iterations, liquid phase.

liquid.temperature Start temperature, liquid phase.

liquid.attraction Attraction, liquid phase.

liquid.damping.mult Damping, liquid phase.

expansion.iterations Number of iterations, expansion phase.

expansion.temperature Start temperature, expansion phase.

expansion.attraction Attraction, expansion phase.

expansion.damping.mult Damping, expansion phase.

cooldown.iterations Number of iterations, cooldown phase.

cooldown.temperature Start temperature, cooldown phase.

cooldown.attraction Attraction, cooldown phase.

cooldown.damping.mult Damping, cooldown phase.

crunch.iterations Number of iterations, crunch phase.

crunch.temperature Start temperature, crunch phase.

crunch.attraction Attraction, crunch phase.

crunch.damping.mult Damping, crunch phase.

simmer.iterations Number of iterations, simmer phase.

simmer.temperature Start temperature, simmer phase.

simmer.attraction Attraction, simmer phase.

simmer.damping.mult Damping, simmer phase.

There are five pre-defined parameter settings as well, these are called `drl_defaults$default`, `drl_defaults$coarsen`, `drl_defaults$coarsest`, `drl_defaults$refine` and `drl_defaults$final`.

Value

A numeric matrix with two columns.

Related documentation in the C library

`vcount()`, `edges()`, `get_eids()`, `ecount()`

Author(s)

Shawn Martin (https://www.cs.otago.ac.nz/homepages/smartin/) and Gabor Csardi <csardi.gabor@gmail.com> for the R/igraph interface and the three dimensional version.

References

See the following technical report: Martin, S., Brown, W.M., Klavans, R., Boyack, K.W., DrL: Distributed Recursive (Graph) Layout. SAND Reports, 2008. 2936: p. 1-10.

See Also

layout() for other layout generators.

Examples

```
g <- as_undirected(sample_pa(100, m = 1))
l <- layout_with_drl(g, options = list(simmer.attraction = 0))
plot(g, layout = l, vertex.size = 3, vertex.label = NA)
```

---

layout_with_fr *The Fruchterman-Reingold layout algorithm*

Description

Place vertices on the plane using the force-directed layout algorithm by Fruchterman and Reingold.

Usage

```
layout_with_fr(
  graph,
  coords = NULL,
  dim = c(2, 3),
  niter = 500,
  start.temp = sqrt(vcount(graph)),
  grid = c("auto", "grid", "nogrid"),
  weights = NULL,
  minx = NULL,
  maxx = NULL,
  miny = NULL,
  maxy = NULL,
  minz = NULL,
  maxz = NULL,
  coolexp = deprecated(),
  maxdelta = deprecated(),
  area = deprecated(),
  repulserad = deprecated(),
  maxiter = deprecated()
)

with_fr(...)
```

Arguments

graph	The graph to lay out. Edge directions are ignored.
coords	Optional starting positions for the vertices. If this argument is not NULL then it should be an appropriate matrix of starting coordinates.

dim	Integer scalar, 2 or 3, the dimension of the layout. Two dimensional layouts are places on a plane, three dimensional ones in the 3d space.
niter	Integer scalar, the number of iterations to perform.
start.temp	Real scalar, the start temperature. This is the maximum amount of movement alloved along one axis, within one step, for a vertex. Currently it is decreased linearly to zero during the iteration.
grid	Character scalar, whether to use the faster, but less accurate grid based implementation of the algorithm. By default ("auto"), the grid-based implementation is used if the graph has more than one thousand vertices.
weights	A vector giving edge weights. The weight edge attribute is used by default, if present. If weights are given, then the attraction along the edges will be multiplied by the given edge weights. This places vertices connected with a highly weighted edge closer to each other. Weights must be positive.
minx	If not NULL, then it must be a numeric vector that gives lower boundaries for the 'x' coordinates of the vertices. The length of the vector must match the number of vertices in the graph.
maxx	Similar to minx, but gives the upper boundaries.
miny	Similar to minx, but gives the lower boundaries of the 'y' coordinates.
maxy	Similar to minx, but gives the upper boundaries of the 'y' coordinates.
minz	Similar to minx, but gives the lower boundaries of the 'z' coordinates.
maxz	Similar to minx, but gives the upper boundaries of the 'z' coordinates.
coolexp, maxdelta, area, repulserad	**[Deprecated]** These arguments are not supported from igraph version 0.8.0 and are ignored (with a warning).
maxiter	A deprecated synonym of niter, for compatibility.
...	Passed to layout_with_fr().

Details

See the referenced paper below for the details of the algorithm.

This function was rewritten from scratch in igraph version 0.8.0.

Value

A two- or three-column matrix, each row giving the coordinates of a vertex, according to the ids of the vertex ids.

Related documentation in the C library

vcount(), edges(), get_eids(), ecount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Fruchterman, T.M.J. and Reingold, E.M. (1991). Graph Drawing by Force-directed Placement. *Software - Practice and Experience*, 21(11):1129-1164.

See Also

layout_with_drl(), layout_with_kk() for other layout algorithms.

Other graph layouts: add_layout_(), component_wise(), layout_(), layout_as_bipartite(), layout_as_star(), layout_as_tree(), layout_in_circle(), layout_nicely(), layout_on_grid(), layout_on_sphere(), layout_randomly(), layout_with_dh(), layout_with_gem(), layout_with_graphopt(), layout_with_kk(), layout_with_lgl(), layout_with_mds(), layout_with_sugiyama(), merge_coords(), norm_coords(), normalize()

Examples

```
# Fixing ego
g <- sample_pa(20, m = 2)
minC <- rep(-Inf, vcount(g))
maxC <- rep(Inf, vcount(g))
minC[1] <- maxC[1] <- 0
co <- layout_with_fr(g,
  minx = minC, maxx = maxC,
  miny = minC, maxy = maxC
)
co[1, ]
plot(g,
  layout = co, vertex.size = 30, edge.arrow.size = 0.2,
  vertex.label = c("ego", rep("", vcount(g) - 1)), rescale = FALSE,
  xlim = range(co[, 1]), ylim = range(co[, 2]), vertex.label.dist = 0,
  vertex.label.color = "red"
)
axis(1)
axis(2)
```

---

layout_with_gem *The GEM layout algorithm*

Description

Place vertices on the plane using the GEM force-directed layout algorithm.

Usage

```
layout_with_gem(
  graph,
  coords = NULL,
  maxiter = 40 * vcount(graph)^2,
```

```
  temp.max = max(vcount(graph), 1),
  temp.min = 1/10,
  temp.init = sqrt(max(vcount(graph), 1))
)

with_gem(...)
```

Arguments

graph	The input graph. Edge directions are ignored.
coords	If not NULL, then the starting coordinates should be given here, in a two or three column matrix, depending on the dim argument.
maxiter	The maximum number of iterations to perform. Updating a single vertex counts as an iteration. A reasonable default is 40 * n * n, where n is the number of vertices. The original paper suggests 4 * n * n, but this usually only works if the other parameters are set up carefully.
temp.max	The maximum allowed local temperature. A reasonable default is the number of vertices.
temp.min	The global temperature at which the algorithm terminates (even before reaching maxiter iterations). A reasonable default is 1/10.
temp.init	Initial local temperature of all vertices. A reasonable default is the square root of the number of vertices.
...	Passed to layout_with_gem().

Details

See the referenced paper below for the details of the algorithm.

Value

A numeric matrix with two columns, and as many rows as the number of vertices.

Related documentation in the C library

layout_gem(), vcount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Arne Frick, Andreas Ludwig, Heiko Mehldau: A Fast Adaptive Layout Algorithm for Undirected Graphs, *Proc. Graph Drawing 1994*, LNCS 894, pp. 388-403, 1995.

See Also

layout_with_fr(), plot.igraph(), tkplot()

Other graph layouts: add_layout_(), component_wise(), layout_(), layout_as_bipartite(), layout_as_star(), layout_as_tree(), layout_in_circle(), layout_nicely(), layout_on_grid(), layout_on_sphere(), layout_randomly(), layout_with_dh(), layout_with_fr(), layout_with_graphopt(), layout_with_kk(), layout_with_lgl(), layout_with_mds(), layout_with_sugiyama(), merge_coords(), norm_coords(), normalize()

Examples

```
set.seed(42)
g <- make_ring(10)
plot(g, layout = layout_with_gem)
```

---

layout_with_graphopt *The graphopt layout algorithm*

Description

A force-directed layout algorithm, that scales relatively well to large graphs.

Usage

```
layout_with_graphopt(
  graph,
  start = NULL,
  niter = 500,
  charge = 0.001,
  mass = 30,
  spring.length = 0,
  spring.constant = 1,
  max.sa.movement = 5
)

with_graphopt(...)
```

Arguments

graph	The input graph.
start	If given, then it should be a matrix with two columns and one line for each vertex. This matrix will be used as starting positions for the algorithm. If not given, then a random starting matrix is used.
niter	Integer scalar, the number of iterations to perform. Should be a couple of hundred in general. If you have a large graph then you might want to only do a few iterations and then check the result. If it is not good enough you can feed it in again in the start argument. The default value is 500.

charge	The charge of the vertices, used to calculate electric repulsion. The default is 0.001.
mass	The mass of the vertices, used for the spring forces. The default is 30.
spring.length	The length of the springs, an integer number. The default value is zero.
spring.constant	The spring constant, the default value is one.
max.sa.movement	Real constant, it gives the maximum amount of movement allowed in a single step along a single axis. The default value is 5.
...	Passed to layout_with_graphopt().

Details

layout_with_graphopt() is a port of the graphopt layout algorithm by Michael Schmuhl. graphopt version 0.4.1 was rewritten in C and the support for layers was removed (might be added later) and a code was a bit reorganized to avoid some unnecessary steps is the node charge (see below) is zero.

graphopt uses physical analogies for defining attracting and repelling forces among the vertices and then the physical system is simulated until it reaches an equilibrium. (There is no simulated annealing or anything like that, so a stable fixed point is not guaranteed.)

Value

A numeric matrix with two columns, and a row for each vertex.

Author(s)

Michael Schmuhl for the original graphopt code, rewritten and wrapped by Gabor Csardi <csardi.gabor@gmail.com>.

See Also

Other graph layouts: add_layout_(), component_wise(), layout_(), layout_as_bipartite(), layout_as_star(), layout_as_tree(), layout_in_circle(), layout_nicely(), layout_on_grid(), layout_on_sphere(), layout_randomly(), layout_with_dh(), layout_with_fr(), layout_with_gem(), layout_with_kk(), layout_with_lgl(), layout_with_mds(), layout_with_sugiyama(), merge_coords(), norm_coords(), normalize()

---

layout_with_kk *The Kamada-Kawai layout algorithm*

Description

Place the vertices on the plane, or in 3D space, based on a physical model of springs.

Usage

```
layout_with_kk(
  graph,
  coords = NULL,
  dim = c(2, 3),
  maxiter = 50 * vcount(graph),
  epsilon = 0,
  kkconst = max(vcount(graph), 1),
  weights = NULL,
  minx = NULL,
  maxx = NULL,
  miny = NULL,
  maxy = NULL,
  minz = NULL,
  maxz = NULL,
  niter = deprecated(),
  sigma = deprecated(),
  initemp = deprecated(),
  coolexp = deprecated(),
  start = deprecated()
)

with_kk(...)
```

Arguments

graph	The input graph. Edge directions are ignored.
coords	If not NULL, then the starting coordinates should be given here, in a two or three column matrix, depending on the dim argument.
dim	Integer scalar, 2 or 3, the dimension of the layout. Two dimensional layouts are places on a plane, three dimensional ones in the 3d space.
maxiter	The maximum number of iterations to perform. The algorithm might terminate earlier, see the epsilon argument.
epsilon	Numeric scalar, the algorithm terminates, if the maximal delta is less than this. (See the reference below for what delta means.) If you set this to zero, then the function always performs maxiter iterations.
kkconst	Numeric scalar, the Kamada-Kawai vertex attraction constant. Typical (and default) value is the number of vertices.
weights	Edge weights, larger values will result in longer edges. Note that this is the opposite of layout_with_fr(), which produces shorter edges for larger weights. Weights must be positive.
minx	If not NULL, then it must be a numeric vector that gives lower boundaries for the 'x' coordinates of the vertices. The length of the vector must match the number of vertices in the graph.
maxx	Similar to minx, but gives the upper boundaries.
miny	Similar to minx, but gives the lower boundaries of the 'y' coordinates.

maxy	Similar to minx, but gives the upper boundaries of the 'y' coordinates.
minz	Similar to minx, but gives the lower boundaries of the 'z' coordinates.
maxz	Similar to minx, but gives the upper boundaries of the 'z' coordinates.
niter, sigma, initemp, coolexp	**[Deprecated]** These arguments are not supported from igraph version 0.8.0 and are ignored (with a warning).
start	Deprecated synonym for coords, for compatibility.
...	Passed to layout_with_kk().

Details

See the referenced paper below for the details of the algorithm.

This function was rewritten from scratch in igraph version 0.8.0 and it follows truthfully the original publication by Kamada and Kawai now.

Value

A numeric matrix with two (dim=2) or three (dim=3) columns, and as many rows as the number of vertices, the x, y and potentially z coordinates of the vertices.

Related documentation in the C library

vcount(), edges(), get_eids(), ecount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Kamada, T. and Kawai, S.: An Algorithm for Drawing General Undirected Graphs. *Information Processing Letters*, 31/1, 7–15, 1989.

See Also

layout_with_drl(), plot.igraph(), tkplot()

Other graph layouts: add_layout_(), component_wise(), layout_(), layout_as_bipartite(), layout_as_star(), layout_as_tree(), layout_in_circle(), layout_nicely(), layout_on_grid(), layout_on_sphere(), layout_randomly(), layout_with_dh(), layout_with_fr(), layout_with_gem(), layout_with_graphopt(), layout_with_lgl(), layout_with_mds(), layout_with_sugiyama(), merge_coords(), norm_coords(), normalize()

Examples

```
g <- make_ring(10)
E(g)$weight <- rep(1:2, length.out = ecount(g))
plot(g, layout = layout_with_kk, edge.label = E(g)$weight)
```

layout_with_lgl *Large Graph Layout*

Description

A layout generator for larger graphs.

Usage

```
layout_with_lgl(
  graph,
  maxiter = 150,
  maxdelta = vcount(graph),
  area = vcount(graph)^2,
  coolexp = 1.5,
  repulserad = area * vcount(graph),
  cellsize = sqrt(sqrt(area)),
  root = NULL
)

with_lgl(...)
```

Arguments

graph	The input graph
maxiter	The maximum number of iterations to perform (150).
maxdelta	The maximum change for a vertex during an iteration (the number of vertices).
area	The area of the surface on which the vertices are placed (square of the number of vertices).
coolexp	The cooling exponent of the simulated annealing (1.5).
repulserad	Cancellation radius for the repulsion (the area times the number of vertices).
cellsize	The size of the cells for the grid. When calculating the repulsion forces between vertices only vertices in the same or neighboring grid cells are taken into account (the fourth root of the number of area.
root	The id of the vertex to place at the middle of the layout. The default value is -1 which means that a random vertex is selected.
...	Passed to layout_with_lgl().

Details

layout_with_lgl() is for large connected graphs, it is similar to the layout generator of the Large Graph Layout software (https://lgl.sourceforge.net/).

Value

A numeric matrix with two columns and as many rows as vertices.

Related documentation in the C library

vcount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Other graph layouts: add_layout_(), component_wise(), layout_(), layout_as_bipartite(), layout_as_star(), layout_as_tree(), layout_in_circle(), layout_nicely(), layout_on_grid(), layout_on_sphere(), layout_randomly(), layout_with_dh(), layout_with_fr(), layout_with_gem(), layout_with_graphopt(), layout_with_kk(), layout_with_mds(), layout_with_sugiyama(), merge_coords(), norm_coords(), normalize()

---

layout_with_mds *Graph layout by multidimensional scaling*

Description

Multidimensional scaling of some distance matrix defined on the vertices of a graph.

Usage

```
layout_with_mds(graph, dist = NULL, dim = 2, options = arpack_defaults())

with_mds(...)
```

Arguments

graph	The input graph.
dist	The distance matrix for the multidimensional scaling. If NULL (the default), then the unweighted shortest path matrix is used.
dim	layout_with_mds() supports dimensions up to the number of nodes minus one, but only if the graph is connected; for unconnected graphs, the only possible value is 2. This is because merge_coords() only works in 2D.
options	This is currently ignored, as ARPACK is not used any more for solving the eigenproblem
...	Passed to layout_with_mds().

Details

`layout_with_mds()` uses classical multidimensional scaling (Torgerson scaling) for generating the coordinates. Multidimensional scaling aims to place points from a higher dimensional space in a (typically) 2 dimensional plane, so that the distances between the points are kept as much as this is possible.

By default igraph uses the shortest path matrix as the distances between the nodes, but the user can override this via the `dist` argument.

Warning: If the graph is symmetric to the exchange of two vertices (as is the case with leaves of a tree connecting to the same parent), classical multidimensional scaling may assign the same coordinates to these vertices.

This function generates the layout separately for each graph component and then merges them via `merge_coords()`.

Value

A numeric matrix with `dim` columns.

Related documentation in the C library

`layout_mds()`

Author(s)

Tamas Nepusz <ntamas@gmail.com> and Gabor Csardi <csardi.gabor@gmail.com>

References

Cox, T. F. and Cox, M. A. A. (2001) *Multidimensional Scaling*. Second edition. Chapman and Hall.

See Also

`layout()`, `plot.igraph()`

Other graph layouts: `add_layout_()`, `component_wise()`, `layout_()`, `layout_as_bipartite()`, `layout_as_star()`, `layout_as_tree()`, `layout_in_circle()`, `layout_nicely()`, `layout_on_grid()`, `layout_on_sphere()`, `layout_randomly()`, `layout_with_dh()`, `layout_with_fr()`, `layout_with_gem()`, `layout_with_graphopt()`, `layout_with_kk()`, `layout_with_lgl()`, `layout_with_sugiyama()`, `merge_coords()`, `norm_coords()`, `normalize()`

Examples

```
g <- sample_gnp(100, 2 / 100)
l <- layout_with_mds(g)
plot(g, layout = l, vertex.label = NA, vertex.size = 3)
```

`layout_with_sugiyama` *The Sugiyama graph layout generator*

Description

Sugiyama layout algorithm for layered directed acyclic graphs. The algorithm minimized edge crossings.

Usage

```
layout_with_sugiyama(
  graph,
  layers = NULL,
  hgap = 1,
  vgap = 1,
  maxiter = 100,
  weights = NULL,
  attributes = c("default", "all", "none")
)

with_sugiyama(...)
```

Arguments

`graph`	The input graph.
`layers`	A numeric vector or `NULL`. If not `NULL`, then it should specify the layer index of the vertices. Layers are numbered from one. If `NULL`, then igraph calculates the layers automatically.
`hgap`	Real scalar, the minimum horizontal gap between vertices in the same layer.
`vgap`	Real scalar, the distance between layers.
`maxiter`	Integer scalar, the maximum number of iterations in the crossing minimization stage. 100 is a reasonable default; if you feel that you have too many edge crossings, increase this.
`weights`	Optional edge weight vector. If `NULL`, then the 'weight' edge attribute is used, if there is one. Supply `NA` here and igraph ignores the edge weights. These are used only if the graph contains cycles; igraph will tend to reverse edges with smaller weights when breaking the cycles.
`attributes`	Which graph/vertex/edge attributes to keep in the extended graph. 'default' keeps the 'size', 'size2', 'shape', 'label' and 'color' vertex attributes and the 'arrow.mode' and 'arrow.size' edge attributes. 'all' keep all graph, vertex and edge attributes, 'none' keeps none of them.
`...`	Passed to `layout_with_sugiyama()`.

Details

This layout algorithm is designed for directed acyclic graphs where each vertex is assigned to a layer. Layers are indexed from zero, and vertices of the same layer will be placed on the same horizontal line. The X coordinates of vertices within each layer are decided by the heuristic proposed by Sugiyama et al. to minimize edge crossings.

You can also try to lay out undirected graphs, graphs containing cycles, or graphs without an a priori layered assignment with this algorithm. igraph will try to eliminate cycles and assign vertices to layers, but there is no guarantee on the quality of the layout in such cases.

The Sugiyama layout may introduce "bends" on the edges in order to obtain a visually more pleasing layout. This is achieved by adding dummy nodes to edges spanning more than one layer. The resulting layout assigns coordinates not only to the nodes of the original graph but also to the dummy nodes. The layout algorithm will also return the extended graph with the dummy nodes.

For more details, see the reference below.

Value

A list with the components:

layout The layout, a two-column matrix, for the original graph vertices.

layout.dummy The layout for the dummy vertices, a two column matrix.

extd_graph The original graph, extended with dummy vertices. The 'dummy' vertex attribute is set on this graph, it is a logical attributes, and it tells you whether the vertex is a dummy vertex. The 'layout' graph attribute is also set, and it is the layout matrix for all (original and dummy) vertices.

Related documentation in the C library

layout_sugiyama(), get_edgelist(), is_directed(), vcount(), edges(), get_eids(), ecount()

Author(s)

Tamas Nepusz <ntamas@gmail.com>

References

K. Sugiyama, S. Tagawa and M. Toda, "Methods for Visual Understanding of Hierarchical Systems". IEEE Transactions on Systems, Man and Cybernetics 11(2):109-125, 1981.

See Also

Other graph layouts: add_layout_(), component_wise(), layout_(), layout_as_bipartite(), layout_as_star(), layout_as_tree(), layout_in_circle(), layout_nicely(), layout_on_grid(), layout_on_sphere(), layout_randomly(), layout_with_dh(), layout_with_fr(), layout_with_gem(), layout_with_graphopt(), layout_with_kk(), layout_with_lgl(), layout_with_mds(), merge_coords(), norm_coords(), normalize()

Examples

```
## Data taken from http://tehnick-8.narod.ru/dc_clients/
DC <- graph_from_literal(
  "DC++" -+ "LinuxDC++":"BCDC++":"EiskaltDC++":"StrongDC++":"DiCe!++",
  "LinuxDC++" -+ "FreeDC++", "BCDC++" -+ "StrongDC++",
  "FreeDC++" -+ "BMDC++":"EiskaltDC++",
  "StrongDC++" -+ "AirDC++":"zK++":"ApexDC++":"TkDC++",
  "StrongDC++" -+ "StrongDC++ SQLite":"RSX++",
  "ApexDC++" -+ "FlylinkDC++ ver <= 4xx",
  "ApexDC++" -+ "ApexDC++ Speed-Mod":"DiCe!++",
  "StrongDC++ SQLite" -+ "FlylinkDC++ ver >= 5xx",
  "ApexDC++ Speed-Mod" -+ "FlylinkDC++ ver <= 4xx",
  "ApexDC++ Speed-Mod" -+ "GreylinkDC++",
  "FlylinkDC++ ver <= 4xx" -+ "FlylinkDC++ ver >= 5xx",
  "FlylinkDC++ ver <= 4xx" -+ AvaLink,
  "GreylinkDC++" -+ AvaLink:"RayLinkDC++":"SparkDC++":PeLink
)

## Use edge types
E(DC)$lty <- 1
E(DC)["BCDC++" %->% "StrongDC++"]$lty <- 2
E(DC)["FreeDC++" %->% "EiskaltDC++"]$lty <- 2
E(DC)["ApexDC++" %->% "FlylinkDC++ ver <= 4xx"]$lty <- 2
E(DC)["ApexDC++" %->% "DiCe!++"]$lty <- 2
E(DC)["StrongDC++ SQLite" %->% "FlylinkDC++ ver >= 5xx"]$lty <- 2
E(DC)["GreylinkDC++" %->% "AvaLink"]$lty <- 2

## Layers, as on the plot
layers <- list(
  c("DC++"),
  c("LinuxDC++", "BCDC++"),
  c("FreeDC++", "StrongDC++"),
  c(
    "BMDC++", "EiskaltDC++", "AirDC++", "zK++", "ApexDC++",
    "TkDC++", "RSX++"
  ),
  c("StrongDC++ SQLite", "ApexDC++ Speed-Mod", "DiCe!++"),
  c("FlylinkDC++ ver <= 4xx", "GreylinkDC++"),
  c(
    "FlylinkDC++ ver >= 5xx", "AvaLink", "RayLinkDC++",
    "SparkDC++", "PeLink"
  )
)

## Check that we have all nodes
all(sort(unlist(layers)) == sort(V(DC)$name))

## Add some graphical parameters
V(DC)$color <- "white"
V(DC)$shape <- "rectangle"
V(DC)$size <- 20
V(DC)$size2 <- 10
```

```
V(DC)$label <- lapply(V(DC)$name, function(x) {
  paste(strwrap(x, 12), collapse = "\n")
})
E(DC)$arrow.size <- 0.5

## Create a similar layout using the predefined layers
lay1 <- layout_with_sugiyama(DC, layers = apply(sapply(
  layers,
  function(x) V(DC)$name %in% x
), 1, which))

## Simple plot, not very nice
par(mar = rep(.1, 4))
plot(DC, layout = lay1$layout, vertex.label.cex = 0.5)

## Sugiyama plot
plot(lay1$extd_graph, vertex.label.cex = 0.5)

## The same with automatic layer calculation
## Keep vertex/edge attributes in the extended graph
lay2 <- layout_with_sugiyama(DC, attributes = "all")
plot(lay2$extd_graph, vertex.label.cex = 0.5)

## Another example, from the following paper:
## Markus Eiglsperger, Martin Siebenhaller, Michael Kaufmann:
## An Efficient Implementation of Sugiyama's Algorithm for
## Layered Graph Drawing, Journal of Graph Algorithms and
## Applications 9, 305--325 (2005).

ex <- graph_from_literal(
  0 -+ 29:6:5:20:4,
  1 -+ 12,
  2 -+ 23:8,
  3 -+ 4,
  4,
  5 -+ 2:10:14:26:4:3,
  6 -+ 9:29:25:21:13,
  7,
  8 -+ 20:16,
  9 -+ 28:4,
  10 -+ 27,
  11 -+ 9:16,
  12 -+ 9:19,
  13 -+ 20,
  14 -+ 10,
  15 -+ 16:27,
  16 -+ 27,
  17 -+ 3,
  18 -+ 13,
  19 -+ 9,
  20 -+ 4,
  21 -+ 22,
  22 -+ 8:9,
```

```
  23 -+ 9:24,
  24 -+ 12:15:28,
  25 -+ 11,
  26 -+ 18,
  27 -+ 13:19,
  28 -+ 7,
  29 -+ 25
)

layers <- list(
  0, c(5, 17), c(2, 14, 26, 3), c(23, 10, 18), c(1, 24),
  12, 6, c(29, 21), c(25, 22), c(11, 8, 15), 16, 27, c(13, 19),
  c(9, 20), c(4, 28), 7
)

layex <- layout_with_sugiyama(ex, layers = apply(
  sapply(
    layers,
    function(x) V(ex)$name %in% as.character(x)
  ),
  1, which
))

origvert <- c(rep(TRUE, vcount(ex)), rep(FALSE, nrow(layex$layout.dummy)))
realedge <- as_edgelist(layex$extd_graph)[, 2] <= vcount(ex)
plot(layex$extd_graph,
  vertex.label.cex = 0.5,
  edge.arrow.size = .5,
  vertex.size = ifelse(origvert, 5, 0),
  vertex.shape = ifelse(origvert, "square", "none"),
  vertex.label = ifelse(origvert, V(ex)$name, ""),
  edge.arrow.mode = ifelse(realedge, 2, 0)
)
```

`local_scan` *Compute local scan statistics on graphs*

Description

The scan statistic is a summary of the locality statistics that is computed from the local neighborhood of each vertex. The `local_scan()` function computes the local statistics for each vertex for a given neighborhood size and the statistic function.

Usage

```
local_scan(
  graph.us,
  graph.them = NULL,
  k = 1,
```

```
  FUN = NULL,
  weighted = FALSE,
  mode = c("out", "in", "all"),
  neighborhoods = NULL,
  weights = NULL,
  ...
)
```

Arguments

graph.us, graph	An igraph object, the graph for which the scan statistics will be computed
graph.them	An igraph object or NULL, if not NULL, then the 'them' statistics is computed, i.e. the neighborhoods calculated from graph.us are evaluated on graph.them.
k	An integer scalar, the size of the local neighborhood for each vertex. Should be non-negative.
FUN	Character, a function name, or a function object itself, for computing the local statistic in each neighborhood. If NULL(the default value), ecount() is used for unweighted graphs (if weighted=FALSE) and a function that computes the sum of edge weights is used for weighted graphs (if weighted=TRUE). This argument is ignored if k is zero.
weighted	Logical scalar, TRUE if the edge weights should be used for computation of the scan statistic. If TRUE, the graph should be weighted. Note that this argument is ignored if FUN is not NULL, "ecount" and "sumweights".
mode	Character scalar, the kind of neighborhoods to use for the calculation. One of 'out', 'in', 'all' or 'total'. This argument is ignored for undirected graphs.
neighborhoods	A list of neighborhoods, one for each vertex, or NULL. If it is not NULL, then the function is evaluated on the induced subgraphs specified by these neighborhoods. In theory this could be useful if the same graph.us graph is used for multiple graph.them arguments. Then the neighborhoods can be calculated on graph.us and used with multiple graphs. In practice, this is currently slower than simply using graph.them multiple times.
weights	Numeric vector, edge weights to use for the scan instead of the edge attribute weight. If NULL (the default) the edge weight attribute is used.
...	Arguments passed to FUN, the function that computes the local statistics.

Details

See the given reference below for the details on the local scan statistics.

local_scan() calculates exact local scan statistics.

If graph.them is NULL, then local_scan() computes the 'us' variant of the scan statistics. Otherwise, graph.them should be an igraph object and the 'them' variant is computed using graph.us to extract the neighborhood information, and applying FUN on these neighborhoods in graph.them.

Value

For `local_scan()` typically a numeric vector containing the computed local statistics for each vertex. In general a list or vector of objects, as returned by `FUN`.

Related documentation in the C library

`local_scan_0()`, `local_scan_0_them()`, `local_scan_1_ecount()`, `local_scan_1_ecount_them()`, `local_scan_k_ecount()`, `local_scan_k_ecount_them()`, `local_scan_neighborhood_ecount()`, `vcount()`, `induced_subgraph()`, `edges()`, `get_eids()`, `ecount()`

References

Priebe, C. E., Conroy, J. M., Marchette, D. J., Park, Y. (2005). Scan Statistics on Enron Graphs. *Computational and Mathematical Organization Theory*.

See Also

Other scan statistics: `scan_stat()`

Examples

```
pair <- sample_correlated_gnp_pair(n = 10^3, corr = 0.8, p = 0.1)
local_0_us <- local_scan(graph.us = pair$graph1, k = 0)
local_1_us <- local_scan(graph.us = pair$graph1, k = 1)

local_0_them <- local_scan(
  graph.us = pair$graph1,
  graph.them = pair$graph2, k = 0
)
local_1_them <- local_scan(
  graph.us = pair$graph1,
  graph.them = pair$graph2, k = 1
)

Neigh_1 <- neighborhood(pair$graph1, order = 1)
local_1_them_nhood <- local_scan(
  graph.us = pair$graph1,
  graph.them = pair$graph2,
  neighborhoods = Neigh_1
)
```

---

make_ *Make a new graph*

Description

This is a generic function for creating graphs.

Usage

```
make_(...)
```

Arguments

...	Parameters, see details below.

Details

`make_()` is a generic function for creating graphs. For every graph constructor in igraph that has a `make_` prefix, there is a corresponding function without the prefix: e.g. for `make_ring()` there is also `ring()`, etc.

The same is true for the random graph samplers, i.e. for each constructor with a `sample_` prefix, there is a corresponding function without that prefix.

These shorter forms can be used together with `make_()`. The advantage of this form is that the user can specify constructor modifiers which work with all constructors. E.g. the `with_vertex_()` modifier adds vertex attributes to the newly created graphs.

See the examples and the various constructor modifiers below.

Related documentation in the C library

`simplify()`, `vcount()`, `edges()`, `get_eids()`, `ecount()`

See Also

Other deterministic constructors: `graph_from_atlas()`, `graph_from_edgelist()`, `graph_from_literal()`, `make_chordal_ring()`, `make_circulant()`, `make_empty_graph()`, `make_full_citation_graph()`, `make_full_graph()`, `make_full_multipartite()`, `make_graph()`, `make_lattice()`, `make_ring()`, `make_star()`, `make_tree()`, `make_turan()`, `make_wheel()`

Constructor modifiers (and related functions): `sample_()`, `simplified()`, `with_edge_()`, `with_graph_()`, `with_vertex_()`, `without_attr()`, `without_loops()`, `without_multiples()`

Examples

```
r <- make_(ring(10))
l <- make_(lattice(c(3, 3, 3)))

r2 <- make_(ring(10), with_vertex_(color = "red", name = LETTERS[1:10]))
l2 <- make_(lattice(c(3, 3, 3)), with_edge_(weight = 2))

ran <- sample_(degseq(c(3, 3, 3, 3, 3, 3), method = "configuration"), simplified())
degree(ran)
is_simple(ran)
```

make_bipartite_graph *Create a bipartite graph*

Description

A bipartite graph has two kinds of vertices and connections are only allowed between different kinds.

Usage

```
make_bipartite_graph(types, edges, directed = FALSE)

bipartite_graph(types, edges, directed = FALSE)
```

Arguments

types	A vector giving the vertex types. It will be coerced into boolean. The length of the vector gives the number of vertices in the graph. When the vector is a named vector, the names will be attached to the graph as the name vertex attribute.
edges	A vector giving the edges of the graph, the same way as for the regular make_graph() function. It is checked that the edges indeed connect vertices of different kind, according to the supplied types vector. The vector may be a string vector if types is a named vector.
directed	Whether to create a directed graph, boolean constant. Note that by default undirected graphs are created, as this is more common for bipartite graphs.

Details

Bipartite graphs have a type vertex attribute in igraph, this is boolean and FALSE for the vertices of the first kind and TRUE for vertices of the second kind.

make_bipartite_graph() basically does three things. First it checks the edges vector against the vertex types. Then it creates a graph using the edges vector and finally it adds the types vector as a vertex attribute called type. edges may contain strings as vertex names; in this case, types must be a named vector that specifies the type for each vertex name that occurs in edges.

Value

make_bipartite_graph() returns a bipartite igraph graph. In other words, an igraph graph that has a vertex attribute named type.

is_bipartite() returns a logical scalar.

Related documentation in the C library

create_bipartite(), vcount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

make_graph() to create one-mode networks

Bipartite graphs: bipartite_mapping(), bipartite_projection(), is_bipartite()

Examples

```
g <- make_bipartite_graph(rep(0:1, length.out = 10), c(1:10))
print(g, v = TRUE)
```

---

make_chordal_ring *Create an extended chordal ring graph*

Description

make_chordal_ring() creates an extended chordal ring. An extended chordal ring is regular graph, each node has the same degree. It can be obtained from a simple ring by adding some extra edges specified by a matrix. Let p denote the number of columns in the 'W' matrix. The extra edges of vertex i are added according to column i mod p in 'W'. The number of extra edges is the number of rows in 'W': for each row j an edge i->i+w[ij] is added if i+w[ij] is less than the number of total nodes. See also Kotsis, G: Interconnection Topologies for Parallel Processing Systems, PARS Mitteilungen 11, 1-6, 1993.

Usage

```
make_chordal_ring(n, w, directed = FALSE)

chordal_ring(n, w, directed = FALSE)
```

Arguments

n	The number of vertices.
w	A matrix which specifies the extended chordal ring. See details below.
directed	Logical scalar, whether or not to create a directed graph.

Value

An igraph graph.

Related documentation in the C library

extended_chordal_ring()

See Also

Other deterministic constructors: graph_from_atlas(), graph_from_edgelist(), graph_from_literal(), make_(), make_circulant(), make_empty_graph(), make_full_citation_graph(), make_full_graph(), make_full_multipartite(), make_graph(), make_lattice(), make_ring(), make_star(), make_tree(), make_turan(), make_wheel()

Examples

```
chord <- make_chordal_ring(
  15,
  matrix(c(3, 12, 4, 7, 8, 11), nr = 2)
)
```

---

make_circulant *Create a circulant graph*

Description

A circulant graph C_n^{shifts} consists of n vertices $v_0, \ldots, v_{n-1}$ such that for each s_i in the list of offsets `shifts`, v_j is connected to $v_{(j+s_i) \mod n}$ for all j.

Usage

```
make_circulant(n, shifts, directed = FALSE)

circulant(n, shifts, directed = FALSE)
```

Arguments

n	Integer, the number of vertices in the circulant graph.
shifts	Integer vector, a list of the offsets within the circulant graph.
directed	Boolean, whether to create a directed graph.

Details

The function can generate either directed or undirected graphs. It does not generate multi-edges or self-loops.

Value

An igraph graph.

Related documentation in the C library

circulant()

See Also

Other deterministic constructors: graph_from_atlas(), graph_from_edgelist(), graph_from_literal(), make_(), make_chordal_ring(), make_empty_graph(), make_full_citation_graph(), make_full_graph(), make_full_multipartite(), make_graph(), make_lattice(), make_ring(), make_star(), make_tree(), make_turan(), make_wheel()

Examples

```
# Create a circulant graph with 10 vertices and shifts 1 and 3
g <- make_circulant(10, c(1, 3))
plot(g, layout = layout_in_circle)

# A directed circulant graph
g2 <- make_circulant(10, c(1, 3), directed = TRUE)
plot(g2, layout = layout_in_circle)
```

---

make_clusters *Creates a communities object.*

Description

This is useful to integrate the results of community finding algorithms that are not included in igraph.

Usage

```
make_clusters(
  graph,
  membership = NULL,
  algorithm = NULL,
  merges = NULL,
  modularity = TRUE
)
```

Arguments

graph	The graph of the community structure.
membership	The membership vector of the community structure, a numeric vector denoting the id of the community for each vertex. It might be NULL for hierarchical community structures.
algorithm	Character string, the algorithm that generated the community structure, it can be arbitrary.
merges	A merge matrix, for hierarchical community structures (or NULL otherwise.
modularity	Modularity value of the community structure. If this is TRUE and the membership vector is available, then it the modularity values is calculated automatically.

Value

A communities object.

membership A numeric vector giving the community id for each vertex.

modularity The modularity score of the partition.

algorithm If known, the algorithm used to obtain the communities.

vcount Number of vertices in the graph.

See Also

Community detection: as_membership(), cluster_edge_betweenness(), cluster_fast_greedy(), cluster_fluid_communities(), cluster_infomap(), cluster_label_prop(), cluster_leading_eigen(), cluster_leiden(), cluster_louvain(), cluster_optimal(), cluster_spinglass(), cluster_walktrap(), compare(), groups(), membership(), modularity.igraph(), plot_dendrogram(), split_join_distance(), voronoi_cells()

---

make_de_bruijn_graph *De Bruijn graphs*

Description

De Bruijn graphs are labeled graphs representing the overlap of strings.

Usage

```
make_de_bruijn_graph(m, n)

de_bruijn_graph(m, n)
```

Arguments

m	Integer scalar, the size of the alphabet. See details below.
n	Integer scalar, the length of the labels. See details below.

Details

A de Bruijn graph represents relationships between strings. An alphabet of m letters are used and strings of length n are considered. A vertex corresponds to every possible string and there is a directed edge from vertex v to vertex w if the string of v can be transformed into the string of w by removing its first letter and appending a letter to it.

Please note that the graph will have m to the power n vertices and even more edges, so probably you don't want to supply too big numbers for m and n.

De Bruijn graphs have some interesting properties, please see another source, e.g. Wikipedia for details.

Value

A graph object.

Related documentation in the C library

de_bruijn()

Author(s)

Gabor Csardi csardi.gabor@gmail.com

See Also

make_kautz_graph(), make_line_graph()

Examples

```
# de Bruijn graphs can be created recursively by line graphs as well
g <- make_de_bruijn_graph(2, 1)
make_de_bruijn_graph(2, 2)
make_line_graph(g)
```

---

make_empty_graph *A graph with no edges*

Description

A graph with no edges

Usage

```
make_empty_graph(n = 0, directed = TRUE)

empty_graph(n = 0, directed = TRUE)
```

Arguments

n	Number of vertices.
directed	Whether to create a directed graph.

Value

An igraph graph.

Related documentation in the C library

empty()

See Also

Other deterministic constructors: graph_from_atlas(), graph_from_edgelist(), graph_from_literal(), make_(), make_chordal_ring(), make_circulant(), make_full_citation_graph(), make_full_graph(), make_full_multipartite(), make_graph(), make_lattice(), make_ring(), make_star(), make_tree(), make_turan(), make_wheel()

Examples

```
make_empty_graph(n = 10)
make_empty_graph(n = 5, directed = FALSE)
```

---

make_from_prufer *Create an undirected tree graph from its Prüfer sequence*

Description

make_from_prufer() creates an undirected tree graph from its Prüfer sequence.

Usage

```
make_from_prufer(prufer)

from_prufer(prufer)
```

Arguments

prufer	The Prüfer sequence to convert into a graph

Details

The Prüfer sequence of a tree graph with n labeled vertices is a sequence of n-2 numbers, constructed as follows. If the graph has more than two vertices, find a vertex with degree one, remove it from the tree and add the label of the vertex that it was connected to to the sequence. Repeat until there are only two vertices in the remaining graph.

Value

A graph object.

Related documentation in the C library

from_prufer()

See Also

to_prufer() to convert a graph into its Prüfer sequence

Other trees: is_forest(), is_tree(), sample_spanning_tree(), to_prufer()

Examples

```
g <- make_tree(13, 3)
to_prufer(g)
```

`make_full_bipartite_graph` *Create a full bipartite graph*

Description

Bipartite graphs are also called two-mode by some. This function creates a bipartite graph in which every possible edge is present.

Usage

```
make_full_bipartite_graph(
  n1,
  n2,
  directed = FALSE,
  mode = c("all", "out", "in")
)

full_bipartite_graph(n1, n2, directed = FALSE, mode = c("all", "out", "in"))
```

Arguments

`n1`	The number of vertices of the first kind.
`n2`	The number of vertices of the second kind.
`directed`	Logical scalar, whether the graphs is directed.
`mode`	Scalar giving the kind of edges to create for directed graphs. If this is '`out`' then all vertices of the first kind are connected to the others; '`in`' specifies the opposite direction; '`all`' creates mutual edges. This argument is ignored for undirected graphs.x

Details

Bipartite graphs have a '`type`' vertex attribute in igraph, this is boolean and `FALSE` for the vertices of the first kind and `TRUE` for vertices of the second kind.

Value

An igraph graph, with the '`type`' vertex attribute set.

Related documentation in the C library

`full_bipartite()`, `vcount()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

make_full_graph() for creating one-mode full graphs

Examples

```
g <- make_full_bipartite_graph(2, 3)
g2 <- make_full_bipartite_graph(2, 3, directed = TRUE)
g3 <- make_full_bipartite_graph(2, 3, directed = TRUE, mode = "in")
g4 <- make_full_bipartite_graph(2, 3, directed = TRUE, mode = "all")
```

---

make_full_citation_graph

Create a complete (full) citation graph

Description

make_full_citation_graph() creates a full citation graph. This is a directed graph, where every i->j edge is present if and only if $j < i$. If directed=FALSE then the graph is just a full graph.

Usage

```
make_full_citation_graph(n, directed = TRUE)

full_citation_graph(n, directed = TRUE)
```

Arguments

n	The number of vertices.
directed	Whether to create a directed graph.

Value

An igraph graph.

Related documentation in the C library

full_citation()

See Also

Other deterministic constructors: graph_from_atlas(), graph_from_edgelist(), graph_from_literal(), make_(), make_chordal_ring(), make_circulant(), make_empty_graph(), make_full_graph(), make_full_multipartite(), make_graph(), make_lattice(), make_ring(), make_star(), make_tree(), make_turan(), make_wheel()

Examples

```
print_all(make_full_citation_graph(10))
```

`make_full_graph` *Create a full graph*

Description

Create a full graph

Usage

```
make_full_graph(n, directed = FALSE, loops = FALSE)

full_graph(n, directed = FALSE, loops = FALSE)
```

Arguments

`n`	Number of vertices.
`directed`	Whether to create a directed graph.
`loops`	Whether to add self-loops to the graph.

Value

An igraph graph

Related documentation in the C library

`full()`

See Also

Other deterministic constructors: `graph_from_atlas()`, `graph_from_edgelist()`, `graph_from_literal()`, `make_()`, `make_chordal_ring()`, `make_circulant()`, `make_empty_graph()`, `make_full_citation_graph()`, `make_full_multipartite()`, `make_graph()`, `make_lattice()`, `make_ring()`, `make_star()`, `make_tree()`, `make_turan()`, `make_wheel()`

Examples

```
make_full_graph(5)
print_all(make_full_graph(4, directed = TRUE))
```

make_full_multipartite *Create a full multipartite graph*

Description

A multipartite graph contains multiple types of vertices and connections are only possible between vertices of different types. This function creates a complete multipartite graph where all possible edges between different partitions are present.

Usage

```
make_full_multipartite(n, directed = FALSE, mode = c("all", "out", "in"))

full_multipartite(n, directed = FALSE, mode = c("all", "out", "in"))
```

Arguments

n	A numeric vector giving the number of vertices in each partition.
directed	Logical scalar, whether to create a directed graph.
mode	Character scalar, the type of connections for directed graphs. If "out", then edges point from vertices of partitions with lower indices to partitions with higher indices; if "in", then the opposite direction is realized; "all" creates mutual edges. This parameter is ignored for undirected graphs.

Value

An igraph graph with a vertex attribute type storing the partition index of each vertex. Partition indices start from 1.

Related documentation in the C library

full_multipartite(), vcount()

See Also

Other deterministic constructors: graph_from_atlas(), graph_from_edgelist(), graph_from_literal(), make_(), make_chordal_ring(), make_circulant(), make_empty_graph(), make_full_citation_graph(), make_full_graph(), make_graph(), make_lattice(), make_ring(), make_star(), make_tree(), make_turan(), make_wheel()

Examples

```
# Create a multipartite graph with partitions of size 2, 3, and 4
g <- make_full_multipartite(c(2, 3, 4))
plot(g)
```

```
# Create a directed multipartite graph
g2 <- make_full_multipartite(c(2, 2, 2), directed = TRUE, mode = "out")
plot(g2)
```

---

make_graph *Create an igraph graph from a list of edges, or a notable graph*

Description

Create an igraph graph from a list of edges, or a notable graph

Usage

```
make_graph(
  edges,
  ...,
  n = max(edges),
  isolates = NULL,
  directed = TRUE,
  dir = directed,
  simplify = TRUE
)

make_directed_graph(edges, n = max(edges))

make_undirected_graph(edges, n = max(edges))

directed_graph(...)

undirected_graph(...)
```

Arguments

edges	A vector defining the edges, the first edge points from the first element to the second, the second edge from the third to the fourth, etc. For a numeric vector, these are interpreted as internal vertex ids. For character vectors, they are interpreted as vertex names. Alternatively, this can be a character scalar, the name of a notable graph. See Notable graphs below. The name is case insensitive. Starting from igraph 0.8.0, you can also include literals here, via igraph's formula notation (see graph_from_literal()). In this case, the first term of the formula has to start with a '~' character, just like regular formulae in R. See examples below.
...	For make_graph(): extra arguments for the case when the graph is given via a literal, see graph_from_literal(). For directed_graph() and undirected_graph(): Passed to make_directed_graph() or make_undirected_graph().

`n`	The number of vertices in the graph. This argument is ignored (with a warning) if `edges` are symbolic vertex names. It is also ignored if there is a bigger vertex id in `edges`. This means that for this function it is safe to supply zero here if the vertex with the largest id is not an isolate.
`isolates`	Character vector, names of isolate vertices, for symbolic edge lists. It is ignored for numeric edge lists.
`directed`	Whether to create a directed graph.
`dir`	It is the same as `directed`, for compatibility. Do not give both of them.
`simplify`	For graph literals, whether to simplify the graph.

Value

An igraph graph.

Notable graphs

`make_graph()` can create some notable graphs. The name of the graph (case insensitive), a character scalar must be supplied as the `edges` argument, and other arguments are ignored. (A warning is given is they are specified.)

`make_graph()` knows the following graphs:

Bull The bull graph, 5 vertices, 5 edges, resembles to the head of a bull if drawn properly.

Chvatal This is the smallest triangle-free graph that is both 4-chromatic and 4-regular. According to the Grunbaum conjecture there exists an m-regular, m-chromatic graph with n vertices for every m>1 and n>2. The Chvatal graph is an example for m=4 and n=12. It has 24 edges.

Coxeter A non-Hamiltonian cubic symmetric graph with 28 vertices and 42 edges.

Cubical The Platonic graph of the cube. A convex regular polyhedron with 8 vertices and 12 edges.

Diamond A graph with 4 vertices and 5 edges, resembles to a schematic diamond if drawn properly.

Dodecahedral, Dodecahedron Another Platonic solid with 20 vertices and 30 edges.

Folkman The semisymmetric graph with minimum number of vertices, 20 and 40 edges. A semisymmetric graph is regular, edge transitive and not vertex transitive.

Franklin This is a graph whose embedding to the Klein bottle can be colored with six colors, it is a counterexample to the necessity of the Heawood conjecture on a Klein bottle. It has 12 vertices and 18 edges.

Frucht The Frucht Graph is the smallest cubical graph whose automorphism group consists only of the identity element. It has 12 vertices and 18 edges.

Grotzsch, Groetzsch The Grötzsch graph is a triangle-free graph with 11 vertices, 20 edges, and chromatic number 4. It is named after German mathematician Herbert Grötzsch, and its existence demonstrates that the assumption of planarity is necessary in Grötzsch's theorem that every triangle-free planar graph is 3-colorable.

Heawood The Heawood graph is an undirected graph with 14 vertices and 21 edges. The graph is cubic, and all cycles in the graph have six or more edges. Every smaller cubic graph has shorter cycles, so this graph is the 6-cage, the smallest cubic graph of girth 6.

Herschel The Herschel graph is the smallest nonhamiltonian polyhedral graph. It is the unique such graph on 11 nodes, and has 18 edges.

House The house graph is a 5-vertex, 6-edge graph, the schematic draw of a house if drawn properly, basically a triangle of the top of a square.

HouseX The same as the house graph with an X in the square. 5 vertices and 8 edges.

Icosahedral, Icosahedron A Platonic solid with 12 vertices and 30 edges.

Krackhardt kite A social network with 10 vertices and 18 edges. Krackhardt, D. Assessing the Political Landscape: Structure, Cognition, and Power in Organizations. Admin. Sci. Quart. 35, 342-369, 1990.

Levi The graph is a 4-arc transitive cubic graph, it has 30 vertices and 45 edges.

McGee The McGee graph is the unique 3-regular 7-cage graph, it has 24 vertices and 36 edges.

Meredith The Meredith graph is a quartic graph on 70 nodes and 140 edges that is a counterexample to the conjecture that every 4-regular 4-connected graph is Hamiltonian.

Noperfectmatching A connected graph with 16 vertices and 27 edges containing no perfect matching. A matching in a graph is a set of pairwise non-adjacent edges; that is, no two edges share a common vertex. A perfect matching is a matching which covers all vertices of the graph.

Nonline A graph whose connected components are the 9 graphs whose presence as a vertex-induced subgraph in a graph makes a nonline graph. It has 50 vertices and 72 edges.

Octahedral, Octahedron Platonic solid with 6 vertices and 12 edges.

Petersen A 3-regular graph with 10 vertices and 15 edges. It is the smallest hypohamiltonian graph, i.e. it is non-hamiltonian but removing any single vertex from it makes it Hamiltonian.

Robertson The unique (4,5)-cage graph, i.e. a 4-regular graph of girth 5. It has 19 vertices and 38 edges.

Smallestcyclicgroup A smallest nontrivial graph whose automorphism group is cyclic. It has 9 vertices and 15 edges.

Tetrahedral, Tetrahedron Platonic solid with 4 vertices and 6 edges.

Thomassen The smallest hypotraceable graph, on 34 vertices and 52 edges. A hypotraceable graph does not contain a Hamiltonian path but after removing any single vertex from it the remainder always contains a Hamiltonian path. A graph containing a Hamiltonian path is called traceable.

Tutte Tait's Hamiltonian graph conjecture states that every 3-connected 3-regular planar graph is Hamiltonian. This graph is a counterexample. It has 46 vertices and 69 edges.

Uniquely3colorable Returns a 12-vertex, triangle-free graph with chromatic number 3 that is uniquely 3-colorable.

Walther An identity graph with 25 vertices and 31 edges. An identity graph has a single graph automorphism, the trivial one.

Zachary Social network of friendships between 34 members of a karate club at a US university in the 1970s. See W. W. Zachary, An information flow model for conflict and fission in small groups, Journal of Anthropological Research 33, 452-473 (1977).

Related documentation in the C library

`create()`, `famous()`, `empty()`, `simplify()`, `vcount()`

See Also

Other deterministic constructors: graph_from_atlas(), graph_from_edgelist(), graph_from_literal(), make_(), make_chordal_ring(), make_circulant(), make_empty_graph(), make_full_citation_graph(), make_full_graph(), make_full_multipartite(), make_lattice(), make_ring(), make_star(), make_tree(), make_turan(), make_wheel()

Examples

```
make_graph(c(1, 2, 2, 3, 3, 4, 5, 6), directed = FALSE)
make_graph(c("A", "B", "B", "C", "C", "D"), directed = FALSE)

solids <- list(
  make_graph("Tetrahedron"),
  make_graph("Cubical"),
  make_graph("Octahedron"),
  make_graph("Dodecahedron"),
  make_graph("Icosahedron")
)

graph <- make_graph(
  ~ A - B - C - D - A, E - A:B:C:D,
  F - G - H - I - F, J - F:G:H:I,
  K - L - M - N - K, O - K:L:M:N,
  P - Q - R - S - P, T - P:Q:R:S,
  B - F, E - J, C - I, L - T, O - T, M - S,
  C - P, C - L, I - L, I - P
)
```

---

make_kautz_graph *Kautz graphs*

Description

Kautz graphs are labeled graphs representing the overlap of strings.

Usage

```
make_kautz_graph(m, n)

kautz_graph(m, n)
```

Arguments

m	Integer scalar, the size of the alphabet. See details below.
n	Integer scalar, the length of the labels. See details below.

Details

A Kautz graph is a labeled graph, vertices are labeled by strings of length `n+1` above an alphabet with `m+1` letters, with the restriction that every two consecutive letters in the string must be different. There is a directed edge from a vertex `v` to another vertex `w` if it is possible to transform the string of `v` into the string of `w` by removing the first letter and appending a letter to it.

Kautz graphs have some interesting properties, see e.g. Wikipedia for details.

Value

A graph object.

Related documentation in the C library

`kautz()`

Author(s)

Gabor Csardi csardi.gabor@gmail.com, the first version in R was written by Vincent Matossian.

See Also

`make_de_bruijn_graph()`, `make_line_graph()`

Examples

```
make_line_graph(make_kautz_graph(2, 1))
make_kautz_graph(2, 2)
```

---

make_lattice *Create a lattice graph*

Description

`make_lattice()` is a flexible function, it can create lattices of arbitrary dimensions, periodic or aperiodic ones. It has two forms. In the first form you only supply `dimvector`, but not `length` and `dim`. In the second form you omit `dimvector` and supply `length` and `dim`.

Usage

```
make_lattice(
  dimvector = NULL,
  length = NULL,
  dim = NULL,
  nei = 1,
  directed = FALSE,
  mutual = FALSE,
```

```
  periodic = FALSE,
  circular = deprecated()
)

lattice(
  dimvector = NULL,
  length = NULL,
  dim = NULL,
  nei = 1,
  directed = FALSE,
  mutual = FALSE,
  periodic = FALSE,
  circular = deprecated()
)
```

Arguments

dimvector	A vector giving the size of the lattice in each dimension.
length	Integer constant, for regular lattices, the size of the lattice in each dimension.
dim	Integer constant, the dimension of the lattice.
nei	The distance within which (inclusive) the neighbors on the lattice will be connected. This parameter is not used right now.
directed	Whether to create a directed lattice.
mutual	Logical, if TRUE directed lattices will be mutually connected.
periodic	Logical vector, Boolean vector, defines whether the generated lattice is periodic along each dimension. This parameter may also be scalar boolen value which will be extended to boolean vector with dimvector length.
circular	Deprecated, use periodic instead.

Value

An igraph graph.

Related documentation in the C library

square_lattice()

See Also

Other deterministic constructors: graph_from_atlas(), graph_from_edgelist(), graph_from_literal(), make_(), make_chordal_ring(), make_circulant(), make_empty_graph(), make_full_citation_graph(), make_full_graph(), make_full_multipartite(), make_graph(), make_ring(), make_star(), make_tree(), make_turan(), make_wheel()

Examples

```
make_lattice(c(5, 5, 5))
make_lattice(length = 5, dim = 3)
```

make_line_graph *Line graph of a graph*

Description

This function calculates the line graph of another graph.

Usage

```
make_line_graph(graph)

line_graph(graph)
```

Arguments

graph	The input graph, it can be directed or undirected.

Details

The line graph L(G) of a G undirected graph is defined as follows. L(G) has one vertex for each edge in G and two vertices in L(G) are connected by an edge if their corresponding edges share an end point.

The line graph L(G) of a G directed graph is slightly different, L(G) has one vertex for each edge in G and two vertices in L(G) are connected by a directed edge if the target of the first vertex's corresponding edge is the same as the source of the second vertex's corresponding edge.

Value

A new graph object.

Related documentation in the C library

linegraph()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>, the first version of the C code was written by Vincent Matossian.

Examples

```
# generate the first De-Bruijn graphs
g <- make_full_graph(2, directed = TRUE, loops = TRUE)
make_line_graph(g)
make_line_graph(make_line_graph(g))
make_line_graph(make_line_graph(make_line_graph(g)))
```

make_ring *Create a ring graph*

Description

A ring is a one-dimensional lattice and this function is a special case of make_lattice().

Usage

```
make_ring(n, directed = FALSE, mutual = FALSE, circular = TRUE)

ring(n, directed = FALSE, mutual = FALSE, circular = TRUE)
```

Arguments

n	Number of vertices.
directed	Whether the graph is directed.
mutual	Whether directed edges are mutual. It is ignored in undirected graphs.
circular	Whether to create a circular ring. A non-circular ring is essentially a "line": a tree where every non-leaf vertex has one child.

Value

An igraph graph.

Related documentation in the C library

ring()

See Also

Other deterministic constructors: graph_from_atlas(), graph_from_edgelist(), graph_from_literal(), make_(), make_chordal_ring(), make_circulant(), make_empty_graph(), make_full_citation_graph(), make_full_graph(), make_full_multipartite(), make_graph(), make_lattice(), make_star(), make_tree(), make_turan(), make_wheel()

Examples

```
print_all(make_ring(10))
print_all(make_ring(10, directed = TRUE, mutual = TRUE))
```

make_star *Create a star graph, a tree with n vertices and n - 1 leaves*

Description

`star()` creates a star graph, in this every single vertex is connected to the center vertex and nobody else.

Usage

```
make_star(n, mode = c("in", "out", "mutual", "undirected"), center = 1)

star(n, mode = c("in", "out", "mutual", "undirected"), center = 1)
```

Arguments

`n`	Number of vertices.
`mode`	It defines the direction of the edges, `in`: the edges point *to* the center, `out`: the edges point *from* the center, `mutual`: a directed star is created with mutual edges, `undirected`: the edges are undirected.
`center`	ID of the center vertex.

Value

An igraph graph.

Related documentation in the C library

`star()`

See Also

Other deterministic constructors: `graph_from_atlas()`, `graph_from_edgelist()`, `graph_from_literal()`, `make_()`, `make_chordal_ring()`, `make_circulant()`, `make_empty_graph()`, `make_full_citation_graph()`, `make_full_graph()`, `make_full_multipartite()`, `make_graph()`, `make_lattice()`, `make_ring()`, `make_tree()`, `make_turan()`, `make_wheel()`

Examples

```
make_star(10, mode = "out")
make_star(5, mode = "undirected")
```

make_tree *Create tree graphs*

Description

Create a k-ary tree graph, where almost all vertices other than the leaves have the same number of children.

Usage

```
make_tree(n, children = 2, mode = c("out", "in", "undirected"))

tree(...)
```

Arguments

n	Number of vertices.
children	Integer scalar, the number of children of a vertex (except for leafs)
mode	Defines the direction of the edges. `out` indicates that the edges point from the parent to the children, `in` indicates that they point from the children to their parents, while `undirected` creates an undirected graph.
...	Passed to `make_tree()` or `sample_tree()`.

Value

An igraph graph

Related documentation in the C library

kary_tree()

See Also

Other deterministic constructors: graph_from_atlas(), graph_from_edgelist(), graph_from_literal(), make_(), make_chordal_ring(), make_circulant(), make_empty_graph(), make_full_citation_graph(), make_full_graph(), make_full_multipartite(), make_graph(), make_lattice(), make_ring(), make_star(), make_turan(), make_wheel()

Examples

```
make_tree(10, 2)
make_tree(10, 3, mode = "undirected")
```

make_turan *Create a Turán graph*

Description

Turán graphs are complete multipartite graphs with the property that the sizes of the partitions are as close to equal as possible.

Usage

```
make_turan(n, r)

turan(n, r)
```

Arguments

n	Integer, the number of vertices in the graph.
r	Integer, the number of partitions in the graph, must be positive.

Details

The Turán graph with n vertices and r partitions is the densest graph on n vertices that does not contain a clique of size r+1.

This function generates undirected graphs. The null graph is returned when the number of vertices is zero. A complete graph is returned if the number of partitions is greater than the number of vertices.

Value

An igraph graph with a vertex attribute type storing the partition index of each vertex. Partition indices start from 1.

Related documentation in the C library

turan(), vcount()

See Also

Other deterministic constructors: graph_from_atlas(), graph_from_edgelist(), graph_from_literal(), make_(), make_chordal_ring(), make_circulant(), make_empty_graph(), make_full_citation_graph(), make_full_graph(), make_full_multipartite(), make_graph(), make_lattice(), make_ring(), make_star(), make_tree(), make_wheel()

Examples

```
# Create a Turán graph with 10 vertices and 3 partitions
g <- make_turan(10, 3)
plot(g)

# The sizes of the partitions are as balanced as possible
table(V(g)$type)
```

`make_wheel` *Create a wheel graph*

Description

[Experimental]

A wheel graph is created by connecting a center vertex to all vertices of a cycle graph. A wheel graph on `n` vertices can be thought of as a wheel with `n - 1` spokes. The cycle graph part makes up the rim, while the star graph part adds the spokes.

Note that the two and three-vertex wheel graphs are non-simple: The two-vertex wheel graph contains a self-loop, while the three-vertex wheel graph contains parallel edges (a 1-cycle and a 2-cycle, respectively).

Usage

```
make_wheel(n, ..., mode = c("in", "out", "mutual", "undirected"), center = 1)

wheel(n, ..., mode = c("in", "out", "mutual", "undirected"), center = 1)
```

Arguments

`n`	Number of vertices.
`...`	These dots are for future extensions and must be empty.
`mode`	It defines the direction of the edges. `in`: the edges point *to* the center, `out`: the edges point *from* the center, `mutual`: a directed wheel is created with mutual edges, `undirected`: the edges are undirected.
`center`	ID of the center vertex.

Value

An igraph graph.

Related documentation in the C library

`wheel()`

See Also

Other deterministic constructors: graph_from_atlas(), graph_from_edgelist(), graph_from_literal(), make_(), make_chordal_ring(), make_circulant(), make_empty_graph(), make_full_citation_graph(), make_full_graph(), make_full_multipartite(), make_graph(), make_lattice(), make_ring(), make_star(), make_tree(), make_turan()

Examples

```
make_wheel(10, mode = "out")
make_wheel(5, mode = "undirected")
```

---

match_vertices *Match Graphs given a seeding of vertex correspondences*

Description

Given two adjacency matrices A and B of the same size, match the two graphs with the help of m seed vertex pairs which correspond to the first m rows (and columns) of the adjacency matrices.

Usage

```
match_vertices(A, B, m, start, iteration)
```

Arguments

A	a numeric matrix, the adjacency matrix of the first graph
B	a numeric matrix, the adjacency matrix of the second graph
m	The number of seeds. The first m vertices of both graphs are matched.
start	a numeric matrix, the permutation matrix estimate is initialized with start
iteration	The number of iterations for the Frank-Wolfe algorithm

Details

The approximate graph matching problem is to find a bijection between the vertices of two graphs , such that the number of edge disagreements between the corresponding vertex pairs is minimized. For seeded graph matching, part of the bijection that consist of known correspondences (the seeds) is known and the problem task is to complete the bijection by estimating the permutation matrix that permutes the rows and columns of the adjacency matrix of the second graph.

It is assumed that for the two supplied adjacency matrices A and B, both of size $n \times n$, the first m rows(and columns) of A and B correspond to the same vertices in both graphs. That is, the $n \times n$ permutation matrix that defines the bijection is $I_m \bigoplus P$ for a $(n-m) \times (n-m)$ permutation matrix P and m times m identity matrix I_m. The function match_vertices() estimates the permutation matrix P via an optimization algorithm based on the Frank-Wolfe algorithm.

See references for further details.

Value

A numeric matrix which is the permutation matrix that determines the bijection between the graphs of A and B

Author(s)

Vince Lyzinski https://www.ams.jhu.edu/~lyzinski/

References

Vogelstein, J. T., Conroy, J. M., Podrazik, L. J., Kratzer, S. G., Harley, E. T., Fishkind, D. E.,Vogelstein, R. J., Priebe, C. E. (2011). Fast Approximate Quadratic Programming for Large (Brain) Graph Matching. Online: https://arxiv.org/abs/1112.5507

Fishkind, D. E., Adali, S., Priebe, C. E. (2012). Seeded Graph Matching Online: https://arxiv.org/abs/1209.0367

See Also

sample_correlated_gnp(),sample_correlated_gnp_pair()

Examples

```
# require(Matrix)
g1 <- sample_gnp(10, 0.1)
randperm <- c(1:3, 3 + sample(7))
g2 <- sample_correlated_gnp(g1, corr = 1, p = g1$p, permutation = randperm)
A <- as_adjacency_matrix(g1)
B <- as_adjacency_matrix(g2)
P <- match_vertices(A, B, m = 3, start = diag(rep(1, nrow(A) - 3)), 20)
P
```

---

max_cardinality *Maximum cardinality search*

Description

Maximum cardinality search is a simple ordering a vertices that is useful in determining the chordality of a graph.

Usage

```
max_cardinality(graph)
```

Arguments

graph	The input graph. It may be directed, but edge directions are ignored, as the algorithm is defined for undirected graphs.

Details

Maximum cardinality search visits the vertices in such an order that every time the vertex with the most already visited neighbors is visited. Ties are broken randomly.

The algorithm provides a simple basis for deciding whether a graph is chordal, see References below, and also `is_chordal()`.

Value

A list with two components:

alpha Numeric vector. The 1-based rank of each vertex in the graph such that the vertex with rank 1 is visited first, the vertex with rank 2 is visited second and so on.

alpham1 Numeric vector. The inverse of `alpha`. In other words, the elements of this vector are the vertices in reverse maximum cardinality search order.

Related documentation in the C library

`maximum_cardinality_search()`, `vcount()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Robert E Tarjan and Mihalis Yannakakis. (1984). Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal of Computation* 13, 566–579.

See Also

`is_chordal()`

Other chordal: `is_chordal()`

Examples

```
## The examples from the Tarjan-Yannakakis paper
g1 <- graph_from_literal(
  A - B:C:I, B - A:C:D, C - A:B:E:H, D - B:E:F,
  E - C:D:F:H, F - D:E:G, G - F:H, H - C:E:G:I,
  I - A:H
)
max_cardinality(g1)
is_chordal(g1, fillin = TRUE)

g2 <- graph_from_literal(
  A - B:E, B - A:E:F:D, C - E:D:G, D - B:F:E:C:G,
  E - A:B:C:D:F, F - B:D:E, G - C:D:H:I, H - G:I:J,
  I - G:H:J, J - H:I
)
```

```
max_cardinality(g2)
is_chordal(g2, fillin = TRUE)
```

`max_flow` *Maximum flow in a graph*

Description

In a graph where each edge has a given flow capacity the maximal flow between two vertices is calculated.

Usage

```
max_flow(graph, source, target, capacity = NULL)
```

Arguments

`graph`	The input graph.
`source`	The id of the source vertex.
`target`	The id of the target vertex (sometimes also called sink).
`capacity`	Vector giving the capacity of the edges. If this is `NULL` (the default) then the `capacity` edge attribute is used. Note that the `weight` edge attribute is not used by this function.

Details

`max_flow()` calculates the maximum flow between two vertices in a weighted (i.e. valued) graph. A flow from `source` to `target` is an assignment of non-negative real numbers to the edges of the graph, satisfying two properties: (1) for each edge the flow (i.e. the assigned number) is not more than the capacity of the edge (the `capacity` parameter or edge attribute), (2) for every vertex, except the source and the target the incoming flow is the same as the outgoing flow. The value of the flow is the incoming flow of the `target` vertex. The maximum flow is the flow of maximum value.

Value

A named list with components:

value A numeric scalar, the value of the maximum flow.

flow A numeric vector, the flow itself, one entry for each edge. For undirected graphs this entry is bit trickier, since for these the flow direction is not predetermined by the edge direction. For these graphs the elements of the this vector can be negative, this means that the flow goes from the bigger vertex id to the smaller one. Positive values mean that the flow goes from the smaller vertex id to the bigger one.

cut A numeric vector of edge ids, the minimum cut corresponding to the maximum flow.

partition1 A numeric vector of vertex ids, the vertices in the first partition of the minimum cut corresponding to the maximum flow.

partition2 A numeric vector of vertex ids, the vertices in the second partition of the minimum cut corresponding to the maximum flow.

stats A list with some statistics from the push-relabel algorithm. Five integer values currently: `nopush` is the number of push operations, `norelabel` the number of relabelings, `nogap` is the number of times the gap heuristics was used, `nogapnodes` is the total number of gap nodes omitted because of the gap heuristics and `nobfs` is the number of times a global breadth-first-search update was performed to assign better height (=distance) values to the vertices.

Related documentation in the C library

`maxflow()`, `edges()`, `ecount()`, `vcount()`, `get_eids()`

References

A. V. Goldberg and R. E. Tarjan: A New Approach to the Maximum Flow Problem *Journal of the ACM* 35:921-940, 1988.

See Also

Other flow: `dominator_tree()`, `edge_connectivity()`, `is_min_separator()`, `is_separator()`, `min_cut()`, `min_separators()`, `min_st_separators()`, `st_cuts()`, `st_min_cuts()`, `vertex_connectivity()`

Examples

```
E <- rbind(c(1, 3, 3), c(3, 4, 1), c(4, 2, 2), c(1, 5, 1), c(5, 6, 2), c(6, 2, 10))
colnames(E) <- c("from", "to", "capacity")
g1 <- graph_from_data_frame(as.data.frame(E))
max_flow(g1, source = V(g1)["1"], target = V(g1)["2"])
```

`membership` *Functions to deal with the result of network community detection*

Description

igraph community detection functions return their results as an object from the `communities` class. This manual page describes the operations of this class.

Usage

```
membership(communities)

## S3 method for class 'communities'
print(x, ...)

## S3 method for class 'communities'
modularity(x, ...)

## S3 method for class 'communities'
```

```
length(x)

sizes(communities)

algorithm(communities)

merges(communities)

crossing(communities, graph)

code_len(communities)

is_hierarchical(communities)

## S3 method for class 'communities'
as.dendrogram(object, hang = -1, use.modularity = FALSE, ...)

## S3 method for class 'communities'
as.hclust(x, hang = -1, use.modularity = FALSE, ...)

cut_at(communities, no, steps)

show_trace(communities)

## S3 method for class 'communities'
plot(
  x,
  y,
  col = membership(x),
  mark.groups = communities(x),
  edge.color = c("black", "red")[crossing(x, y) + 1],
  ...
)

communities(x)
```

Arguments

communities, x, object	A communities object, the result of an igraph community detection function.
...	Additional arguments. plot.communities passes these to plot.igraph(). The other functions silently ignore them.
graph	An igraph graph object, corresponding to communities.
hang	Numeric scalar indicating how the height of leaves should be computed from the heights of their parents; see plot.hclust().
use.modularity	Logical scalar, whether to use the modularity values to define the height of the branches.

no	Integer scalar, the desired number of communities. If too low or two high, then an error message is given. Exactly one of no and steps must be supplied.
steps	The number of merge operations to perform to produce the communities. Exactly one of no and steps must be supplied.
y	An igraph graph object, corresponding to the communities in x.
col	A vector of colors, in any format that is accepted by the regular R plotting methods. This vector gives the colors of the vertices explicitly.
mark.groups	A list of numeric vectors. The communities can be highlighted using colored polygons. The groups for which the polygons are drawn are given here. The default is to use the groups given by the communities. Supply NULL here if you do not want to highlight any groups.
edge.color	The colors of the edges. By default the edges within communities are colored green and other edges are red.
membership	Numeric vector, one value for each vertex, the membership vector of the community structure. Might also be NULL if the community structure is given in another way, e.g. by a merge matrix.
algorithm	If not NULL (meaning an unknown algorithm), then a character scalar, the name of the algorithm that produced the community structure.
merges	If not NULL, then the merge matrix of the hierarchical community structure. See merges() below for more information on its format.
modularity	Numeric scalar or vector, the modularity value of the community structure. It can also be NULL, if the modularity of the (best) split is not available.

Details

Community structure detection algorithms try to find dense subgraphs in directed or undirected graphs, by optimizing some criteria, and usually using heuristics.

igraph implements a number of community detection methods (see them below), all of which return an object of the class communities. Because the community structure detection algorithms are different, communities objects do not always have the same structure. Nevertheless, they have some common operations, these are documented here.

The print() generic function is defined for communities, it prints a short summary.

The length generic function call be called on communities and returns the number of communities.

The sizes() function returns the community sizes, in the order of their ids.

membership() gives the division of the vertices, into communities. It returns a numeric vector, one value for each vertex, the id of its community. Community ids start from one. Note that some algorithms calculate the complete (or incomplete) hierarchical structure of the communities, and not just a single partitioning. For these algorithms typically the membership for the highest modularity value is returned, but see also the manual pages of the individual algorithms.

communities() is also the name of a function, that returns a list of communities, each identified by their vertices. The vertices will have symbolic names if the add.vertex.names igraph option is set, and the graph itself was named. Otherwise numeric vertex ids are used.

modularity() gives the modularity score of the partitioning. (See modularity.igraph() for details. For algorithms that do not result a single partitioning, the highest modularity value is returned.

`algorithm()` gives the name of the algorithm that was used to calculate the community structure.

`crossing()` returns a logical vector, with one value for each edge, ordered according to the edge ids. The value is `TRUE` iff the edge connects two different communities, according to the (best) membership vector, as returned by `membership()`.

`is_hierarchical()` checks whether a hierarchical algorithm was used to find the community structure. Some functions only make sense for hierarchical methods (e.g. `merges()`, `cut_at()` and `as.dendrogram()`).

`merges()` returns the merge matrix for hierarchical methods. An error message is given, if a non-hierarchical method was used to find the community structure. You can check this by calling `is_hierarchical()` on the `communities` object.

`cut_at()` cuts the merge tree of a hierarchical community finding method, at the desired place and returns a membership vector. The desired place can be expressed as the desired number of communities or as the number of merge steps to make. The function gives an error message, if called with a non-hierarchical method.

`as.dendrogram()` converts a hierarchical community structure to a `dendrogram` object. It only works for hierarchical methods, and gives an error message to others. See `stats::dendrogram()` for details.

`stats::as.hclust()` is similar to `as.dendrogram()`, but converts a hierarchical community structure to a `hclust` object.

`ape::as.phylo()` converts a hierarchical community structure to a `phylo` object, you will need the `ape` package for this.

`show_trace()` works (currently) only for communities found by the leading eigenvector method (`cluster_leading_eigen()`), and returns a character vector that gives the steps performed by the algorithm while finding the communities.

`code_len()` is defined for the InfoMAP method (`cluster_infomap()` and returns the code length of the partition.

It is possibly to call the `plot()` function on `communities` objects. This will plot the graph (and uses `plot.igraph()` internally), with the communities shown. By default it colores the vertices according to their communities, and also marks the vertex groups corresponding to the communities. It passes additional arguments to `plot.igraph()`, please see that and also igraph.plotting on how to change the plot.

Value

`print()` returns the `communities` object itself, invisibly.

`length` returns an integer scalar.

`sizes()` returns a numeric vector.

`membership()` returns a numeric vector, one number for each vertex in the graph that was the input of the community detection.

`modularity()` returns a numeric scalar.

`algorithm()` returns a character scalar.

`crossing()` returns a logical vector.

`is_hierarchical()` returns a logical scalar.

`merges()` returns a two-column numeric matrix.

`cut_at()` returns a numeric vector, the membership vector of the vertices.

`as.dendrogram()` returns a dendrogram object.

`show_trace()` returns a character vector.

`code_len()` returns a numeric scalar for communities found with the InfoMAP method and `NULL` for other methods.

`plot()` for `communities` objects returns `NULL`, invisibly.

Related documentation in the C library

`get_edgelist()`, `vcount()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

See `plot_dendrogram()` for plotting community structure dendrograms.

See `compare()` for comparing two community structures on the same graph.

Community detection: `as_membership()`, `cluster_edge_betweenness()`, `cluster_fast_greedy()`, `cluster_fluid_communities()`, `cluster_infomap()`, `cluster_label_prop()`, `cluster_leading_eigen()`, `cluster_leiden()`, `cluster_louvain()`, `cluster_optimal()`, `cluster_spinglass()`, `cluster_walktrap()`, `compare()`, `groups()`, `make_clusters()`, `modularity.igraph()`, `plot_dendrogram()`, `split_join_distance()`, `voronoi_cells()`

Examples

```
karate <- make_graph("Zachary")
wc <- cluster_walktrap(karate)
modularity(wc)
membership(wc)
plot(wc, karate)
```

---

merge_coords *Merging graph layouts*

Description

Place several graphs on the same layout

Usage

```
merge_coords(graphs, layouts, method = "dla")

layout_components(graph, layout = layout_with_kk, ...)
```

Arguments

graphs	A list of graph objects.
layouts	A list of two-column matrices.
method	Character constant giving the method to use. Right now only dla is implemented.
graph	The input graph.
layout	A function object, the layout function to use.
...	Additional arguments to pass to the layout layout function.

Details

merge_coords() takes a list of graphs and a list of coordinates and places the graphs in a common layout. The method to use is chosen via the method parameter, although right now only the dla method is implemented.

The dla method covers the graph with circles. Then it sorts the graphs based on the number of vertices first and places the largest graph at the center of the layout. Then the other graphs are placed in decreasing order via a DLA (diffision limited aggregation) algorithm: the graph is placed randomly on a circle far away from the center and a random walk is conducted until the graph walks into the larger graphs already placed or walks too far from the center of the layout.

The layout_components() function disassembles the graph first into maximal connected components and calls the supplied layout function for each component separately. Finally it merges the layouts via calling merge_coords().

Value

A matrix with two columns and as many lines as the total number of vertices in the graphs.

Related documentation in the C library

decompose(), vcount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

plot.igraph(), tkplot(), layout(), disjoint_union()

Other graph layouts: add_layout_(), component_wise(), layout_(), layout_as_bipartite(), layout_as_star(), layout_as_tree(), layout_in_circle(), layout_nicely(), layout_on_grid(), layout_on_sphere(), layout_randomly(), layout_with_dh(), layout_with_fr(), layout_with_gem(), layout_with_graphopt(), layout_with_kk(), layout_with_lgl(), layout_with_mds(), layout_with_sugiyama(), norm_coords(), normalize()

Examples

```
# create 20 scale-free graphs and place them in a common layout
graphs <- lapply(sample(5:20, 20, replace = TRUE),
  barabasi.game,
  directed = FALSE
)
layouts <- lapply(graphs, layout_with_kk)
lay <- merge_coords(graphs, layouts)
g <- disjoint_union(graphs)
plot(g, layout = lay, vertex.size = 3, labels = NA, edge.color = "black")
```

---

min_cut *Minimum cut in a graph*

Description

`min_cut()` calculates the minimum st-cut between two vertices in a graph (if the `source` and `target` arguments are given) or the minimum cut of the graph (if both `source` and `target` are `NULL`).

Usage

```
min_cut(
  graph,
  source = NULL,
  target = NULL,
  capacity = NULL,
  value.only = TRUE
)
```

Arguments

`graph`	The input graph.
`source`	The id of the source vertex.
`target`	The id of the target vertex (sometimes also called sink).
`capacity`	Vector giving the capacity of the edges. If this is `NULL` (the default) then the `capacity` edge attribute is used.
`value.only`	Logical scalar, if `TRUE` only the minimum cut value is returned, if `FALSE` the edges in the cut and a the two (or more) partitions are also returned.

Details

The minimum st-cut between `source` and `target` is the minimum total weight of edges needed to remove to eliminate all paths from `source` to `target`.

The minimum cut of a graph is the minimum total weight of the edges needed to remove to separate the graph into (at least) two components. (Which is to make the graph *not* strongly connected in the directed case.)

The maximum flow between two vertices in a graph is the same as the minimum st-cut, so `max_flow()` and `min_cut()` essentially calculate the same quantity, the only difference is that `min_cut()` can be invoked without giving the `source` and `target` arguments and then minimum of all possible minimum cuts is calculated.

For undirected graphs the Stoer-Wagner algorithm (see reference below) is used to calculate the minimum cut.

Value

For `min_cut()` a nuieric constant, the value of the minimum cut, except if `value.only = FALSE`. In this case a named list with components:

value Numeric scalar, the cut value.

cut Numeric vector, the edges in the cut.

partition1 The vertices in the first partition after the cut edges are removed. Note that these vertices might be actually in different components (after the cut edges are removed), as the graph may fall apart into more than two components.

partition2 The vertices in the second partition after the cut edges are removed. Note that these vertices might be actually in different components (after the cut edges are removed), as the graph may fall apart into more than two components.

Related documentation in the C library

`mincut()`, `mincut_value()`, `st_mincut()`, `st_mincut_value()`, `edges()`, `ecount()`, `vcount()`, `get_eids()`

References

M. Stoer and F. Wagner: A simple min-cut algorithm, *Journal of the ACM*, 44 585-591, 1997.

See Also

Other flow: `dominator_tree()`, `edge_connectivity()`, `is_min_separator()`, `is_separator()`, `max_flow()`, `min_separators()`, `min_st_separators()`, `st_cuts()`, `st_min_cuts()`, `vertex_connectivity()`

Examples

```
g <- make_ring(100)
min_cut(g, capacity = rep(1, vcount(g)))
min_cut(g, value.only = FALSE, capacity = rep(1, vcount(g)))

g2 <- make_graph(c(1, 2, 2, 3, 3, 4, 1, 6, 6, 5, 5, 4, 4, 1))
E(g2)$capacity <- c(3, 1, 2, 10, 1, 3, 2)
min_cut(g2, value.only = FALSE)
```

`min_separators` *Minimum size vertex separators*

Description

Find all vertex sets of minimal size whose removal separates the graph into more components

Usage

```
min_separators(graph)
```

Arguments

`graph`	The input graph. It may be directed, but edge directions are ignored.

Details

This function implements the Kanevsky algorithm for finding all minimal-size vertex separators in an undirected graph. See the reference below for the details.

In the special case of a fully connected input graph with n vertices, all subsets of size $n - 1$ are listed as the result.

Value

A list of numeric vectors. Each numeric vector is a vertex separator.

Related documentation in the C library

`minimum_size_separators()`, `vcount()`

References

Arkady Kanevsky: Finding all minimum-size separating vertex sets in a graph. *Networks* 23 533–541, 1993.

JS Provan and DR Shier: A Paradigm for listing (s,t)-cuts in graphs, *Algorithmica* 15, 351–372, 1996.

J. Moody and D. R. White. Structural cohesion and embeddedness: A hierarchical concept of social groups. *American Sociological Review*, 68 103–127, Feb 2003.

See Also

Other flow: `dominator_tree()`, `edge_connectivity()`, `is_min_separator()`, `is_separator()`, `max_flow()`, `min_cut()`, `min_st_separators()`, `st_cuts()`, `st_min_cuts()`, `vertex_connectivity()`

Examples

```
# The graph from the Moody-White paper
mw <- graph_from_literal(
  1 - 2:3:4:5:6, 2 - 3:4:5:7, 3 - 4:6:7, 4 - 5:6:7,
  5 - 6:7:21, 6 - 7, 7 - 8:11:14:19, 8 - 9:11:14, 9 - 10,
  10 - 12:13, 11 - 12:14, 12 - 16, 13 - 16, 14 - 15, 15 - 16,
  17 - 18:19:20, 18 - 20:21, 19 - 20:22:23, 20 - 21,
  21 - 22:23, 22 - 23
)

# Cohesive subgraphs
mw1 <- induced_subgraph(mw, as.character(c(1:7, 17:23)))
mw2 <- induced_subgraph(mw, as.character(7:16))
mw3 <- induced_subgraph(mw, as.character(17:23))
mw4 <- induced_subgraph(mw, as.character(c(7, 8, 11, 14)))
mw5 <- induced_subgraph(mw, as.character(1:7))

min_separators(mw)
min_separators(mw1)
min_separators(mw2)
min_separators(mw3)
min_separators(mw4)
min_separators(mw5)

# Another example, the science camp network
camp <- graph_from_literal(
  Harry:Steve:Don:Bert - Harry:Steve:Don:Bert,
  Pam:Brazey:Carol:Pat - Pam:Brazey:Carol:Pat,
  Holly - Carol:Pat:Pam:Jennie:Bill,
  Bill - Pauline:Michael:Lee:Holly,
  Pauline - Bill:Jennie:Ann,
  Jennie - Holly:Michael:Lee:Ann:Pauline,
  Michael - Bill:Jennie:Ann:Lee:John,
  Ann - Michael:Jennie:Pauline,
  Lee - Michael:Bill:Jennie,
  Gery - Pat:Steve:Russ:John,
  Russ - Steve:Bert:Gery:John,
  John - Gery:Russ:Michael
)
min_separators(camp)
```

---

min_st_separators *Minimum size vertex separators*

Description

List all vertex sets that are minimal (s, t) separators for some s and t, in an undirected graph.

Usage

```
min_st_separators(graph)
```

Arguments

graph	The input graph. It may be directed, but edge directions are ignored.

Details

A (s, t) vertex separator is a set of vertices, such that after their removal from the graph, there is no path between s and t in the graph.

A (s, t) vertex separator is minimal if none of its proper subsets is an (s, t) vertex separator for the same s and t.

Value

A list of numeric vectors. Each vector contains a vertex set (defined by vertex ids), each vector is an (s,t) separator of the input graph, for some s and t.

Note

Note that the code below returns `{1, 3}` despite its subset `{1}` being a separator as well. This is because `{1, 3}` is minimal with respect to separating vertices 2 and 4.

```
g <- make_graph(~ 0-1-2-3-4-1)
min_st_separators(g)
```

```
#> [[1]]
#> + 1/5 vertex, named:
#> [1] 1
#>
#> [[2]]
#> + 2/5 vertices, named:
#> [1] 2 4
#>
#> [[3]]
#> + 2/5 vertices, named:
#> [1] 1 3
```

Related documentation in the C library

`all_minimal_st_separators()`, `vcount()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Anne Berry, Jean-Paul Bordat and Olivier Cogis: Generating All the Minimal Separators of a Graph, In: Peter Widmayer, Gabriele Neyer and Stephan Eidenbenz (editors): *Graph-theoretic concepts in computer science*, 1665, 167–172, 1999. Springer.

See Also

Other flow: dominator_tree(), edge_connectivity(), is_min_separator(), is_separator(), max_flow(), min_cut(), min_separators(), st_cuts(), st_min_cuts(), vertex_connectivity()

Examples

```
ring <- make_ring(4)
min_st_separators(ring)

chvatal <- make_graph("chvatal")
min_st_separators(chvatal)
# https://github.com/r-lib/roxygen2/issues/1092
```

---

modularity.igraph *Modularity of a community structure of a graph*

Description

This function calculates how modular is a given division of a graph into subgraphs.

Usage

```
## S3 method for class 'igraph'
modularity(x, membership, weights = NULL, resolution = 1, directed = TRUE, ...)

modularity_matrix(
  graph,
  membership = lifecycle::deprecated(),
  weights = NULL,
  resolution = 1,
  directed = TRUE
)
```

Arguments

x, graph	The input graph.
membership	Numeric vector, one value for each vertex, the membership vector of the community structure.
weights	If not NULL then a numeric vector giving edge weights.
resolution	The resolution parameter. Must be greater than or equal to 0. Set it to 1 to use the classical definition of modularity.
directed	Whether to use the directed or undirected version of modularity. Ignored for undirected graphs.
...	Additional arguments, none currently.

Details

`modularity()` calculates the modularity of a graph with respect to the given `membership` vector.

The modularity of a graph with respect to some division (or vertex types) measures how good the division is, or how separated are the different vertex types from each other. It defined as

$$Q = \frac{1}{2m} \sum_{i,j} (A_{ij} - \gamma \frac{k_i k_j}{2m}) \delta(c_i, c_j),$$

here m is the number of edges, A_{ij} is the element of the A adjacency matrix in row i and column j, k_i is the degree of i, k_j is the degree of j, c_i is the type (or component) of i, c_j that of j, the sum goes over all i and j pairs of vertices, and $\delta(x, y)$ is 1 if $x = y$ and 0 otherwise. For directed graphs, it is defined as

$$Q = \frac{1}{m} \sum_{i,j} (A_{ij} - \gamma \frac{k_i^{out} k_j^{in}}{m}) \delta(c_i, c_j).$$

The resolution parameter γ allows weighting the random null model, which might be useful when finding partitions with a high modularity. Maximizing modularity with higher values of the resolution parameter typically results in more, smaller clusters when finding partitions with a high modularity. Lower values typically results in fewer, larger clusters. The original definition of modularity is retrieved when setting γ to 1.

If edge weights are given, then these are considered as the element of the A adjacency matrix, and k_i is the sum of weights of adjacent edges for vertex i.

`modularity_matrix()` calculates the modularity matrix. This is a dense matrix, and it is defined as the difference of the adjacency matrix and the configuration model null model matrix. In other words element M_{ij} is given as $A_{ij} - d_i d_j/(2m)$, where A_{ij} is the (possibly weighted) adjacency matrix, d_i is the degree of vertex i, and m is the number of edges (or the total weights in the graph, if it is weighed).

Value

For `modularity()` a numeric scalar, the modularity score of the given configuration.

For `modularity_matrix()` a numeric square matrix, its order is the number of vertices in the graph.

Related documentation in the C library

`modularity()`, `edges()`, `get_eids()`, `vcount()`, `ecount()`, `modularity_matrix()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Clauset, A.; Newman, M. E. J. & Moore, C. Finding community structure in very large networks, *Physical Review E* 2004, 70, 066111

See Also

cluster_walktrap(), cluster_edge_betweenness(), cluster_fast_greedy(), cluster_spinglass(), cluster_louvain() and cluster_leiden() for various community detection methods.

Community detection: as_membership(), cluster_edge_betweenness(), cluster_fast_greedy(), cluster_fluid_communities(), cluster_infomap(), cluster_label_prop(), cluster_leading_eigen(), cluster_leiden(), cluster_louvain(), cluster_optimal(), cluster_spinglass(), cluster_walktrap(), compare(), groups(), make_clusters(), membership(), plot_dendrogram(), split_join_distance(), voronoi_cells()

Examples

```
g <- make_full_graph(5) %du% make_full_graph(5) %du% make_full_graph(5)
g <- add_edges(g, c(1, 6, 1, 11, 6, 11))
wtc <- cluster_walktrap(g)
modularity(wtc)
modularity(g, membership(wtc))
```

---

motifs *Graph motifs*

Description

Graph motifs are small connected induced subgraphs with a well-defined structure. These functions search a graph for various motifs.

Usage

```
motifs(graph, size = 3, cut.prob = NULL, callback = NULL)
```

Arguments

graph	Graph object, the input graph.
size	The size of the motif, currently sizes 3 and 4 are supported in directed graphs and sizes 3 to 6 in undirected graphs.
cut.prob	Numeric vector giving the probabilities that the search graph is cut at a certain level. Its length should be the same as the size of the motif (the size argument). If NULL, the default, no cuts are made.
callback	Optional callback function to call for each motif found. The function should accept two arguments: vids (integer vector of vertex IDs in the motif) and isoclass (the isomorphism class of the motif). The function should return FALSE to continue the search or TRUE to stop it. If NULL (the default), motif counts are returned as a numeric vector. **Important limitation:** Callback functions must NOT call any igraph functions (including simple queries like vcount() or ecount()). Doing so will cause R to crash due to reentrancy issues. Extract any needed graph information before calling the function with a callback, or use collector mode (the default) and process results afterward.

Details

motifs() searches a graph for motifs of a given size and returns a numeric vector containing the number of different motifs. The order of the motifs is defined by their isomorphism class, see isomorphism_class().

Value

When callback is NULL, motifs() returns a numeric vector, the number of occurrences of each motif in the graph. The motifs are ordered by their isomorphism classes. Note that for unconnected subgraphs, which are not considered to be motifs, the result will be NA.

When callback is provided, the function returns NULL invisibly and calls the callback function for each motif found.

Related documentation in the C library

motifs_randesu(), motifs_randesu_callback()

See Also

isomorphism_class()

Other graph motifs: count_motifs(), dyad_census(), sample_motifs()

Examples

```
g <- sample_pa(100)
motifs(g, 3)
count_motifs(g, 3)
sample_motifs(g, 3)

# Using callback to stop search after finding 5 motifs
count <- 0
motifs(g, 3, callback = function(vids, isoclass) {
  count <<- count + 1
  count < 5  # stop after 5 motifs
})
```

---

mst *Minimum spanning tree*

Description

A *spanning tree* of a connected graph is a connected subgraph with the smallest number of edges that includes all vertices of the graph. A graph will have many spanning trees. Among these, the *minimum spanning tree* will have the smallest sum of edge weights.

Usage

```
mst(graph, weights = NULL, algorithm = NULL, ...)
```

Arguments

graph	The graph object to analyze.
weights	Numeric vector giving the weights of the edges in the graph. The order is determined by the edge ids. This is ignored if the unweighted algorithm is chosen. Edge weights are interpreted as distances.
algorithm	The algorithm to use for calculation. unweighted can be used for unweighted graphs, and prim runs Prim's algorithm for weighted graphs. If this is NULL then igraph will select the algorithm automatically: if the graph has an edge attribute called weight or the weights argument is not NULL then Prim's algorithm is chosen, otherwise the unweighted algorithm is used.
...	Additional arguments, unused.

Details

The *minimum spanning forest* of a disconnected graph is the collection of minimum spanning trees of all of its components.

If the graph is not connected a minimum spanning forest is returned.

Value

A graph object with the minimum spanning forest. To check whether it is a tree, check that the number of its edges is vcount(graph)-1. The edge and vertex attributes of the original graph are preserved in the result.

Related documentation in the C library

minimum_spanning_tree_unweighted(), minimum_spanning_tree_prim(), edges(), get_eids(), vcount(), ecount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Prim, R.C. 1957. Shortest connection networks and some generalizations *Bell System Technical Journal*, 37 1389–1401.

See Also

components()

Examples

```
g <- sample_gnp(100, 3 / 100)
g_mst <- mst(g)
```

neighbors *Neighboring (adjacent) vertices in a graph*

Description

A vertex is a neighbor of another one (in other words, the two vertices are adjacent), if they are incident to the same edge.

Usage

```
neighbors(graph, v, mode = c("out", "in", "all", "total"))
```

Arguments

graph	The input graph.
v	The vertex of which the adjacent vertices are queried.
mode	Whether to query outgoing ('out'), incoming ('in') edges, or both types ('all'). This is ignored for undirected graphs.

Value

A vertex sequence containing the neighbors of the input vertex.

Related documentation in the C library

neighbors(), vcount()

See Also

Other structural queries: [.igraph(), [[.igraph(), adjacent_vertices(), are_adjacent(), ends(), get_edge_ids(), gorder(), gsize(), head_of(), incident(), incident_edges(), is_directed(), tail_of()

Examples

```
g <- make_graph("Zachary")
n1 <- neighbors(g, 1)
n34 <- neighbors(g, 34)
intersection(n1, n34)
```

norm_coords *Normalize coordinates for plotting graphs*

Description

Rescale coordinates linearly to be within given bounds.

Usage

```
norm_coords(
  layout,
  xmin = -1,
  xmax = 1,
  ymin = -1,
  ymax = 1,
  zmin = -1,
  zmax = 1
)
```

Arguments

layout	A matrix with two or three columns, the layout to normalize.
xmin, xmax	The limits for the first coordinate, if one of them or both are NULL then no normalization is performed along this direction.
ymin, ymax	The limits for the second coordinate, if one of them or both are NULL then no normalization is performed along this direction.
zmin, zmax	The limits for the third coordinate, if one of them or both are NULL then no normalization is performed along this direction.

Details

norm_coords() normalizes a layout, it linearly transforms each coordinate separately to fit into the given limits.

Value

A numeric matrix with at the same dimension as layout.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Other graph layouts: add_layout_(), component_wise(), layout_(), layout_as_bipartite(), layout_as_star(), layout_as_tree(), layout_in_circle(), layout_nicely(), layout_on_grid(), layout_on_sphere(), layout_randomly(), layout_with_dh(), layout_with_fr(), layout_with_gem(), layout_with_graphopt(), layout_with_kk(), layout_with_lgl(), layout_with_mds(), layout_with_sugiyama(), merge_coords(), normalize()

---

normalize *Normalize layout*

Description

Scale coordinates of a layout.

Usage

```
normalize(
  xmin = -1,
  xmax = 1,
  ymin = xmin,
  ymax = xmax,
  zmin = xmin,
  zmax = xmax
)
```

Arguments

xmin, xmax	Minimum and maximum for x coordinates.
ymin, ymax	Minimum and maximum for y coordinates.
zmin, zmax	Minimum and maximum for z coordinates.

See Also

merge_coords(), layout_().

Other layout modifiers: component_wise(), layout_modifier()

Other graph layouts: add_layout_(), component_wise(), layout_(), layout_as_bipartite(), layout_as_star(), layout_as_tree(), layout_in_circle(), layout_nicely(), layout_on_grid(), layout_on_sphere(), layout_randomly(), layout_with_dh(), layout_with_fr(), layout_with_gem(), layout_with_graphopt(), layout_with_kk(), layout_with_lgl(), layout_with_mds(), layout_with_sugiyama(), merge_coords(), norm_coords()

Examples

```
layout_(make_ring(10), with_fr(), normalize())
```

page_rank *The Page Rank algorithm*

Description

Calculates the Google PageRank for the specified vertices.

Usage

```
page_rank(
  graph,
  algo = c("prpack", "arpack"),
  vids = V(graph),
  directed = TRUE,
  damping = 0.85,
  personalized = NULL,
  weights = NULL,
  options = NULL
)
```

Arguments

graph	The graph object.
algo	Character scalar, which implementation to use to carry out the calculation. The default is "prpack", which uses the PRPACK library (https://github.com/dgleich/prpack) to calculate PageRank scores by solving a set of linear equations. This is a new implementation in igraph version 0.7, and the suggested one, as it is the most stable and the fastest for all but small graphs. "arpack" uses the ARPACK library, the default implementation from igraph version 0.5 until version 0.7. It computes PageRank scores by solving an eingevalue problem.
vids	The vertices of interest.
directed	Logical, if true directed paths will be considered for directed graphs. It is ignored for undirected graphs.
damping	The damping factor ('d' in the original paper).
personalized	Optional vector giving a probability distribution to calculate personalized PageRank. For personalized PageRank, the probability of jumping to a node when abandoning the random walk is not uniform, but it is given by this vector. The vector should contains an entry for each vertex and it will be rescaled to sum up to one.
weights	A numerical vector or NULL. This argument can be used to give edge weights for calculating the weighted PageRank of vertices. If this is NULL and the graph has a weight edge attribute then that is used. If weights is a numerical vector then it used, even if the graph has a weights edge attribute. If this is NA, then no edge weights are used (even if the graph has a weight edge attribute. This function interprets edge weights as connection strengths. In the random surfer model, an edge with a larger weight is more likely to be selected by the surfer.

options A named list, to override some ARPACK options. See arpack() for details. This argument is ignored if the PRPACK implementation is used.

Details

For the explanation of the PageRank algorithm, see the following webpage: http://infolab.stanford.edu/~backrub/google.html, or the following reference:

Sergey Brin and Larry Page: The Anatomy of a Large-Scale Hypertextual Web Search Engine. Proceedings of the 7th World-Wide Web Conference, Brisbane, Australia, April 1998.

The page_rank() function can use either the PRPACK library or ARPACK (see arpack()) to perform the calculation.

Please note that the PageRank of a given vertex depends on the PageRank of all other vertices, so even if you want to calculate the PageRank for only some of the vertices, all of them must be calculated. Requesting the PageRank for only some of the vertices does not result in any performance increase at all.

Value

A named list with entries:

vector A numeric vector with the PageRank scores.

value When using the ARPACK method, the eigenvalue corresponding to the eigenvector with the PageRank scores is returned here. It is expected to be exactly one, and can be used to check that ARPACK has successfully converged to the expected eingevector. When using the PRPACK method, it is always set to 1.0.

options Some information about the underlying ARPACK calculation. See arpack() for details. This entry is NULL if not the ARPACK implementation was used.

Related documentation in the C library

personalized_pagerank(), vcount(), edges(), get_eids(), ecount()

Author(s)

Tamas Nepusz <ntamas@gmail.com> and Gabor Csardi <csardi.gabor@gmail.com>

References

Sergey Brin and Larry Page: The Anatomy of a Large-Scale Hypertextual Web Search Engine. Proceedings of the 7th World-Wide Web Conference, Brisbane, Australia, April 1998.

See Also

Other centrality scores: closeness(), betweenness(), degree()

Centrality measures: alpha_centrality(), authority_score(), betweenness(), closeness(), diversity(), eigen_centrality(), harmonic_centrality(), hits_scores(), power_centrality(), spectrum(), strength(), subgraph_centrality()

Examples

```
g <- sample_gnp(20, 5 / 20, directed = TRUE)
page_rank(g)$vector

g2 <- make_star(10)
page_rank(g2)$vector

# Personalized PageRank
g3 <- make_ring(10)
page_rank(g3)$vector
reset <- seq(vcount(g3))
page_rank(g3, personalized = reset)$vector
```

---

path *Helper function to add or delete edges along a path*

Description

This function can be used to add or delete edges that form a path.

Usage

```
path(...)
```

Arguments

...	See details below.

Details

When adding edges via +, all unnamed arguments are concatenated, and each element of a final vector is interpreted as a vertex in the graph. For a vector of length $n + 1$, n edges are then added, from vertex 1 to vertex 2, from vertex 2 to vertex 3, etc. Named arguments will be used as edge attributes for the new edges.

When deleting edges, all attributes are concatenated and then passed to `delete_edges()`.

Value

A special object that can be used together with igraph graphs and the plus and minus operators.

See Also

Other functions for manipulating graph structure: `+.igraph()`, `add_edges()`, `add_vertices()`, `complementer()`, `compose()`, `connect()`, `contract()`, `delete_edges()`, `delete_vertices()`, `difference()`, `difference.igraph()`, `disjoint_union()`, `edge()`, `igraph-minus`, `intersection()`, `intersection.igraph()`, `permute()`, `rep.igraph()`, `reverse_edges()`, `simplify()`, `transitive_closure()`, `union()`, `union.igraph()`, `vertex()`

Examples

```
# Create a (directed) wheel
g <- make_star(11, center = 1) + path(2:11, 2)
plot(g)

g <- make_empty_graph(directed = FALSE, n = 10) %>%
  set_vertex_attr("name", value = letters[1:10])

g2 <- g + path("a", "b", "c", "d")
plot(g2)

g3 <- g2 + path("e", "f", "g", weight = 1:2, color = "red")
E(g3)[[]]

g4 <- g3 + path(c("f", "c", "j", "d"), width = 1:3, color = "green")
E(g4)[[]]
```

---

permute *Permute the vertices of a graph*

Description

Create a new graph, by permuting vertex ids.

Usage

```
permute(graph, permutation)
```

Arguments

graph	The input graph, it can directed or undirected.
permutation	A numeric vector giving the permutation to apply. The first element is the new id of vertex 1, etc. Every number between one and `vcount(graph)` must appear exactly once.

Details

This function creates a new graph from the input graph by permuting its vertices according to the specified mapping. Call this function with the output of `canonical_permutation()` to create the canonical form of a graph.

`permute()` keeps all graph, vertex and edge attributes of the graph.

Value

A new graph object.

Related documentation in the C library

`permute_vertices()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

canonical_permutation()

Other functions for manipulating graph structure: +.igraph(), add_edges(), add_vertices(), complementer(), compose(), connect(), contract(), delete_edges(), delete_vertices(), difference(), difference.igraph(), disjoint_union(), edge(), igraph-minus, intersection(), intersection.igraph(), path(), rep.igraph(), reverse_edges(), simplify(), transitive_closure(), union(), union.igraph(), vertex()

Examples

```
# Random permutation of a random graph
g <- sample_gnm(20, 50)
g2 <- permute(g, sample(vcount(g)))
isomorphic(g, g2)

# Permutation keeps all attributes
g$name <- "Random graph, Gnm, 20, 50"
V(g)$name <- letters[1:vcount(g)]
E(g)$weight <- sample(1:5, ecount(g), replace = TRUE)
g2 <- permute(g, sample(vcount(g)))
isomorphic(g, g2)
g2$name
V(g2)$name
E(g2)$weight
all(sort(E(g2)$weight) == sort(E(g)$weight))
```

---

plot.common *Drawing graphs*

Description

The common bits of the three plotting functions plot.igraph, tkplot and rglplot are discussed in this manual page.

Details

There are currently three different functions in the igraph package which can draw graph in various ways:

plot.igraph does simple non-interactive 2D plotting to R devices. Actually it is an implementation of the graphics::plot() generic function, so you can write plot(graph) instead of plot.igraph(graph). As it used the standard R devices it supports every output format for which R has an output device. The list is quite impressing: PostScript, PDF files, XFig files, SVG files, JPG, PNG and of course you can plot to the screen as well using the default devices, or the good-looking anti-aliased Cairo device. See plot.igraph() for some more information.

`tkplot()` does interactive 2D plotting using the `tcltk` package. It can only handle graphs of moderate size, a thousand vertices is probably already too many. Some parameters of the plotted graph can be changed interactively after issuing the `tkplot` command: the position, color and size of the vertices and the color and width of the edges. See `tkplot()` for details.

`rglplot()` is an experimental function to draw graphs in 3D using OpenGL. See `rglplot()` for some more information.

Please also check the examples below.

How to specify graphical parameters

There are three ways to give values to the parameters described below, in section 'Parameters'. We give these three ways here in the order of their precedence.

The first method is to supply named arguments to the plotting commands: `plot.igraph()`, `tkplot()` or rglplot()]. Parameters for vertices start with prefix '`vertex.`', parameters for edges have prefix '`edge.`', and global parameters have no prefix. Eg. the color of the vertices can be given via argument `vertex.color`, whereas `edge.color` sets the color of the edges. `layout` gives the layout of the graphs.

The second way is to assign vertex, edge and graph attributes to the graph. These attributes have no prefix, ie. the color of the vertices is taken from the `color` vertex attribute and the color of the edges from the `color` edge attribute. The layout of the graph is given by the `layout` graph attribute. (Always assuming that the corresponding command argument is not present.) Setting vertex and edge attributes are handy if you want to assign a given 'look' to a graph, attributes are saved with the graph is you save it with `base::save()` or in GraphML format with `write_graph()`, so the graph will have the same look after loading it again.

If a parameter is not given in the command line, and the corresponding vertex/edge/graph attribute is also missing then the general igraph parameters handled by `igraph_options()` are also checked. Vertex parameters have prefix '`vertex.`', edge parameters are prefixed with '`edge.`', general parameters like `layout` are prefixed with '`plot`'. These parameters are useful if you want all or most of your graphs to have the same look, vertex size, vertex color, etc. Then you don't need to set these at every plotting, and you also don't need to assign vertex/edge attributes to every graph.

If the value of a parameter is not specified by any of the three ways described here, its default valued is used, as given in the source code.

Different parameters can have different type, eg. vertex colors can be given as a character vector with color names, or as an integer vector with the color numbers from the current palette. Different types are valid for different parameters, this is discussed in detail in the next section. It is however always true that the parameter can always be a function object in which it will be called with the graph as its single argument to get the "proper" value of the parameter. (If the function returns another function object that will *not* be called again...)

The list of parameters

Vertex parameters first, note that the '`vertex.`' prefix needs to be added if they are used as an argument or when setting via `igraph_options()`. The value of the parameter may be scalar valid for every vertex or a vector with a separate value for each vertex. (Shorter vectors are recycled.)

size The size of the vertex, a numeric scalar or vector, in the latter case each vertex sizes may differ. This vertex sizes are scaled in order have about the same size of vertices for a given value for

all three plotting commands. It does not need to be an integer number. The default value is 15. This is big enough to place short labels on vertices. If `size.scaling` is `TRUE`, `relative.size` is used to scale the size appropriately.

size2 The "other" size of the vertex, for some vertex shapes. For the various rectangle shapes this gives the height of the vertices, whereas `size` gives the width. It is ignored by shapes for which the size can be specified with a single number.

The default is 15.

color The fill color of the vertex. If it is numeric then the current palette is used, see `grDevices::palette()`. If it is a character vector then it may either contain integer values, named colors or RGB specified colors with three or four bytes. All strings starting with '#' are assumed to be RGB color specifications. It is possible to mix named color and RGB colors. Note that `tkplot()` ignores the fourth byte (alpha channel) in the RGB color specification.

For `plot.igraph` and integer values, the default igraph palette is used (see the 'palette' parameter below. Note that this is different from the R palette.

If you don't want (some) vertices to have any color, supply `NA` as the color name.

The default value is "`SkyBlue2`".

frame.color The color of the frame of the vertices, the same formats are allowed as for the fill color.

If you don't want vertices to have a frame, supply `NA` as the color name.

By default it is "black".

frame.width The width of the frame of the vertices. The default value is 1.

shape The shape of the vertex, currently "`circle`", "`square`", "`csquare`", "`rectangle`", "`crectangle`", "`vrectangle`", "`pie`" (see `vertex.shape.pie()`), '`sphere`', and "`none`" are supported, and only by the `plot.igraph()` command. "`none`" does not draw the vertices at all, although vertex label are plotted (if given). See `shapes()` for details about vertex shapes and `vertex.shape.pie()` for using pie charts as vertices.

The "`sphere`" vertex shape plots vertices as 3D ray-traced spheres, in the given color and size. This produces a raster image and it is only supported with some graphics devices. On some devices raster transparency is not supported and the spheres do not have a transparent background. See dev.capabilities and the '`rasterImage`' capability to check that your device is supported.

By default vertices are drawn as circles.

label The vertex labels. They will be converted to character. Specify `NA` to omit vertex labels. The default vertex labels are the vertex ids.

label.family The font family to be used for vertex labels. As different plotting commands can used different fonts, they interpret this parameter different ways. The basic notation is, however, understood by both `plot.igraph()` and `tkplot()`. `rglplot()` does not support fonts at all right now, it ignores this parameter completely.

For `plot.igraph()` this parameter is simply passed to `graphics::text()` as argument `family`.

For `tkplot()` some conversion is performed. If this parameter is the name of an existing Tk font, then that font is used and the `label.font` and `label.cex` parameters are ignored completely. If it is one of the base families (serif, sans, mono) then Times, Helvetica or Courier fonts are used, there are guaranteed to exist on all systems. For the 'symbol' base family we used the symbol font is available, otherwise the first font which has 'symbol' in its name. If the parameter is not a name of the base families and it is also not a named Tk font

then we pass it to tcltk::tkfont.create() and hope the user knows what she is doing. The label.font and label.cex parameters are also passed to tcltk::tkfont.create() in this case.

The default value is 'serif'.

label.font The font within the font family to use for the vertex labels. It is interpreted the same way as the the font graphical parameter: 1 is plain text, 2 is bold face, 3 is italic, 4 is bold and italic and 5 specifies the symbol font.

For plot.igraph() this parameter is simply passed to graphics::text().

For tkplot(), if the label.family parameter is not the name of a Tk font then this parameter is used to set whether the newly created font should be italic and/or boldface. Otherwise it is ignored.

For rglplot() it is ignored.

The default value is 1.

label.cex The font size for vertex labels. It is interpreted as a multiplication factor of some device-dependent base font size.

For plot.igraph() it is simply passed to graphics::text() as argument cex.

For tkplot() it is multiplied by 12 and then used as the size argument for tcltk::tkfont.create(). The base font is thus 12 for tkplot.

For rglplot() it is ignored.

The default value is 1.

label.dist The distance of the label from the center of the vertex. If it is 0 then the label is centered on the vertex. If it is 1 then the label is displayed beside the vertex.

The default value is 0.

label.degree It defines the position of the vertex labels, relative to the center of the vertices. It is interpreted as an angle in radians, zero means 'to the right', and 'pi' means to the left, up is -pi/2 and down is pi/2.

The default value is -pi/4.

label.color The color of the labels, see the color vertex parameter discussed earlier for the possible values.

The default value is black.

label.angle The rotation of the vertex labels, in degrees. Corresponds to the srt parameter of graphics::text().

label.adj one or two numeric values, giving the horizontal and vertical adjustment of the vertex labels. See also adj in graphics::text().

size.scaling Switches between absolute vertex sizing (FALSE,default) and relative (TRUE). If FALSE, vertex.size and vertex.size2 are used as is. If TRUE, relative.size is used to scale both appropriately with relative.size.

relative.size The relative size of the smallest and largest vertices as percentage of the plotting region. When all vertices have the same size, then by default the relative size observed in the plot will be equal to relative.size[2]. The default value is c(.01,.025) (1\

Only used if size.scaling is TRUE'.

Edge parameters require to add the 'edge.' prefix when used as arguments or set by igraph_options(). The edge parameters:

color The color of the edges, see the `color` vertex parameter for the possible values. By default this parameter is `darkgrey`.

width The width of the edges. The default value is 1.

arrow.size The size of the arrows. The default value is 1.

arrow.width The width of the arrows. The default value is 1.

lty The line type for the edges. Almost the same format is accepted as for the standard graphics `graphics::par()`, 0 and "blank" mean no edges, 1 and "solid" are for solid lines, the other possible values are: 2 ("dashed"), 3 ("dotted"), 4 ("dotdash"), 5 ("longdash"), 6 ("twodash").

`tkplot()` also accepts standard Tk line type strings, it does not however support "blank" lines, instead of type '0' type '1', ie. solid lines will be drawn.

This argument is ignored for `rglplot()`.

The default value is type 1, a solid line.

label The edge labels. They will be converted to character. Specify `NA` to omit edge labels.

Edge labels are omitted by default.

label.family Font family of the edge labels. See the vertex parameter with the same name for the details.

label.font The font for the edge labels. See the corresponding vertex parameter discussed earlier for details.

label.cex The font size for the edge labels, see the corresponding vertex parameter for details.

label.color The color of the edge labels, see the `color` vertex parameters on how to specify colors.

label.x The horizontal `NA` elements will be replaced by automatically calculated coordinates. If `NULL`, then all edge horizontal coordinates are calculated automatically. This parameter is only supported by `plot.igraph`.

label.y The same as `label.x`, but for vertical coordinates.

curved Specifies whether to draw curved edges, or not. This can be a logical or a numeric vector or scalar.

First the vector is replicated to have the same length as the number of edges in the graph. Then it is interpreted for each edge separately. A numeric value specifies the curvature of the edge; zero curvature means straight edges, negative values means the edge bends clockwise, positive values the opposite. `TRUE` means curvature 0.5, `FALSE` means curvature zero.

By default the vector specifying the curvature is calculated via a call to the `curve_multiple()` function. This function makes sure that multiple edges are curved and are all visible. This parameter is ignored for loop edges.

The default value is `FALSE`.

This parameter is currently ignored by `rglplot()`.

arrow.mode This parameter can be used to specify for which edges should arrows be drawn. If this parameter is given by the user (in either of the three ways) then it specifies which edges will have forward, backward arrows, or both, or no arrows at all. As usual, this parameter can be a vector or a scalar value. It can be an integer or character type. If it is integer then 0 means no arrows, 1 means backward arrows, 2 is for forward arrows and 3 for both. If it is a character vector then "<" and "<-" specify backward, ">" and "->" forward arrows and "<>" and "<->" stands for both arrows. All other values mean no arrows, perhaps you should use "-" or "–" to specify no arrows.

Hint: this parameter can be used as a 'cheap' solution for drawing "mixed" graphs: graphs in which some edges are directed some are not. If you want do this, then please create a *directed* graph, because as of version 0.4 the vertex pairs in the edge lists can be swapped in undirected graphs.

By default, no arrows will be drawn for undirected graphs, and for directed graphs, an arrow will be drawn for each edge, according to its direction. This is not very surprising, it is the expected behavior.

loop.angle Gives the angle in radians for plotting loop edges. See the `label.dist` vertex parameter to see how this is interpreted.

The default value is NULL. This means that the loop edges will be drawn automatically in the largest gap possible.

loop.angle2 Gives the second angle in radians for plotting loop edges. This is only used in 3D, `loop.angle` is enough in 2D.

The default value is 0.

Other parameters:

layout Either a function or a numeric matrix. It specifies how the vertices will be placed on the plot.

If it is a numeric matrix, then the matrix has to have one line for each vertex, specifying its coordinates. The matrix should have at least two columns, for the `x` and `y` coordinates, and it can also have third column, this will be the `z` coordinate for 3D plots and it is ignored for 2D plots.

If a two column matrix is given for the 3D plotting function `rglplot()` then the third column is assumed to be 1 for each vertex.

If `layout` is a function, this function will be called with the `graph` as the single parameter to determine the actual coordinates. The function should return a matrix with two or three columns. For the 2D plots the third column is ignored. The default value is `layout_nicely`, a smart function that chooses a layout based on the graph.

margin The amount of empty space below, over, at the left and right of the plot, it is a numeric vector of length four. Usually values between 0 and 0.5 are meaningful, but negative values are also possible, that will make the plot zoom in to a part of the graph. If it is shorter than four then it is recycled. `rglplot()` does not support this parameter, as it can zoom in and out the graph in a more flexible way. Its default value is 0.

palette The color palette to use for vertex color. The default is `categorical_pal`, which is a color-blind friendly categorical palette. See its manual page for details and other palettes. This parameters is only supported by `plot`, and not by `tkplot` and `rglplot`.

rescale Logical constant, whether to rescale the coordinates to the `[-1,1]x[-1,1](x[-1,1])` interval. This parameter is not implemented for `tkplot`. Defaults to `TRUE`, the layout will be rescaled.

asp A numeric constant, it gives the `asp` parameter for `plot()`, the aspect ratio. Supply 0 here if you don't want to give an aspect ratio. It is ignored by `tkplot` and `rglplot`. Defaults to 1.

frame Boolean, whether to plot a frame around the graph. It is ignored by `tkplot` and `rglplot`. Defaults to `FALSE`.

main Overall title for the main plot. The default is empty if the `annotate.plot` igraph option is `FALSE`, and the graph's `name` attribute otherwise. See the same argument of the base `plot` function. Only supported by `plot`.

sub Subtitle of the main plot, the default is empty. Only supported by `plot`.

xlab Title for the x axis, the default is empty if the `annotate.plot` igraph option is `FALSE`, and the number of vertices and edges, if it is `TRUE`. Only supported by `plot`.

ylab Title for the y axis, the default is empty. Only supported by `plot`.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

`plot.igraph()`, `tkplot()`, `rglplot()`, `igraph_options()`

Examples

```
## Not run:

# plotting a simple ring graph, all default parameters, except the layout
g <- make_ring(10)
g$layout <- layout_in_circle
plot(g)
tkplot(g)
rglplot(g)

# plotting a random graph, set the parameters in the command arguments
g <- barabasi.game(100)
plot(g,
  layout = layout_with_fr, vertex.size = 4,
  vertex.label.dist = 0.5, vertex.color = "red", edge.arrow.size = 0.5
)

# plot a random graph, different color for each component
g <- sample_gnp(100, 1 / 100)
comps <- components(g)$membership
colbar <- rainbow(max(comps) + 1)
V(g)$color <- colbar[comps + 1]
plot(g, layout = layout_with_fr, vertex.size = 5, vertex.label = NA)

# plot communities in a graph
g <- make_full_graph(5) %du% make_full_graph(5) %du% make_full_graph(5)
g <- add_edges(g, c(1, 6, 1, 11, 6, 11))
com <- cluster_spinglass(g, spins = 5)
V(g)$color <- com$membership + 1
g <- set_graph_attr(g, "layout", layout_with_kk(g))
plot(g, vertex.label.dist = 1.5)

# draw a bunch of trees, fix layout
igraph_options(plot.layout = layout_as_tree)
plot(make_tree(20, 2))
plot(make_tree(50, 3), vertex.size = 3, vertex.label = NA)
tkplot(make_tree(50, 2, mode = "undirected"),
  vertex.size = 10,
```

```
  vertex.color = "green"
)

# use relative scaling instead of absolute
g <- make_famous_graph("Zachary")
igraph_options(plot.layout = layout_nicely)
plot(g, vertex.size = degree(g))
plot(g, vertex.size = degree(g), size.scaling = TRUE)
plot(g, vertex.size = degree(g), size.scaling = TRUE, relative.size = c(0.05, 0.1))

## End(Not run)
```

---

plot.igraph *Plotting of graphs*

Description

plot.igraph() is able to plot graphs to any R device. It is the non-interactive companion of the tkplot() function.

Usage

```
## S3 method for class 'igraph'
plot(
  x,
  axes = FALSE,
  add = FALSE,
  xlim = NULL,
  ylim = NULL,
  mark.groups = list(),
  mark.shape = 1/2,
  mark.col = rainbow(length(mark.groups), alpha = 0.3),
  mark.border = rainbow(length(mark.groups), alpha = 1),
  mark.expand = 15,
  mark.lwd = 1,
  loop.size = 1,
  ...
)
```

Arguments

x	The graph to plot.
axes	Logical, whether to plot axes, defaults to FALSE.
add	Logical scalar, whether to add the plot to the current device, or delete the device's current contents first.
xlim	The limits for the horizontal axis, it is unlikely that you want to modify this.
ylim	The limits for the vertical axis, it is unlikely that you want to modify this.

mark.groups	A list of vertex id vectors. It is interpreted as a set of vertex groups. Each vertex group is highlighted, by plotting a colored smoothed polygon around and "under" it. See the arguments below to control the look of the polygons.
mark.shape	A numeric scalar or vector. Controls the smoothness of the vertex group marking polygons. This is basically the 'shape' parameter of the graphics::xspline() function, its possible values are between -1 and 1. If it is a vector, then a different value is used for the different vertex groups.
mark.col	A scalar or vector giving the colors of marking the polygons, in any format accepted by graphics::xspline(); e.g. numeric color ids, symbolic color names, or colors in RGB.
mark.border	A scalar or vector giving the colors of the borders of the vertex group marking polygons. If it is NA, then no border is drawn.
mark.expand	A numeric scalar or vector, the size of the border around the marked vertex groups. It is in the same units as the vertex sizes. If a vector is given, then different values are used for the different vertex groups.
mark.lwd	A numeric scalar or vector, the linewidth of the border around the marked vertex groups. If a vector is given, then different values are used for the different vertex groups.
loop.size	A numeric scalar that allows the user to scale the loop edges of the network. The default loop size is 1. Larger values will produce larger loops.
...	Additional plotting parameters. See igraph.plotting for the complete list.

Details

One convenient way to plot graphs is to plot with tkplot() first, handtune the placement of the vertices, query the coordinates by the tk_coords() function and use them with plot() to plot the graph to any R device.

Value

Returns NULL, invisibly.

Related documentation in the C library

get_edgelist(), incident(), edges(), vcount(), is_loop(), is_directed(), convex_hull_2d(), ecount(), get_eids()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

layout() for different layouts, igraph.plotting for the detailed description of the plotting parameters and tkplot() and rglplot() for other graph plotting functions.

Other plot: rglplot()

Examples

```
g <- make_ring(10)
plot(g, layout = layout_with_kk, vertex.color = "green")
```

`plot.sir` *Plotting the results on multiple SIR model runs*

Description

This function can conveniently plot the results of multiple SIR model simulations.

Usage

```
## S3 method for class 'sir'
plot(
  x,
  comp = c("NI", "NS", "NR"),
  median = TRUE,
  quantiles = c(0.1, 0.9),
  color = NULL,
  median_color = NULL,
  quantile_color = NULL,
  lwd.median = 2,
  lwd.quantile = 2,
  lty.quantile = 3,
  xlim = NULL,
  ylim = NULL,
  xlab = "Time",
  ylab = NULL,
  ...
)
```

Arguments

`x`	The output of the SIR simulation, coming from the `sir()` function.
`comp`	Character scalar, which component to plot. Either 'NI' (infected, default), 'NS' (susceptible) or 'NR' (recovered).
`median`	Logical scalar, whether to plot the (binned) median.
`quantiles`	A vector of (binned) quantiles to plot.
`color`	Color of the individual simulation curves.
`median_color`	Color of the median curve.
`quantile_color`	Color(s) of the quantile curves. (It is recycled if needed and non-needed entries are ignored if too long.)
`lwd.median`	Line width of the median.

lwd.quantile	Line width of the quantile curves.
lty.quantile	Line type of the quantile curves.
xlim	The x limits, a two-element numeric vector. If NULL, then it is calculated from the data.
ylim	The y limits, a two-element numeric vector. If NULL, then it is calculated from the data.
xlab	The x label.
ylab	The y label. If NULL then it is automatically added based on the comp argument.
...	Additional arguments are passed to plot(), that is run before any of the curves are added, to create the figure.

Details

The number of susceptible/infected/recovered individuals is plotted over time, for multiple simulations.

Value

Nothing.

Author(s)

Eric Kolaczyk (https://kolaczyk.github.io/) and Gabor Csardi <csardi.gabor@gmail.com>.

References

Bailey, Norman T. J. (1975). The mathematical theory of infectious diseases and its applications (2nd ed.). London: Griffin.

See Also

sir() for running the actual simulation.

Processes on graphs: time_bins()

Examples

```
g <- sample_gnm(100, 100)
sm <- sir(g, beta = 5, gamma = 1)
plot(sm)
```

plot_dendrogram *Community structure dendrogram plots*

Description

Plot a hierarchical community structure as a dendrogram.

Usage

```
plot_dendrogram(x, mode = igraph_opt("dend.plot.type"), ...)

## S3 method for class 'communities'
plot_dendrogram(
  x,
  mode = igraph_opt("dend.plot.type"),
  ...,
  use.modularity = FALSE,
  palette = categorical_pal(8)
)
```

Arguments

x	An object containing the community structure of a graph. See communities() for details.
mode	Which dendrogram plotting function to use. See details below.
...	Additional arguments to supply to the dendrogram plotting function.
use.modularity	Logical scalar, whether to use the modularity values to define the height of the branches.
palette	The color palette to use for colored plots.

Details

plot_dendrogram() supports three different plotting functions, selected via the mode argument. By default the plotting function is taken from the dend.plot.type igraph option, and it has for possible values:

- auto Choose automatically between the plotting functions. As plot.phylo is the most sophisticated, that is choosen, whenever the ape package is available. Otherwise plot.hclust is used.
- phylo Use plot.phylo from the ape package.
- hclust Use plot.hclust from the stats package.
- dendrogram Use plot.dendrogram from the stats package.

The different plotting functions take different sets of arguments. When using plot.phylo (mode="phylo"), we have the following syntax:

```
plot_dendrogram(x, mode="phylo", colbar = palette(),
        edge.color = NULL, use.edge.length = FALSE, \dots)
```

The extra arguments not documented above:

- `colbar` Color bar for the edges.
- `edge.color` Edge colors. If `NULL`, then the `colbar` argument is used.
- `use.edge.length` Passed to `plot.phylo`.
- `dots` Attitional arguments to pass to `plot.phylo`.

The syntax for `plot.hclust` (`mode="hclust"`):

```
plot_dendrogram(x, mode="hclust", rect = 0, colbar = palette(),
        hang = 0.01, ann = FALSE, main = "", sub = "", xlab = "",
        ylab = "", \dots)
```

The extra arguments not documented above:

- `rect` A numeric scalar, the number of groups to mark on the dendrogram. The dendrogram is cut into exactly `rect` groups and they are marked via the `rect.hclust` command. Set this to zero if you don't want to mark any groups.
- `colbar` The colors of the rectangles that mark the vertex groups via the `rect` argument.
- `hang` Where to put the leaf nodes, this corresponds to the `hang` argument of `plot.hclust`.
- `ann` Whether to annotate the plot, the `ann` argument of `plot.hclust`.
- `main` The main title of the plot, the `main` argument of `plot.hclust`.
- `sub` The sub-title of the plot, the `sub` argument of `plot.hclust`.
- `xlab` The label on the horizontal axis, passed to `plot.hclust`.
- `ylab` The label on the vertical axis, passed to `plot.hclust`.
- `dots` Attitional arguments to pass to `plot.hclust`.

The syntax for `plot.dendrogram` (`mode="dendrogram"`):

```
plot_dendrogram(x, \dots)
```

The extra arguments are simply passed to `as.dendrogram()`.

Value

Returns whatever the return value was from the plotting function, `plot.phylo`, `plot.dendrogram` or `plot.hclust`.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Community detection: as_membership(), cluster_edge_betweenness(), cluster_fast_greedy(), cluster_fluid_communities(), cluster_infomap(), cluster_label_prop(), cluster_leading_eigen(), cluster_leiden(), cluster_louvain(), cluster_optimal(), cluster_spinglass(), cluster_walktrap(), compare(), groups(), make_clusters(), membership(), modularity.igraph(), split_join_distance(), voronoi_cells()

Examples

```
karate <- make_graph("Zachary")
fc <- cluster_fast_greedy(karate)
plot_dendrogram(fc)
```

---

plot_dendrogram.igraphHRG

HRG dendrogram plot

Description

Plot a hierarchical random graph as a dendrogram.

Usage

```
## S3 method for class 'igraphHRG'
plot_dendrogram(x, mode = igraph_opt("dend.plot.type"), ...)
```

Arguments

x	An igraphHRG, a hierarchical random graph, as returned by the fit_hrg() function.
mode	Which dendrogram plotting function to use. See details below.
...	Additional arguments to supply to the dendrogram plotting function.

Details

plot_dendrogram() supports three different plotting functions, selected via the mode argument. By default the plotting function is taken from the dend.plot.type igraph option, and it has for possible values:

- auto Choose automatically between the plotting functions. As plot.phylo is the most sophisticated, that is choosen, whenever the ape package is available. Otherwise plot.hclust is used.
- phylo Use plot.phylo from the ape package.
- hclust Use plot.hclust from the stats package.
- dendrogram Use plot.dendrogram from the stats package.

The different plotting functions take different sets of arguments. When using `plot.phylo` (`mode="phylo"`), we have the following syntax:

```
plot_dendrogram(x, mode="phylo", colbar = rainbow(11, start=0.7,
        end=0.1), edge.color = NULL, use.edge.length = FALSE, \dots)
```

The extra arguments not documented above:

- `colbar` Color bar for the edges.
- `edge.color` Edge colors. If `NULL`, then the `colbar` argument is used.
- `use.edge.length` Passed to `plot.phylo`.
- `dots` Attitional arguments to pass to `plot.phylo`.

The syntax for `plot.hclust` (`mode="hclust"`):

```
plot_dendrogram(x, mode="hclust", rect = 0, colbar = rainbow(rect),
        hang = 0.01, ann = FALSE, main = "", sub = "", xlab = "",
        ylab = "", \dots)
```

The extra arguments not documented above:

- `rect` A numeric scalar, the number of groups to mark on the dendrogram. The dendrogram is cut into exactly `rect` groups and they are marked via the `rect.hclust` command. Set this to zero if you don't want to mark any groups.
- `colbar` The colors of the rectangles that mark the vertex groups via the `rect` argument.
- `hang` Where to put the leaf nodes, this corresponds to the `hang` argument of `plot.hclust`.
- `ann` Whether to annotate the plot, the `ann` argument of `plot.hclust`.
- `main` The main title of the plot, the `main` argument of `plot.hclust`.
- `sub` The sub-title of the plot, the `sub` argument of `plot.hclust`.
- `xlab` The label on the horizontal axis, passed to `plot.hclust`.
- `ylab` The label on the vertical axis, passed to `plot.hclust`.
- `dots` Attitional arguments to pass to `plot.hclust`.

The syntax for `plot.dendrogram` (`mode="dendrogram"`):

```
plot_dendrogram(x, \dots)
```

The extra arguments are simply passed to `as.dendrogram()`.

Value

Returns whatever the return value was from the plotting function, `plot.phylo`, `plot.dendrogram` or `plot.hclust`.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

Examples

```
g <- make_full_graph(5) + make_full_graph(5)
hrg <- fit_hrg(g)
plot_dendrogram(hrg)
```

---

power_centrality *Find Bonacich Power Centrality Scores of Network Positions*

Description

power_centrality() takes a graph (dat) and returns the Boncich power centralities of positions (selected by nodes). The decay rate for power contributions is specified by exponent (1 by default).

Usage

```
power_centrality(
  graph,
  nodes = V(graph),
  loops = FALSE,
  exponent = 1,
  rescale = FALSE,
  tol = 1e-07,
  sparse = TRUE
)
```

Arguments

graph	the input graph.
nodes	vertex sequence indicating which vertices are to be included in the calculation. By default, all vertices are included.
loops	boolean indicating whether or not the diagonal should be treated as valid data. Set this true if and only if the data can contain loops. loops is FALSE by default.
exponent	exponent (decay rate) for the Bonacich power centrality score; can be negative
rescale	if true, centrality scores are rescaled such that they sum to 1.
tol	tolerance for near-singularities during matrix inversion (see solve())
sparse	Logical scalar, whether to use sparse matrices for the calculation. The 'Matrix' package is required for sparse matrix support

Details

Bonacich's power centrality measure is defined by $C_{BP}(\alpha, \beta) = \alpha(\mathbf{I} - \beta\mathbf{A})^{-1}\mathbf{A}\mathbf{1}$, where β is an attenuation parameter (set here by exponent) and $\mathbf{A}$ is the graph adjacency matrix. (The coefficient α acts as a scaling parameter, and is set here (following Bonacich (1987)) such that the sum of squared scores is equal to the number of vertices. This allows 1 to be used as a reference

value for the "middle" of the centrality range.) When $\beta \rightarrow 1/\lambda_{\mathbf{A}1}$ (the reciprocal of the largest eigenvalue of $\mathbf{A}$), this is to within a constant multiple of the familiar eigenvector centrality score; for other values of β, the behavior of the measure is quite different. In particular, β gives positive and negative weight to even and odd walks, respectively, as can be seen from the series expansion $C_{BP}(\alpha, \beta) = \alpha \sum_{k=0}^{\infty} \beta^k \mathbf{A}^{k+1}\mathbf{1}$ which converges so long as $|\beta| < 1/\lambda_{\mathbf{A}1}$. The magnitude of β controls the influence of distant actors on ego's centrality score, with larger magnitudes indicating slower rates of decay. (High rates, hence, imply a greater sensitivity to edge effects.)

Interpretively, the Bonacich power measure corresponds to the notion that the power of a vertex is recursively defined by the sum of the power of its alters. The nature of the recursion involved is then controlled by the power exponent: positive values imply that vertices become more powerful as their alters become more powerful (as occurs in cooperative relations), while negative values imply that vertices become more powerful only as their alters become *weaker* (as occurs in competitive or antagonistic relations). The magnitude of the exponent indicates the tendency of the effect to decay across long walks; higher magnitudes imply slower decay. One interesting feature of this measure is its relative instability to changes in exponent magnitude (particularly in the negative case). If your theory motivates use of this measure, you should be very careful to choose a decay parameter on a non-ad hoc basis.

For directed networks, the Bonacich power measure can be understood as similar to status in the network where higher status nodes have more edges that point from them to others with status. Node A's centrality depends on the centrality of all the nodes that A points toward, and their centrality depends on the nodes they point toward, etc. Note, this means that a node with an out-degree of 0 will have a Bonacich power centrality of 0 as they do not point towards anyone. When using this with directed network it is important to think about the edge direction and what it represents.

Value

A vector, containing the centrality scores.

Warning

Singular adjacency matrices cause no end of headaches for this algorithm; thus, the routine may fail in certain cases. This will be fixed when we get a better algorithm.

Related documentation in the C library

vcount(), simplify(), degree(), get_adjacency(), get_adjacency_sparse(), edges(), get_eids(), ecount()

Note

This function was ported (i.e. copied) from the SNA package.

Author(s)

Carter T. Butts (https://www.faculty.uci.edu/profile.cfm?faculty_id=5057), ported to igraph by Gabor Csardi <csardi.gabor@gmail.com>

References

Bonacich, P. (1972). "Factoring and Weighting Approaches to Status Scores and Clique Identification." *Journal of Mathematical Sociology*, 2, 113-120.

Bonacich, P. (1987). "Power and Centrality: A Family of Measures." *American Journal of Sociology*, 92, 1170-1182.

See Also

eigen_centrality() and alpha_centrality()

Centrality measures: alpha_centrality(), authority_score(), betweenness(), closeness(), diversity(), eigen_centrality(), harmonic_centrality(), hits_scores(), page_rank(), spectrum(), strength(), subgraph_centrality()

Examples

```
# Generate some test data from Bonacich, 1987:
g.c <- make_graph(c(1, 2, 1, 3, 2, 4, 3, 5), dir = FALSE)
g.d <- make_graph(c(1, 2, 1, 3, 1, 4, 2, 5, 3, 6, 4, 7), dir = FALSE)
g.e <- make_graph(c(1, 2, 1, 3, 1, 4, 2, 5, 2, 6, 3, 7, 3, 8, 4, 9, 4, 10), dir = FALSE)
g.f <- make_graph(
  c(1, 2, 1, 3, 1, 4, 2, 5, 2, 6, 2, 7, 3, 8, 3, 9, 3, 10, 4, 11, 4, 12, 4, 13),
  dir = FALSE
)
# Compute power centrality scores
for (e in seq(-0.5, .5, by = 0.1)) {
  print(round(power_centrality(g.c, exp = e)[c(1, 2, 4)], 2))
}

for (e in seq(-0.4, .4, by = 0.1)) {
  print(round(power_centrality(g.d, exp = e)[c(1, 2, 5)], 2))
}

for (e in seq(-0.4, .4, by = 0.1)) {
  print(round(power_centrality(g.e, exp = e)[c(1, 2, 5)], 2))
}

for (e in seq(-0.4, .4, by = 0.1)) {
  print(round(power_centrality(g.f, exp = e)[c(1, 2, 5)], 2))
}
```

`predict_edges` *Predict edges based on a hierarchical random graph model*

Description

`predict_edges()` uses a hierarchical random graph model to predict missing edges from a network. This is done by sampling hierarchical models around the optimum model, proportionally to their likelihood. The MCMC sampling is stated from `hrg()`, if it is given and the `start` argument is set to `TRUE`. Otherwise a HRG is fitted to the graph first.

Usage

```
predict_edges(
  graph,
  hrg = NULL,
  start = FALSE,
  num.samples = 10000,
  num.bins = 25
)
```

Arguments

`graph`	The graph to fit the model to. Edge directions are ignored in directed graphs.
`hrg`	A hierarchical random graph model, in the form of an `igraphHRG` object. `predict_edges()` allow this to be `NULL` as well, then a HRG is fitted to the graph first, from a random starting point.
`start`	Logical, whether to start the fitting/sampling from the supplied `igraphHRG` object, or from a random starting point.
`num.samples`	Number of samples to use for consensus generation or missing edge prediction.
`num.bins`	Number of bins for the edge probabilities. Give a higher number for a more accurate prediction.

Value

A list with entries:

edges The predicted edges, in a two-column matrix of vertex ids.

prob Probabilities of these edges, according to the fitted model.

hrg The (supplied or fitted) hierarchical random graph model.

Related documentation in the C library

`hrg_predict()`, `vcount()`

References

A. Clauset, C. Moore, and M.E.J. Newman. Hierarchical structure and the prediction of missing links in networks. *Nature* 453, 98–101 (2008);

A. Clauset, C. Moore, and M.E.J. Newman. Structural Inference of Hierarchies in Networks. In E. M. Airoldi et al. (Eds.): ICML 2006 Ws, *Lecture Notes in Computer Science* 4503, 1–13. Springer-Verlag, Berlin Heidelberg (2007).

See Also

Other hierarchical random graph functions: consensus_tree(), fit_hrg(), hrg(), hrg-methods, hrg_tree(), print.igraphHRG(), print.igraphHRGConsensus(), sample_hrg()

Examples

```
## A graph with two dense groups
g <- sample_gnp(10, p = 1 / 2) + sample_gnp(10, p = 1 / 2)
hrg <- fit_hrg(g)
hrg

## The consensus tree for it
consensus_tree(g, hrg = hrg, start = TRUE)

## Prediction of missing edges
g2 <- make_full_graph(4) + (make_full_graph(4) - path(1, 2))
predict_edges(g2)
```

---

print.igraph *Print graphs to the terminal*

Description

These functions attempt to print a graph to the terminal in a human readable form.

Usage

```
## S3 method for class 'igraph'
print(
  x,
  full = igraph_opt("print.full"),
  graph.attributes = igraph_opt("print.graph.attributes"),
  vertex.attributes = igraph_opt("print.vertex.attributes"),
  edge.attributes = igraph_opt("print.edge.attributes"),
  names = TRUE,
  max.lines = igraph_opt("auto.print.lines"),
  id = igraph_opt("print.id"),
  ...
)

## S3 method for class 'igraph'
summary(object, ...)
```

Arguments

x	The graph to print.
full	Logical scalar, whether to print the graph structure itself as well.
graph.attributes	Logical constant, whether to print graph attributes.
vertex.attributes	Logical constant, whether to print vertex attributes.
edge.attributes	Logical constant, whether to print edge attributes.
names	Logical constant, whether to print symbolic vertex names (i.e. the name vertex attribute) or vertex ids.
max.lines	The maximum number of lines to use. The rest of the output will be truncated.
id	Whether to print the graph ID.
...	Additional agruments.
object	The graph of which the summary will be printed.

Details

summary.igraph prints the number of vertices, edges and whether the graph is directed.

print_all() prints the same information, and also lists the edges, and optionally graph, vertex and/or edge attributes.

print.igraph() behaves either as summary.igraph or print_all() depending on the full argument. See also the 'print.full' igraph option and igraph_opt().

The graph summary printed by summary.igraph (and print.igraph() and print_all()) consists of one or more lines. The first line contains the basic properties of the graph, and the rest contains its attributes. Here is an example, a small star graph with weighted directed edges and named vertices:

```
IGRAPH badcafe DNW- 10 9 -- In-star
+ attr: name (g/c), mode (g/c), center (g/n), name (v/c),
  weight (e/n)
```

The first line always starts with IGRAPH, showing you that the object is an igraph graph. Then a seven character code is printed, this the first seven characters of the unique id of the graph. See graph_id() for more. Then a four letter long code string is printed. The first letter distinguishes between directed ('D') and undirected ('U') graphs. The second letter is 'N' for named graphs, i.e. graphs with the name vertex attribute set. The third letter is 'W' for weighted graphs, i.e. graphs with the weight edge attribute set. The fourth letter is 'B' for bipartite graphs, i.e. for graphs with the type vertex attribute set.

This is followed by the number of vertices and edges, then two dashes.

Finally, after two dashes, the name of the graph is printed, if it has one, i.e. if the name graph attribute is set.

From the second line, the attributes of the graph are listed, separated by a comma. After the attribute names, the kind of the attribute – graph ('g'), vertex ('v') or edge ('e') – is denoted, and the type of the attribute as well, character ('c'), numeric ('n'), logical ('l'), or other ('x').

As of igraph 0.4 `print_all()` and `print.igraph()` use the `max.print` option, see `base::options()` for details.

As of igraph 1.1.1, the `str.igraph` function is defunct, use `print_all()`.

Value

All these functions return the graph invisibly.

Related documentation in the C library

`degree()`, `is_directed()`, `vcount()`, `edges()`, `ecount()`, `get_eids()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

Examples

```
g <- make_ring(10)
g
summary(g)
```

---

print.igraph.es *Print an edge sequence to the screen*

Description

For long edge sequences, the printing is truncated to fit to the screen. Use `print()` explicitly and the `full` argument to see the full sequence.

Usage

```
## S3 method for class 'igraph.es'
print(x, full = igraph_opt("print.full"), id = igraph_opt("print.id"), ...)
```

Arguments

`x`	An edge sequence.
`full`	Whether to show the full sequence, or truncate the output to the screen size.
`id`	Whether to print the graph ID.
`...`	Currently ignored.

Details

Edge sequences created with the double bracket operator are printed differently, together with all attributes of the edges in the sequence, as a table.

Value

The edge sequence, invisibly.

Related documentation in the C library

ecount(), edges(), is_directed(), get_eids(), vcount()

See Also

Other vertex and edge sequences: E(), V(), as_ids(), igraph-es-attributes, igraph-es-indexing, igraph-es-indexing2, igraph-vs-attributes, igraph-vs-indexing, igraph-vs-indexing2, print.igraph.vs()

Examples

```
# Unnamed graphs
g <- make_ring(10)
E(g)

# Named graphs
g2 <- make_ring(10) %>%
  set_vertex_attr("name", value = LETTERS[1:10])
E(g2)

# All edges in a long sequence
g3 <- make_ring(200)
E(g3)
E(g3) %>% print(full = TRUE)

# Metadata
g4 <- make_ring(10) %>%
  set_vertex_attr("name", value = LETTERS[1:10]) %>%
  set_edge_attr("weight", value = 1:10) %>%
  set_edge_attr("color", value = "green")
E(g4)
E(g4)[[]]
E(g4)[[1:5]]
```

---

print.igraph.vs *Show a vertex sequence on the screen*

Description

For long vertex sequences, the printing is truncated to fit to the screen. Use print() explicitly and the full argument to see the full sequence.

Usage

```
## S3 method for class 'igraph.vs'
print(x, full = igraph_opt("print.full"), id = igraph_opt("print.id"), ...)
```

Arguments

x	A vertex sequence.
full	Whether to show the full sequence, or truncate the output to the screen size.
id	Whether to print the graph ID.
...	These arguments are currently ignored.

Details

Vertex sequence created with the double bracket operator are printed differently, together with all attributes of the vertices in the sequence, as a table.

Value

The vertex sequence, invisibly.

Related documentation in the C library

vcount()

See Also

Other vertex and edge sequences: E(), V(), as_ids(), igraph-es-attributes, igraph-es-indexing, igraph-es-indexing2, igraph-vs-attributes, igraph-vs-indexing, igraph-vs-indexing2, print.igraph.es()

Examples

```
# Unnamed graphs
g <- make_ring(10)
V(g)

# Named graphs
g2 <- make_ring(10) %>%
  set_vertex_attr("name", value = LETTERS[1:10])
V(g2)

# All vertices in the sequence
g3 <- make_ring(1000)
V(g3)
print(V(g3), full = TRUE)

# Metadata
g4 <- make_ring(10) %>%
  set_vertex_attr("name", value = LETTERS[1:10]) %>%
  set_vertex_attr("color", value = "red")
V(g4)[[]]
V(g4)[[2:5, 7:8]]
```

print.igraphHRG *Print a hierarchical random graph model to the screen*

Description

igraphHRG objects can be printed to the screen in two forms: as a tree or as a list, depending on the type argument of the print function. By default the auto type is used, which selects tree for small graphs and simple (=list) for bigger ones. The tree format looks like this:

```
Hierarchical random graph, at level 3:
g1        p=   0
'- g15    p=0.33  1
   '- g13 p=0.88  6  3  9  4  2  10 7  5  8
'- g8     p= 0.5
   '- g16 p= 0.2  20 14 17 19 11 15 16 13
   '- g5  p=   0  12 18
```

This is a graph with 20 vertices, and the top three levels of the fitted hierarchical random graph are printed. The root node of the HRG is always vertex group #1 ('g1' in the the printout). Vertex pairs in the left subtree of g1 connect to vertices in the right subtree with probability zero, according to the fitted model. g1 has two subgroups, g15 and g8. g15 has a subgroup of a single vertex (vertex 1), and another larger subgroup that contains vertices 6, 3, etc. on lower levels, etc. The plain printing is simpler and faster to produce, but less visual:

```
Hierarchical random graph:
g1  p=0.0 -> g12 g10   g2  p=1.0 -> 7 10      g3  p=1.0 -> g18 14
g4  p=1.0 -> g17 15    g5  p=0.4 -> g15 17    g6  p=0.0 -> 1 4
g7  p=1.0 -> 11 16     g8  p=0.1 -> g9 3      g9  p=0.3 -> g11 g16
g10 p=0.2 -> g4 g5     g11 p=1.0 -> g6 5      g12 p=0.8 -> g8 8
g13 p=0.0 -> g14 9     g14 p=1.0 -> 2 6       g15 p=0.2 -> g19 18
g16 p=1.0 -> g13 g2    g17 p=0.5 -> g7 13     g18 p=1.0 -> 12 19
g19 p=0.7 -> g3 20
```

It lists the two subgroups of each internal node, in as many columns as the screen width allows.

Usage

```
## S3 method for class 'igraphHRG'
print(x, type = c("auto", "tree", "plain"), level = 3, ...)
```

Arguments

x	igraphHRG object to print.
type	How to print the dendrogram, see details below.
level	The number of top levels to print from the dendrogram.
...	Additional arguments, not used currently.

Value

The hierarchical random graph model itself, invisibly.

See Also

Other hierarchical random graph functions: consensus_tree(), fit_hrg(), hrg(), hrg-methods, hrg_tree(), predict_edges(), print.igraphHRGConsensus(), sample_hrg()

---

print.igraphHRGConsensus *Print a hierarchical random graph consensus tree to the screen*

Description

Consensus dendrograms (igraphHRGConsensus objects) are printed simply by listing the children of each internal node of the dendrogram:

```
HRG consensus tree:
g1 -> 11 12 13 14 15 16 17 18 19 20
g2 -> 1  2  3  4  5  6  7  8  9  10
g3 -> g1 g2
```

The root of the dendrogram is g3 (because it has no incoming edges), and it has two subgroups, g1 and g2.

Usage

```
## S3 method for class 'igraphHRGConsensus'
print(x, ...)
```

Arguments

x	igraphHRGConsensus object to print.
...	Ignored.

Value

The input object, invisibly, to allow method chaining.

See Also

Other hierarchical random graph functions: consensus_tree(), fit_hrg(), hrg(), hrg-methods, hrg_tree(), predict_edges(), print.igraphHRG(), sample_hrg()

`printer_callback` *Create a printer callback function*

Description

A printer callback function is a function can performs the actual printing. It has a number of subcommands, that are called by the `printer` package, in a form

```
printer_callback("subcommand", argument1, argument2, ...)
```

See the examples below.

Usage

```
printer_callback(fun)
```

Arguments

`fun`	The function to use as a printer callback function.

Details

The subcommands:

`length` The length of the data to print, the number of items, in natural units. E.g. for a list of objects, it is the number of objects.

`min_width` TODO

`width` Width of one item, if `no` items will be printed. TODO

`print` Argument: `no`. Do the actual printing, print `no` items.

`done` TODO

See Also

Other printer callbacks: `is_printer_callback()`

r_pal *The default R palette*

Description

This is the default R palette, to be able to reproduce the colors of older igraph versions. Its colors are appropriate for categories, but they are not very attractive.

Usage

```
r_pal(n)
```

Arguments

n	The number of colors to use, the maximum is eight.

Value

A character vector of color names.

See Also

Other palettes: categorical_pal(), diverging_pal(), sequential_pal()

radius *Radius of a graph*

Description

The eccentricity of a vertex is its distance from the farthest other node in the graph. The smallest eccentricity in a graph is called its radius.

Usage

```
radius(graph, ..., weights = NULL, mode = c("all", "out", "in", "total"))
```

Arguments

graph	The input graph, it can be directed or undirected.
...	These dots are for future extensions and must be empty.
weights	Possibly a numeric vector giving edge weights. If this is NULL and the graph has a weight edge attribute, then the attribute is used. If this is NA then no weights are used (even if the graph has a weight attribute). In a weighted graph, the length of a path is the sum of the weights of its constituent edges.

mode — Character constant, gives whether the shortest paths to or from the given vertices should be calculated for directed graphs. If `out` then the shortest paths *from* the vertex, if `in` then *to* it will be considered. If `all`, the default, then the graph is treated as undirected, i.e. edge directions are not taken into account. This argument is ignored for undirected graphs.

Details

The eccentricity of a vertex is calculated by measuring the shortest distance from (or to) the vertex, to (or from) all vertices in the graph, and taking the maximum.

This implementation ignores vertex pairs that are in different components. Isolated vertices have eccentricity zero.

Value

A numeric scalar, the radius of the graph.

Related documentation in the C library

`radius_dijkstra()`, `edges()`, `get_eids()`, `vcount()`, `ecount()`

References

Harary, F. Graph Theory. Reading, MA: Addison-Wesley, p. 35, 1994.

See Also

`eccentricity()` for the underlying calculations, distances for general shortest path calculations.

Other paths: `all_simple_paths()`, `diameter()`, `distance_table()`, `eccentricity()`, `graph_center()`

Examples

```
g <- make_star(10, mode = "undirected")
eccentricity(g)
radius(g)
```

---

random_walk *Random walk on a graph*

Description

`random_walk()` performs a random walk on the graph and returns the vertices that the random walk passed through. `random_edge_walk()` is the same but returns the edges that that random walk passed through.

Usage

```
random_walk(
  graph,
  start,
  steps,
  weights = NULL,
  mode = c("out", "in", "all", "total"),
  stuck = c("return", "error")
)

random_edge_walk(
  graph,
  start,
  steps,
  weights = NULL,
  mode = c("out", "in", "all", "total"),
  stuck = c("return", "error")
)
```

Arguments

graph	The input graph, might be undirected or directed.
start	The start vertex.
steps	The number of steps to make.
weights	The edge weights. Larger edge weights increase the probability that an edge is selected by the random walker. In other words, larger edge weights correspond to stronger connections. The 'weight' edge attribute is used if present. Supply 'NA' here if you want to ignore the 'weight' edge attribute.
mode	How to follow directed edges. "out" steps along the edge direction, "in" is opposite to that. "all" ignores edge directions. This argument is ignored for undirected graphs.
stuck	What to do if the random walk gets stuck. "return" returns the partial walk, "error" raises an error.

Details

Do a random walk. From the given start vertex, take the given number of steps, choosing an edge from the actual vertex uniformly randomly. Edge directions are observed in directed graphs (see the mode argument as well). Multiple and loop edges are also observed.

For igraph < 1.6.0, random_walk() counted steps differently, and returned a sequence of length steps instead of steps + 1. This has changed to improve consistency with the underlying C library.

Value

For random_walk(), a vertex sequence of length steps + 1 containing the vertices along the walk, starting with start. For random_edge_walk(), an edge sequence of length steps containing the edges along the walk.

Related documentation in the C library

random_walk(), edges(), ecount(), vcount(), get_eids()

Examples

```
## Stationary distribution of a Markov chain
g <- make_ring(10, directed = TRUE) %u%
  make_star(11, center = 11) + edge(11, 1)

ec <- eigen_centrality(g, directed = TRUE)$vector
pg <- page_rank(g, damping = 0.999)$vector
w <- random_walk(g, start = 1, steps = 10000)

## These are similar, but not exactly the same
cor(table(w), ec)

## But these are (almost) the same
cor(table(w), pg)
```

---

read_graph *Reading foreign file formats*

Description

The read_graph() function is able to read graphs in various representations from a file, or from a http connection. Various formats are supported.

Usage

```
read_graph(
  file,
 format = c("edgelist", "pajek", "ncol", "lgl", "graphml", "dimacs", "graphdb", "gml",
    "dl"),
  ...
)
```

Arguments

file	The connection to read from. This can be a local file, or a http or ftp connection. It can also be a character string with the file name or URI.
format	Character constant giving the file format. Right now edgelist, pajek, ncol, lgl, graphml, dimacs, graphdb, gml and dl are supported, the default is edgelist. As of igraph 0.4 this argument is case insensitive.
...	Additional arguments, see below.

Details

The read_graph() function may have additional arguments depending on the file format (the format argument). See the details separately for each file format, below.

Value

A graph object.

Edge list format

This format is a simple text file with numeric vertex IDs defining the edges. There is no need to have newline characters between the edges, a simple space will also do. Vertex IDs contained in the file are assumed to start at zero.

Additional arguments:

n The number of vertices in the graph. If it is smaller than or equal to the largest integer in the file, then it is ignored; so it is safe to set it to zero (the default).

directed Logical scalar, whether to create a directed graph. The default value is `TRUE`.

Pajek format

Currently igraph only supports Pajek network files, with a `.net` extension, but not Pajek project files with a `.paj` extension. Only network data is supported; permutations, hierarchies, clusters and vectors are not.

NCOL format

Additional arguments:

predef Names of the vertices in the file. If `character(0)` (the default) is given here then vertex IDs will be assigned to vertex names in the order of their appearance in the .ncol file. If it is not `character(0)` and some unknown vertex names are found in the .ncol file then new vertex ids will be assigned to them.

names Logical value, if `TRUE` (the default) the symbolic names of the vertices will be added to the graph as a vertex attribute called "name".

weights Whether to add the weights of the edges to the graph as an edge attribute called "weight". `"yes"` adds the weights (even if they are not present in the file, in this case they are assumed to be zero). `"no"` does not add any edge attribute. `"auto"` (the default) adds the attribute if and only if there is at least one explicit edge weight in the input file.

directed Whether to create a directed graph (default: `FALSE`). As this format was originally used only for undirected graphs there is no information in the file about the directedness of the graph.

GraphML format

GraphML is an XML-based file format for representing various types of graphs. Currently only the most basic import functionality is implemented in igraph: it can read GraphML files without nested graphs and hyperedges.

index Integer, specifies which graph to read from a GraphML file containing multiple graphs. Defaults to 0 for the first graph.

LGL format

The .lgl format is used by the Large Graph Layout visualization software (https://lgl.sourceforge.net), it can describe undirected optionally weighted graphs

names Logical, whether to add vertex names as a vertex attribute called "name". Default is TRUE.

weights Whether to add the weights of the edges to the graph as an edge attribute called "weight". "yes" adds the weights (even if they are not present in the file, in this case they are assumed to be zero). "no" does not add any edge attribute. "auto" (the default) adds the attribute if and only if there is at least one explicit edge weight in the input file.

directed Logical, whether to create a directed graph. Default is FALSE.

DIMACS format

This is a line-oriented text file (ASCII) format. The first character of each line defines the type of the line. If the first character is c the line is a comment line and it is ignored. There is one problem line (p in the file), it must appear before any node and arc descriptor lines. The problem line has three fields separated by spaces: the problem type (max or edge), the number of vertices, and number of edges in the graph. In MAX problems, exactly two node identification lines are expected (n), one for the source, and one for the target vertex. These have two fields: the ID of the vertex and the type of the vertex, either s (= source) or t (= target). Arc lines start with a and have three fields: the source vertex, the target vertex and the edge capacity. In EDGE problems, there may be a node line (n) for each node. It specifies the node index and an integer node label. Nodes for which no explicit label was specified will use their index as label. In EDGE problems, each edge is specified as an edge line (e).

directed Logical, whether to create a directed graph. Default is TRUE.

DL format

This is a simple textual file format used by UCINET. See http://www.analytictech.com/networks/dataentry.htm for examples. All the forms described here are supported by igraph. Vertex names and edge weights are also supported and they are added as attributes. (If an attribute handler is attached.) Note the specification does not mention whether the format is case sensitive or not. For igraph DL files are case sensitive, i.e. Larry and larry are not the same.

directed Logical, whether to create a directed graph. Default is TRUE.

GML format

GML is a quite general textual format. For the specifics of the implementation, see the linked documentation of the cClibrary.

GraphDB format

This is a binary format, used in the ARG Graph Database for isomorphism testing. For more information, see https://mivia.unisa.it/datasets/graph-database/arg-database/

directed Logical, whether to create a directed graph. Default is TRUE.

Related documentation in the C library

read_graph_pajek(), read_graph_graphml(), read_graph_gml(), read_graph_dl(), read_graph_graphdb(), vcount(), edges(), get_eids(), ecount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

write_graph()

Foreign format readers: graph_from_graphdb(), write_graph()

---

realize_bipartite_degseq

Creating a bipartite graph from two degree sequences, deterministically

---

Description

[Experimental]

Constructs a bipartite graph from the degree sequences of its partitions, if one exists. This function uses a Havel-Hakimi style construction algorithm.

Usage

```
realize_bipartite_degseq(
  degrees1,
  degrees2,
  ...,
  allowed.edge.types = c("simple", "multiple"),
  method = c("smallest", "largest", "index")
)
```

Arguments

degrees1	The degrees of the first partition.
degrees2	The degrees of the second partition.
...	These dots are for future extensions and must be empty.
allowed.edge.types	Character, specifies the types of allowed edges. "simple" allows simple graphs only (no multiple edges). "multiple" allows multiple edges.
method	Character, the method for generating the graph; see below.

Details

The 'method' argument controls in which order the vertices are selected during the course of the algorithm.

The "smallest" method selects the vertex with the smallest remaining degree, from either partition. The result is usually a graph with high negative degree assortativity. In the undirected case, this method is guaranteed to generate a connected graph, regardless of whether multi-edges are allowed, provided that a connected realization exists. This is the default method.

The "largest" method selects the vertex with the largest remaining degree. The result is usually a graph with high positive degree assortativity, and is often disconnected.

The "index" method selects the vertices in order of their index.

Value

The new graph object.

Related documentation in the C library

`realize_bipartite_degree_sequence()`, `vcount()`

See Also

`realize_degseq()` to create a not necessarily bipartite graph.

Examples

```
g <- realize_bipartite_degseq(c(3, 3, 2, 1, 1), c(2, 2, 2, 2, 2))
degree(g)
```

---

realize_degseq *Creating a graph from a given degree sequence, deterministically*

Description

It is often useful to create a graph with given vertex degrees. This function creates such a graph in a deterministic manner.

Usage

```
realize_degseq(
  out.deg,
  in.deg = NULL,
  allowed.edge.types = c("simple", "loops", "multi", "all"),
  method = c("smallest", "largest", "index")
)
```

Arguments

`out.deg`	Numeric vector, the sequence of degrees (for undirected graphs) or out-degrees (for directed graphs). For undirected graphs its sum should be even. For directed graphs its sum should be the same as the sum of `in.deg`.
`in.deg`	For directed graph, the in-degree sequence. By default this is `NULL` and an undirected graph is created.
`allowed.edge.types`	Character, specifies the types of allowed edges. "simple" allows simple graphs only (no loops, no multiple edges). "multiple" allows multiple edges but disallows loop. "loops" allows loop edges but disallows multiple edges (currently unimplemented). "all" allows all types of edges. The default is "simple".
`method`	Character, the method for generating the graph; see below.

Details

Simple undirected graphs are constructed using the Havel-Hakimi algorithm (undirected case), or the analogous Kleitman-Wang algorithm (directed case). These algorithms work by choosing an arbitrary vertex and connecting all its stubs to other vertices. This step is repeated until all degrees have been connected up.

The 'method' argument controls in which order the vertices are selected during the course of the algorithm.

The "smallest" method selects the vertex with the smallest remaining degree. The result is usually a graph with high negative degree assortativity. In the undirected case, this method is guaranteed to generate a connected graph, regardless of whether multi-edges are allowed, provided that a connected realization exists. See Horvát and Modes (2021) for details. In the directed case it tends to generate weakly connected graphs, but this is not guaranteed. This is the default method.

The "largest" method selects the vertex with the largest remaining degree. The result is usually a graph with high positive degree assortativity, and is often disconnected.

The "index" method selects the vertices in order of their index.

Value

The new graph object.

Related documentation in the C library

`realize_degree_sequence()`

References

V. Havel, Poznámka o existenci konečných grafů (A remark on the existence of finite graphs), Časopis pro pěstování matematiky 80, 477-480 (1955). https://eudml.org/doc/19050

S. L. Hakimi, On Realizability of a Set of Integers as Degrees of the Vertices of a Linear Graph, Journal of the SIAM 10, 3 (1962). doi:10.1137/0111010

D. J. Kleitman and D. L. Wang, Algorithms for Constructing Graphs and Digraphs with Given Valences and Factors, Discrete Mathematics 6, 1 (1973). doi:10.1016/0012365X(73)90037X

Sz. Horvát and C. D. Modes, Connectedness matters: construction and exact random sampling of connected networks (2021). doi:10.1088/2632072X/abced5

See Also

sample_degseq() for a randomized variant that samples from graphs with the given degree sequence.

Examples

```
g <- realize_degseq(rep(2, 100))
degree(g)
is_simple(g)

## Exponential degree distribution, with high positive assortativity.
## Loop and multiple edges are explicitly allowed.
## Note that we correct the degree sequence if its sum is odd.
degs <- sample(1:100, 100, replace = TRUE, prob = exp(-0.5 * (1:100)))
if (sum(degs) %% 2 != 0) {
  degs[1] <- degs[1] + 1
}
g4 <- realize_degseq(degs, method = "largest", allowed.edge.types = "all")
all(degree(g4) == degs)

## Power-law degree distribution, no loops allowed but multiple edges
## are okay.
## Note that we correct the degree sequence if its sum is odd.
degs <- sample(1:100, 100, replace = TRUE, prob = (1:100)^-2)
if (sum(degs) %% 2 != 0) {
  degs[1] <- degs[1] + 1
}
g5 <- realize_degseq(degs, allowed.edge.types = "multi")
all(degree(g5) == degs)
```

---

reciprocity *Reciprocity of graphs*

Description

Calculates the reciprocity of a directed graph.

Usage

```
reciprocity(graph, ignore.loops = TRUE, mode = c("default", "ratio"))
```

Arguments

graph	The graph object.
ignore.loops	Logical constant, whether to ignore loop edges.
mode	See below.

Details

The measure of reciprocity defines the proportion of mutual connections, in a directed graph. It is most commonly defined as the probability that the opposite counterpart of a directed edge is also included in the graph. Or in adjacency matrix notation: $1 - \left(\sum_{i,j} |A_{ij} - A_{ji}|\right) / \left(2 \sum_{i,j} A_{ij}\right)$. This measure is calculated if the `mode` argument is `default`.

Prior to igraph version 0.6, another measure was implemented, defined as the probability of mutual connection between a vertex pair, if we know that there is a (possibly non-mutual) connection between them. In other words, (unordered) vertex pairs are classified into three groups: (1) not-connected, (2) non-reciprocally connected, (3) reciprocally connected. The result is the size of group (3), divided by the sum of group sizes (2)+(3). This measure is calculated if `mode` is `ratio`.

Value

A numeric scalar between zero and one.

Related documentation in the C library

`reciprocity()`

Author(s)

Tamas Nepusz <ntamas@gmail.com> and Gabor Csardi <csardi.gabor@gmail.com>

See Also

Other structural.properties: `bfs()`, `component_distribution()`, `connect()`, `constraint()`, `coreness()`, `degree()`, `dfs()`, `distance_table()`, `edge_density()`, `feedback_arc_set()`, `feedback_vertex_set()`, `girth()`, `is_acyclic()`, `is_dag()`, `is_matching()`, `k_shortest_paths()`, `knn()`, `subcomponent()`, `subgraph()`, `topo_sort()`, `transitivity()`, `unfold_tree()`, `which_multiple()`, `which_mutual()`

Examples

```
g <- sample_gnp(20, 5 / 20, directed = TRUE)
reciprocity(g)
```

---

rep.igraph *Replicate a graph multiple times*

Description

The new graph will contain the input graph the given number of times, as unconnected components.

Usage

```
## S3 method for class 'igraph'
rep(x, n, mark = TRUE, ...)

## S3 method for class 'igraph'
x * n
```

Arguments

x	The input graph.
n	Number of times to replicate it.
mark	Whether to mark the vertices with a `which` attribute, an integer number denoting which replication the vertex is coming from.
...	Additional arguments to satisfy S3 requirements, currently ignored.

Related documentation in the C library

vcount()

See Also

Other functions for manipulating graph structure: +.igraph(), add_edges(), add_vertices(), complementer(), compose(), connect(), contract(), delete_edges(), delete_vertices(), difference(), difference.igraph(), disjoint_union(), edge(), igraph-minus, intersection(), intersection.igraph(), path(), permute(), reverse_edges(), simplify(), transitive_closure(), union(), union.igraph(), vertex()

Examples

```
rings <- make_ring(5) * 5
```

---

rev.igraph.es *Reverse the order in an edge sequence*

Description

Reverse the order in an edge sequence

Usage

```
## S3 method for class 'igraph.es'
rev(x)
```

Arguments

x	The edge sequence to reverse.

Value

The reversed edge sequence.

See Also

Other vertex and edge sequence operations: c.igraph.es(), c.igraph.vs(), difference.igraph.es(), difference.igraph.vs(), igraph-es-indexing, igraph-es-indexing2, igraph-vs-indexing, igraph-vs-indexing2, intersection.igraph.es(), intersection.igraph.vs(), rev.igraph.vs(), union.igraph.es(), union.igraph.vs(), unique.igraph.es(), unique.igraph.vs()

Examples

```
g <- make_(ring(10), with_vertex_(name = LETTERS[1:10]))
E(g)
E(g) %>% rev()
```

---

rev.igraph.vs *Reverse the order in a vertex sequence*

Description

Reverse the order in a vertex sequence

Usage

```
## S3 method for class 'igraph.vs'
rev(x)
```

Arguments

x	The vertex sequence to reverse.

Value

The reversed vertex sequence.

See Also

Other vertex and edge sequence operations: c.igraph.es(), c.igraph.vs(), difference.igraph.es(), difference.igraph.vs(), igraph-es-indexing, igraph-es-indexing2, igraph-vs-indexing, igraph-vs-indexing2, intersection.igraph.es(), intersection.igraph.vs(), rev.igraph.es(), union.igraph.es(), union.igraph.vs(), unique.igraph.es(), unique.igraph.vs()

Examples

```
g <- make_(ring(10), with_vertex_(name = LETTERS[1:10]))
V(g) %>% rev()
```

reverse_edges *Reverse edges in a graph*

Description

The new graph will contain the same vertices, edges and attributes as the original graph, except that the direction of the edges selected by their edge IDs in the `eids` argument will be reversed. When reversing all edges, this operation is also known as graph transpose.

Usage

```
reverse_edges(graph, eids = E(graph))

## S3 method for class 'igraph'
t(x)
```

Arguments

`graph`	The input graph.
`eids`	The edge IDs of the edges to reverse.
`x`	The input graph.

Value

The result graph where the direction of the edges with the given IDs are reversed

Related documentation in the C library

`reverse_edges()`, `edges()`, `get_eids()`, `vcount()`, `ecount()`

See Also

Other functions for manipulating graph structure: `+.igraph()`, `add_edges()`, `add_vertices()`, `complementer()`, `compose()`, `connect()`, `contract()`, `delete_edges()`, `delete_vertices()`, `difference()`, `difference.igraph()`, `disjoint_union()`, `edge()`, `igraph-minus`, `intersection()`, `intersection.igraph()`, `path()`, `permute()`, `rep.igraph()`, `simplify()`, `transitive_closure()`, `union()`, `union.igraph()`, `vertex()`

Examples

```
g <- make_graph(~ 1 -+ 2, 2 -+ 3, 3 -+ 4)
reverse_edges(g, 2)
```

rewire *Rewiring edges of a graph*

Description

See the links below for the implemented rewiring methods.

Usage

```
rewire(graph, with)
```

Arguments

graph	The graph to rewire
with	A function call to one of the rewiring methods, see details below.

Value

The rewired graph.

See Also

Other rewiring functions: each_edge(), keeping_degseq()

Examples

```
g <- make_ring(10)
g %>%
  rewire(each_edge(p = .1, loops = FALSE)) %>%
  plot(layout = layout_in_circle)
print_all(rewire(g, with = keeping_degseq(niter = vcount(g) * 10)))
```

rglplot *3D plotting of graphs with OpenGL*

Description

Using the rgl package, rglplot() plots a graph in 3D. The plot can be zoomed, rotated, shifted, etc. but the coordinates of the vertices is fixed.

Usage

```
rglplot(x, ...)
```

Arguments

x	The graph to plot.
...	Additional arguments, see igraph.plotting for the details

Details

Note that `rglplot()` is considered to be highly experimental. It is not very useful either. See igraph.plotting for the possible arguments.

Value

`NULL`, invisibly.

Related documentation in the C library

`get_edgelist()`, `is_directed()`, `vcount()`, `edges()`, `get_eids()`, `ecount()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

igraph.plotting, `plot.igraph()` for the 2D version, `tkplot()` for interactive graph drawing in 2D.

Other plot: `plot.igraph()`

Examples

```
g <- make_lattice(c(5, 5, 5))
coords <- layout_with_fr(g, dim = 3)


rglplot(g, layout = coords)
```

---

running_mean *Running mean of a time series*

Description

`running_mean()` calculates the running mean in a vector with the given bin width.

Usage

```
running_mean(v, binwidth)
```

Arguments

v	The numeric vector.
binwidth	Numeric constant, the size of the bin, should be meaningful, i.e. smaller than the length of v.

Details

The running mean of v is a w vector of length length(v)-binwidth+1. The first element of w id the average of the first binwidth elements of v, the second element of w is the average of elements 2:(binwidth+1), etc.

Value

A numeric vector of length length(v)-binwidth+1

Related documentation in the C library

running_mean()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Other other: convex_hull(), sample_seq()

Examples

```
running_mean(1:100, 10)
```

---

sample_ *Sample from a random graph model*

Description

Generic function for sampling from network models.

Usage

```
sample_(...)
```

Arguments

...	Parameters, see details below.

Details

sample_() is a generic function for creating graphs. For every graph constructor in igraph that has a sample_ prefix, there is a corresponding function without the prefix: e.g. for sample_pa() there is also pa(), etc.

The same is true for the deterministic graph samplers, i.e. for each constructor with a make_ prefix, there is a corresponding function without that prefix.

These shorter forms can be used together with sample_(). The advantage of this form is that the user can specify constructor modifiers which work with all constructors. E.g. the with_vertex_() modifier adds vertex attributes to the newly created graphs.

See the examples and the various constructor modifiers below.

Related documentation in the C library

simplify(), vcount(), edges(), get_eids(), ecount()

See Also

Random graph models (games): bipartite_gnm(), erdos.renyi.game(), sample_bipartite(), sample_chung_lu(), sample_correlated_gnp(), sample_correlated_gnp_pair(), sample_degseq(), sample_dot_product(), sample_fitness(), sample_fitness_pl(), sample_forestfire(), sample_gnm(), sample_gnp(), sample_grg(), sample_growing(), sample_hierarchical_sbm(), sample_islands(), sample_k_regular(), sample_last_cit(), sample_pa(), sample_pa_age(), sample_pref(), sample_sbm(), sample_smallworld(), sample_traits_callaway(), sample_tree()

Constructor modifiers (and related functions): make_(), simplified(), with_edge_(), with_graph_(), with_vertex_(), without_attr(), without_loops(), without_multiples()

Examples

```
pref_matrix <- cbind(c(0.8, 0.1), c(0.1, 0.7))
blocky <- sample_(sbm(
  n = 20, pref.matrix = pref_matrix,
  block.sizes = c(10, 10)
))

blocky2 <- pref_matrix %>%
  sample_sbm(n = 20, block.sizes = c(10, 10))
```

---

sample_bipartite *Bipartite random graphs*

Description

[Deprecated] Generate bipartite graphs using the Erdős-Rényi model. Use sample_bipartite_gnm() and sample_bipartite_gnp() instead.

Usage

```
sample_bipartite(
  n1,
  n2,
  type = c("gnp", "gnm"),
  p,
  m,
  directed = FALSE,
  mode = c("out", "in", "all")
)

bipartite(..., type = NULL)
```

Arguments

n1	Integer scalar, the number of bottom vertices.
n2	Integer scalar, the number of top vertices.
type	Character scalar, the type of the graph, 'gnp' creates a $G(n, p)$ graph, 'gnm' creates a $G(n, m)$ graph. See details below.
p	Real scalar, connection probability for $G(n, p)$ graphs. Should not be given for $G(n, m)$ graphs.
m	Integer scalar, the number of edges for $G(n, m)$ graphs. Should not be given for $G(n, p)$ graphs.
directed	Logical scalar, whether to create a directed graph. See also the mode argument.
mode	Character scalar, specifies how to direct the edges in directed graphs. If it is 'out', then directed edges point from bottom vertices to top vertices. If it is 'in', edges point from top vertices to bottom vertices. 'out' and 'in' do not generate mutual edges. If this argument is 'all', then each edge direction is considered independently and mutual edges might be generated. This argument is ignored for undirected graphs.
...	Passed to sample_bipartite().

Value

A bipartite igraph graph.

Related documentation in the C library

bipartite_game_gnm(), bipartite_game_gnp(), vcount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Random graph models (games): bipartite_gnm(), erdos.renyi.game(), sample_(), sample_chung_lu(), sample_correlated_gnp(), sample_correlated_gnp_pair(), sample_degseq(), sample_dot_product(), sample_fitness(), sample_fitness_pl(), sample_forestfire(), sample_gnm(), sample_gnp(), sample_grg(), sample_growing(), sample_hierarchical_sbm(), sample_islands(), sample_k_regular(), sample_last_cit(), sample_pa(), sample_pa_age(), sample_pref(), sample_sbm(), sample_smallworld(), sample_traits_callaway(), sample_tree()

Examples

```
## empty graph
sample_bipartite(10, 5, p = 0)

## full graph
sample_bipartite(10, 5, p = 1)

## random bipartite graph
sample_bipartite(10, 5, p = .1)

## directed bipartite graph, G(n,m)
sample_bipartite(10, 5, type = "Gnm", m = 20, directed = TRUE, mode = "all")
```

---

sample_chung_lu *Random graph with given expected degrees*

Description

[Experimental]

The Chung-Lu model is useful for generating random graphs with fixed expected degrees. This function implements both the original model of Chung and Lu, as well as some additional variants with useful properties.

Usage

```
sample_chung_lu(
  out.weights,
  in.weights = NULL,
  ...,
  loops = TRUE,
  variant = c("original", "maxent", "nr")
)

chung_lu(
  out.weights,
  in.weights = NULL,
  ...,
```

```
  loops = TRUE,
  variant = c("original", "maxent", "nr")
)
```

Arguments

`out.weights`	A vector of non-negative vertex weights (or out-weights). In sparse graphs, these will be approximately equal to the expected (out-)degrees.
`in.weights`	A vector of non-negative in-weights, approximately equal to the expected in-degrees in sparse graphs. May be set to `NULL`, in which case undirected graphs are generated.
`...`	These dots are for future extensions and must be empty.
`loops`	Logical, whether to allow the creation of self-loops. Since vertex pairs are connected independently, setting this to `FALSE` is equivalent to simply discarding self-loops from an existing loopy Chung-Lu graph.
`variant`	The model variant to sample from, with different definitions of the connection probability between vertices i and j. Given $q_{ij} = \frac{w_i w_j}{S}$, the following formulations are available: **"original"** the original Chung-Lu model, $p_{ij} = \min(q_{ij}, 1)$. **"maxent"** maximum entropy model with fixed expected degrees, $p_{ij} = \frac{q_{ij}}{1+q_{ij}}$. **"nr"** Norros and Reittu's model, $p_{ij} = 1 - \exp(-q_{ij})$.

Details

In the original Chung-Lu model, each pair of vertices i and j is connected with independent probability

$$p_{ij} = \frac{w_i w_j}{S},$$

where w_i is a weight associated with vertex i and

$$S = \sum_k w_k$$

is the sum of weights. In the directed variant, vertices have both out-weights, w^{out}, and in-weights, w^{in}, with equal sums,

$$S = \sum_k w_k^{\text{out}} = \sum_k w_k^{\text{in}}.$$

The connection probability between i and j is

$$p_{ij} = \frac{w_i^{\text{out}} w_j^{\text{in}}}{S}.$$

This model is commonly used to create random graphs with a fixed *expected* degree sequence. The expected degree of vertex i is approximately equal to the weight w_i. Specifically, if the graph is directed and self-loops are allowed, then the expected out- and in-degrees are precisely w^{out} and w^{in}. If self-loops are disallowed, then the expected out- and in-degrees are $\frac{w^{\text{out}}(S-w^{\text{in}})}{S}$ and $\frac{w^{\text{in}}(S-w^{\text{out}})}{S}$, respectively. If the graph is undirected, then the expected degrees with and without self-loops are $\frac{w(S+w)}{S}$ and $\frac{w(S-w)}{S}$, respectively.

A limitation of the original Chung-Lu model is that when some of the weights are large, the formula for p_{ij} yields values larger than 1. Chung and Lu's original paper excludes the use of such weights. When $p_{ij} > 1$, this function simply issues a warning and creates a connection between i and j. However, in this case the expected degrees will no longer relate to the weights in the manner stated above. Thus, the original Chung-Lu model cannot produce certain (large) expected degrees.

To overcome this limitation, this function implements additional variants of the model, with modified expressions for the connection probability p_{ij} between vertices i and j. Let $q_{ij} = \frac{w_i w_j}{S}$, or $q_{ij} = \frac{w_i^{\text{out}} w_j^{\text{in}}}{S}$ in the directed case. All model variants become equivalent in the limit of sparse graphs where q_{ij} approaches zero. In the original Chung-Lu model, selectable by setting `variant` to "original", $p_{ij} = \min(q_{ij}, 1)$. The "maxent" variant, sometimes referred to as the generalized random graph, uses $p_{ij} = \frac{q_{ij}}{1+q_{ij}}$, and is equivalent to a maximum entropy model (i.e., exponential random graph model) with a constraint on expected degrees; see Park and Newman (2004), Section B, setting $\exp(-\Theta_{ij}) = \frac{w_i w_j}{S}$. This model is also discussed by Britton, Deijfen, and Martin-Löf (2006). By virtue of being a degree-constrained maximum entropy model, it generates graphs with the same degree sequence with the same probability. A third variant can be requested with "nr", and uses $p_{ij} = 1 - \exp(-q_{ij})$. This is the underlying simple graph of a multigraph model introduced by Norros and Reittu (2006). For a discussion of these three model variants, see Section 16.4 of Bollobás, Janson, Riordan (2007), as well as Van Der Hofstad (2013).

Value

An igraph graph.

Related documentation in the C library

`chung_lu_game()`

References

Chung, F., and Lu, L. (2002). Connected components in a random graph with given degree sequences. Annals of Combinatorics, 6, 125-145. doi:10.1007/PL00012580

Miller, J. C., and Hagberg, A. (2011). Efficient Generation of Networks with Given Expected Degrees. doi:10.1007/9783642212864_10

Park, J., and Newman, M. E. J. (2004). Statistical mechanics of networks. Physical Review E, 70, 066117. doi:10.1103/PhysRevE.70.066117

Britton, T., Deijfen, M., and Martin-Löf, A. (2006). Generating Simple Random Graphs with Prescribed Degree Distribution. Journal of Statistical Physics, 124, 1377-1397. doi:10.1007/s10955-0069168x

Norros, I., and Reittu, H. (2006). On a conditionally Poissonian graph process. Advances in Applied Probability, 38, 59-75. doi:10.1239/aap/1143936140

Bollobás, B., Janson, S., and Riordan, O. (2007). The phase transition in inhomogeneous random graphs. Random Structures & Algorithms, 31, 3-122. doi:10.1002/rsa.20168

Van Der Hofstad, R. (2013). Critical behavior in inhomogeneous random graphs. Random Structures & Algorithms, 42, 480-508. doi:10.1002/rsa.20450

See Also

sample_fitness() implements a similar model with a sharp constraint on the number of edges. sample_degseq() samples random graphs with sharply specified degrees. sample_gnp() creates random graphs with a fixed connection probability p between all vertex pairs.

Random graph models (games): bipartite_gnm(), erdos.renyi.game(), sample_(), sample_bipartite(), sample_correlated_gnp(), sample_correlated_gnp_pair(), sample_degseq(), sample_dot_product(), sample_fitness(), sample_fitness_pl(), sample_forestfire(), sample_gnm(), sample_gnp(), sample_grg(), sample_growing(), sample_hierarchical_sbm(), sample_islands(), sample_k_regular(), sample_last_cit(), sample_pa(), sample_pa_age(), sample_pref(), sample_sbm(), sample_smallworld(), sample_traits_callaway(), sample_tree()

Examples

```
g <- sample_chung_lu(c(3, 3, 2, 2, 2, 1, 1))

rowMeans(replicate(
  100,
  degree(sample_chung_lu(c(1, 3, 2, 1), c(2, 1, 2, 2)), mode = "out")
))

rowMeans(replicate(
  100,
  degree(sample_chung_lu(c(1, 3, 2, 1), c(2, 1, 2, 2), variant = "maxent"), mode = "out")
))
```

---

sample_correlated_gnp *Generate a new random graph from a given graph by randomly adding/removing edges*

Description

Sample a new graph by perturbing the adjacency matrix of a given graph and shuffling its vertices.

Usage

```
sample_correlated_gnp(
  old.graph,
  corr,
  p = edge_density(old.graph),
  permutation = NULL
)
```

Arguments

old.graph	The original graph.
corr	A scalar in the unit interval, the target Pearson correlation between the adjacency matrices of the original and the generated graph (the adjacency matrix being used as a vector).

p — A numeric scalar, the probability of an edge between two vertices, it must in the open (0,1) interval. The default is the empirical edge density of the graph. If you are resampling an Erdős-Rényi graph and you know the original edge probability of the Erdős-Rényi model, you should supply that explicitly.

permutation — A numeric vector, a permutation vector that is applied on the vertices of the first graph, to get the second graph. If NULL, the vertices are not permuted.

Details

Please see the reference given below.

Value

An unweighted graph of the same size as old.graph such that the correlation coefficient between the entries of the two adjacency matrices is corr. Note each pair of corresponding matrix entries is a pair of correlated Bernoulli random variables.

Related documentation in the C library

correlated_game(), density()

References

Lyzinski, V., Fishkind, D. E., Priebe, C. E. (2013). Seeded graph matching for correlated Erdős-Rényi graphs. https://arxiv.org/abs/1304.7844

See Also

Random graph models (games): bipartite_gnm(), erdos.renyi.game(), sample_(), sample_bipartite(), sample_chung_lu(), sample_correlated_gnp_pair(), sample_degseq(), sample_dot_product(), sample_fitness(), sample_fitness_pl(), sample_forestfire(), sample_gnm(), sample_gnp(), sample_grg(), sample_growing(), sample_hierarchical_sbm(), sample_islands(), sample_k_regular(), sample_last_cit(), sample_pa(), sample_pa_age(), sample_pref(), sample_sbm(), sample_smallworld(), sample_traits_callaway(), sample_tree()

Examples

```
g <- sample_gnp(1000, .1)
g2 <- sample_correlated_gnp(g, corr = 0.5)
cor(as.vector(g[]), as.vector(g2[]))
g
g2
```

`sample_correlated_gnp_pair` *Sample a pair of correlated $G(n,p)$ random graphs*

Description

Sample a new graph by perturbing the adjacency matrix of a given graph and shuffling its vertices.

Usage

```
sample_correlated_gnp_pair(n, corr, p, directed = FALSE, permutation = NULL)
```

Arguments

`n`	Numeric scalar, the number of vertices for the sampled graphs.
`corr`	A scalar in the unit interval, the target Pearson correlation between the adjacency matrices of the original the generated graph (the adjacency matrix being used as a vector).
`p`	A numeric scalar, the probability of an edge between two vertices, it must in the open (0,1) interval.
`directed`	Logical scalar, whether to generate directed graphs.
`permutation`	A numeric vector, a permutation vector that is applied on the vertices of the first graph, to get the second graph. If `NULL`, the vertices are not permuted.

Details

Please see the reference given below.

Value

A list of two igraph objects, named `graph1` and `graph2`, which are two graphs whose adjacency matrix entries are correlated with `corr`.

Related documentation in the C library

`correlated_pair_game()`

References

Lyzinski, V., Fishkind, D. E., Priebe, C. E. (2013). Seeded graph matching for correlated Erdős-Rényi graphs. https://arxiv.org/abs/1304.7844

See Also

Random graph models (games): bipartite_gnm(), erdos.renyi.game(), sample_(), sample_bipartite(), sample_chung_lu(), sample_correlated_gnp(), sample_degseq(), sample_dot_product(), sample_fitness(), sample_fitness_pl(), sample_forestfire(), sample_gnm(), sample_gnp(), sample_grg(), sample_growing(), sample_hierarchical_sbm(), sample_islands(), sample_k_regular(), sample_last_cit(), sample_pa(), sample_pa_age(), sample_pref(), sample_sbm(), sample_smallworld(), sample_traits_callaway(), sample_tree()

Examples

```
gg <- sample_correlated_gnp_pair(
  n = 10, corr = .8, p = .5,
  directed = FALSE
)
gg
cor(as.vector(gg[[1]][]), as.vector(gg[[2]][]))
```

---

sample_degseq *Generate random graphs with a given degree sequence*

Description

It is often useful to create a graph with given vertex degrees. This function creates such a graph in a randomized manner.

Usage

```
sample_degseq(
  out.deg,
  in.deg = NULL,
  method = c("configuration", "vl", "fast.heur.simple", "configuration.simple",
    "edge.switching.simple")
)

degseq(..., deterministic = FALSE)
```

Arguments

out.deg	Numeric vector, the sequence of degrees (for undirected graphs) or out-degrees (for directed graphs). For undirected graphs its sum should be even. For directed graphs its sum should be the same as the sum of in.deg.
in.deg	For directed graph, the in-degree sequence. By default this is NULL and an undirected graph is created.
method	Character, the method for generating the graph. See Details.
...	Passed to realize_degseq() if 'deterministic' is true, or to sample_degseq() otherwise.
deterministic	Whether the construction should be deterministic

Details

The "configuration" method (formerly called "simple") implements the configuration model. For undirected graphs, it puts all vertex IDs in a bag such that the multiplicity of a vertex in the bag is the same as its degree. Then it draws pairs from the bag until the bag becomes empty. This method may generate both loop (self) edges and multiple edges. For directed graphs, the algorithm is basically the same, but two separate bags are used for the in- and out-degrees. Undirected graphs are generated with probability proportional to $(\prod_{i<j} A_{ij}! \prod_i A_{ii}!!)^{-1}$, where A denotes the adjacency matrix and !! denotes the double factorial. Here A is assumed to have twice the number of self-loops on its diagonal. The corresponding expression for directed graphs is $(\prod_{i,j} A_{ij}!)^{-1}$. Thus the probability of all simple graphs (which only have 0s and 1s in the adjacency matrix) is the same, while that of non-simple ones depends on their edge and self-loop multiplicities.

The "fast.heur.simple" method (formerly called "simple.no.multiple") generates simple graphs. It is similar to "configuration" but tries to avoid multiple and loop edges and restarts the generation from scratch if it gets stuck. It can generate all simple realizations of a degree sequence, but it is not guaranteed to sample them uniformly. This method is relatively fast and it will eventually succeed if the provided degree sequence is graphical, but there is no upper bound on the number of iterations.

The "configuration.simple" method (formerly called "simple.no.multiple.uniform") is identical to "configuration", but if the generated graph is not simple, it rejects it and re-starts the generation. It generates all simple graphs with the same probability.

The "vl" method samples undirected connected graphs approximately uniformly. It is a Monte Carlo method based on degree-preserving edge switches. This generator should be favoured if undirected and connected graphs are to be generated and execution time is not a concern. igraph uses the original implementation of Fabien Viger; for the algorithm, see https://web.archive.org/web/20250428012457/https://www-complexnetworks.lip6.fr/~latapy/FV/generation.html and the paper https://arxiv.org/abs/cs/0502085.

The "edge.switching.simple" is an MCMC sampler based on degree-preserving edge switches. It generates simple undirected or directed graphs.

Value

The new graph object.

Related documentation in the C library

degree_sequence_game()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

simplify() to get rid of the multiple and/or loops edges, realize_degseq() for a deterministic variant.

Random graph models (games): bipartite_gnm(), erdos.renyi.game(), sample_(), sample_bipartite(), sample_chung_lu(), sample_correlated_gnp(), sample_correlated_gnp_pair(), sample_dot_product(), sample_fitness(), sample_fitness_pl(), sample_forestfire(), sample_gnm(), sample_gnp(),

sample_grg(), sample_growing(), sample_hierarchical_sbm(), sample_islands(), sample_k_regular(), sample_last_cit(), sample_pa(), sample_pa_age(), sample_pref(), sample_sbm(), sample_smallworld(), sample_traits_callaway(), sample_tree()

Examples

```
## The simple generator
undirected_graph <- sample_degseq(rep(2, 100))
degree(undirected_graph)
is_simple(undirected_graph) # sometimes TRUE, but can be FALSE


directed_graph <- sample_degseq(1:10, 10:1)
degree(directed_graph, mode = "out")
degree(directed_graph, mode = "in")

## The vl generator
vl_graph <- sample_degseq(rep(2, 100), method = "vl")
degree(vl_graph)
is_simple(vl_graph) # always TRUE

## Exponential degree distribution
## We fix the seed as there's no guarantee
##  that randomly picked integers will form a graphical degree sequence
## (i.e. that there's a graph with these degrees)
## withr::with_seed(42, {
## exponential_degrees <- sample(1:100, 100, replace = TRUE, prob = exp(-0.5 * (1:100)))
## })
exponential_degrees <- c(
  5L, 6L, 1L, 4L, 3L, 2L, 3L, 1L, 3L, 3L, 2L, 3L, 6L, 1L, 2L,
  6L, 8L, 1L, 2L, 2L, 5L, 1L, 10L, 6L, 1L, 2L, 1L, 5L, 2L, 4L,
  3L, 4L, 1L, 3L, 1L, 4L, 1L, 1L, 5L, 2L, 1L, 2L, 1L, 8L, 2L, 7L,
  5L, 3L, 8L, 2L, 1L, 1L, 2L, 4L, 1L, 3L, 3L, 1L, 1L, 2L, 3L, 9L,
  3L, 2L, 4L, 1L, 1L, 4L, 3L, 1L, 1L, 1L, 1L, 2L, 1L, 3L, 1L, 1L,
  2L, 1L, 2L, 1L, 1L, 3L, 3L, 2L, 1L, 1L, 1L, 1L, 3L, 1L, 1L, 6L,
  6L, 3L, 1L, 2L, 3L, 2L
)
## Note, that we'd have to correct the degree sequence if its sum is odd
is_exponential_degrees_sum_odd <- (sum(exponential_degrees) %% 2 != 0)
if (is_exponential_degrees_sum_odd) {
  exponential_degrees[1] <- exponential_degrees[1] + 1
}
exp_vl_graph <- sample_degseq(exponential_degrees, method = "vl")
all(degree(exp_vl_graph) == exponential_degrees)

## An example that does not work

## withr::with_seed(11, {
## exponential_degrees <- sample(1:100, 100, replace = TRUE, prob = exp(-0.5 * (1:100)))
## })
exponential_degrees <- c(
  1L, 1L, 2L, 1L, 1L, 7L, 1L, 1L, 5L, 1L, 1L, 2L, 5L, 4L, 3L,
  2L, 2L, 1L, 1L, 2L, 1L, 3L, 1L, 1L, 1L, 2L, 2L, 1L, 1L, 2L, 2L,
```

```
  1L, 2L, 1L, 4L, 3L, 1L, 1L, 1L, 1L, 1L, 1L, 2L, 3L, 1L, 4L, 3L,
  1L, 2L, 4L, 2L, 2L, 2L, 1L, 1L, 2L, 2L, 4L, 1L, 2L, 1L, 3L, 1L,
  2L, 3L, 1L, 1L, 2L, 1L, 2L, 3L, 2L, 2L, 1L, 6L, 2L, 1L, 1L, 1L,
  1L, 1L, 2L, 2L, 1L, 4L, 2L, 1L, 3L, 4L, 1L, 1L, 3L, 1L, 2L, 4L,
  1L, 3L, 1L, 2L, 1L
)
## Note, that we'd have to correct the degree sequence if its sum is odd
is_exponential_degrees_sum_odd <- (sum(exponential_degrees) %% 2 != 0)
if (is_exponential_degrees_sum_odd) {
  exponential_degrees[1] <- exponential_degrees[1] + 1
}
exp_vl_graph <- sample_degseq(exponential_degrees, method = "vl")

## Power-law degree distribution
## We fix the seed as there's no guarantee
##  that randomly picked integers will form a graphical degree sequence
## (i.e. that there's a graph with these degrees)
## withr::with_seed(1, {
##  powerlaw_degrees <- sample(1:100, 100, replace = TRUE, prob = (1:100)^-2)
## })
powerlaw_degrees <- c(
  1L, 1L, 1L, 6L, 1L, 6L, 10L, 2L, 2L, 1L, 1L, 1L, 2L, 1L, 3L,
  1L, 2L, 43L, 1L, 3L, 9L, 1L, 2L, 1L, 1L, 1L, 1L, 1L, 4L, 1L,
  1L, 1L, 1L, 1L, 3L, 2L, 3L, 1L, 2L, 1L, 3L, 2L, 3L, 1L, 1L, 3L,
  1L, 1L, 2L, 2L, 1L, 4L, 1L, 1L, 1L, 1L, 1L, 1L, 2L, 1L, 7L, 1L,
  1L, 1L, 2L, 1L, 1L, 3L, 1L, 5L, 1L, 4L, 1L, 1L, 1L, 5L, 4L, 1L,
  3L, 13L, 1L, 2L, 1L, 1L, 2L, 1L, 2L, 1L, 1L, 1L, 1L, 1L, 2L,
  5L, 3L, 3L, 1L, 1L, 3L, 1L
)
## Note, that we correct the degree sequence if its sum is odd
is_exponential_degrees_sum_odd <- (sum(powerlaw_degrees) %% 2 != 0)
if (is_exponential_degrees_sum_odd) {
  powerlaw_degrees[1] <- powerlaw_degrees[1] + 1
}
powerlaw_vl_graph <- sample_degseq(powerlaw_degrees, method = "vl")
all(degree(powerlaw_vl_graph) == powerlaw_degrees)

## An example that does not work

## withr::with_seed(2, {
##  powerlaw_degrees <- sample(1:100, 100, replace = TRUE, prob = (1:100)^-2)
## })
powerlaw_degrees <- c(
  1L, 2L, 1L, 1L, 10L, 10L, 1L, 4L, 1L, 1L, 1L, 1L, 2L, 1L, 1L,
  4L, 21L, 1L, 1L, 1L, 2L, 1L, 4L, 1L, 1L, 1L, 1L, 1L, 14L, 1L,
  1L, 1L, 3L, 4L, 1L, 2L, 4L, 1L, 2L, 1L, 25L, 1L, 1L, 1L, 10L,
  3L, 19L, 1L, 1L, 3L, 1L, 1L, 2L, 8L, 1L, 3L, 3L, 36L, 2L, 2L,
  3L, 5L, 2L, 1L, 4L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 2L,
  1L, 4L, 1L, 1L, 1L, 2L, 1L, 1L, 1L, 4L, 18L, 1L, 2L, 1L, 21L,
  1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L
)
## Note, that we correct the degree sequence if its sum is odd
is_exponential_degrees_sum_odd <- (sum(powerlaw_degrees) %% 2 != 0)
```

```
if (is_exponential_degrees_sum_odd) {
  powerlaw_degrees[1] <- powerlaw_degrees[1] + 1
}
powerlaw_vl_graph <- sample_degseq(powerlaw_degrees, method = "vl")
all(degree(powerlaw_vl_graph) == powerlaw_degrees)
```

`sample_dirichlet` *Sample from a Dirichlet distribution*

Description

Sample finite-dimensional vectors to use as latent position vectors in random dot product graphs

Usage

```
sample_dirichlet(n, alpha)
```

Arguments

`n`	Integer scalar, the sample size.
`alpha`	Numeric vector, the vector of α parameter for the Dirichlet distribution.

Details

`sample_dirichlet()` generates samples from the Dirichlet distribution with given α parameter. The sample is drawn from `length(alpha)-1`-simplex.

Value

A `dim` (length of the `alpha` vector for `sample_dirichlet()`) times `n` matrix, whose columns are the sample vectors.

Related documentation in the C library

`sample_dirichlet()`

See Also

Other latent position vector samplers: `sample_sphere_surface()`, `sample_sphere_volume()`

Examples

```
lpvs.dir <- sample_dirichlet(n = 20, alpha = rep(1, 10))
RDP.graph.2 <- sample_dot_product(lpvs.dir)
colSums(lpvs.dir)
```

`sample_dot_product` *Generate random graphs according to the random dot product graph model*

Description

In this model, each vertex is represented by a latent position vector. Probability of an edge between two vertices are given by the dot product of their latent position vectors.

Usage

```
sample_dot_product(vecs, directed = FALSE)

dot_product(vecs, directed = FALSE)
```

Arguments

`vecs`	A numeric matrix in which each latent position vector is a column.
`directed`	A logical scalar, TRUE if the generated graph should be directed.

Details

The dot product of the latent position vectors should be in the [0,1] interval, otherwise a warning is given. For negative dot products, no edges are added; dot products that are larger than one always add an edge.

Value

An igraph graph object which is the generated random dot product graph.

Related documentation in the C library

`dot_product_game()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Christine Leigh Myers Nickel: Random dot product graphs, a model for social networks. Dissertation, Johns Hopkins University, Maryland, USA, 2006.

See Also

sample_dirichlet(), sample_sphere_surface() and sample_sphere_volume() for sampling position vectors.

Random graph models (games): bipartite_gnm(), erdos.renyi.game(), sample_(), sample_bipartite(), sample_chung_lu(), sample_correlated_gnp(), sample_correlated_gnp_pair(), sample_degseq(), sample_fitness(), sample_fitness_pl(), sample_forestfire(), sample_gnm(), sample_gnp(), sample_grg(), sample_growing(), sample_hierarchical_sbm(), sample_islands(), sample_k_regular(), sample_last_cit(), sample_pa(), sample_pa_age(), sample_pref(), sample_sbm(), sample_smallworld(), sample_traits_callaway(), sample_tree()

Examples

```
## A randomly generated  graph
lpvs <- matrix(rnorm(200), 20, 10)
lpvs <- apply(lpvs, 2, function(x) {
  return(abs(x) / sqrt(sum(x^2)))
})
g <- sample_dot_product(lpvs)
g

## Sample latent vectors from the surface of the unit sphere
lpvs2 <- sample_sphere_surface(dim = 5, n = 20)
g2 <- sample_dot_product(lpvs2)
g2
```

---

sample_fitness *Random graphs from vertex fitness scores*

Description

This function generates a non-growing random graph with edge probabilities proportional to node fitness scores.

Usage

```
sample_fitness(
  no.of.edges,
  fitness.out,
  fitness.in = NULL,
  loops = FALSE,
  multiple = FALSE
)
```

Arguments

no.of.edges	The number of edges in the generated graph.
fitness.out	A numeric vector containing the fitness of each vertex. For directed graphs, this specifies the out-fitness of each vertex.

`fitness.in`	If `NULL` (the default), the generated graph will be undirected. If not `NULL`, then it should be a numeric vector and it specifies the in-fitness of each vertex. If this argument is not `NULL`, then a directed graph is generated, otherwise an undirected one.
`loops`	Logical scalar, whether to allow loop edges in the graph.
`multiple`	Logical scalar, whether to allow multiple edges in the graph.

Details

This game generates a directed or undirected random graph where the probability of an edge between vertices i and j depends on the fitness scores of the two vertices involved. For undirected graphs, each vertex has a single fitness score. For directed graphs, each vertex has an out- and an in-fitness, and the probability of an edge from i to j depends on the out-fitness of vertex i and the in-fitness of vertex j.

The generation process goes as follows. We start from N disconnected nodes (where N is given by the length of the fitness vector). Then we randomly select two vertices i and j, with probabilities proportional to their fitnesses. (When the generated graph is directed, i is selected according to the out-fitnesses and j is selected according to the in-fitnesses). If the vertices are not connected yet (or if multiple edges are allowed), we connect them; otherwise we select a new pair. This is repeated until the desired number of links are created.

It can be shown that the *expected* degree of each vertex will be proportional to its fitness, although the actual, observed degree will not be. If you need to generate a graph with an exact degree sequence, consider `sample_degseq()` instead.

This model is commonly used to generate static scale-free networks. To achieve this, you have to draw the fitness scores from the desired power-law distribution. Alternatively, you may use `sample_fitness_pl()` which generates the fitnesses for you with a given exponent.

Value

An igraph graph, directed or undirected.

Related documentation in the C library

`static_fitness_game()`

Author(s)

Tamas Nepusz <ntamas@gmail.com>

References

Goh K-I, Kahng B, Kim D: Universal behaviour of load distribution in scale-free networks. *Phys Rev Lett* 87(27):278701, 2001.

See Also

Random graph models (games): bipartite_gnm(), erdos.renyi.game(), sample_(), sample_bipartite(), sample_chung_lu(), sample_correlated_gnp(), sample_correlated_gnp_pair(), sample_degseq(), sample_dot_product(), sample_fitness_pl(), sample_forestfire(), sample_gnm(), sample_gnp(), sample_grg(), sample_growing(), sample_hierarchical_sbm(), sample_islands(), sample_k_regular(), sample_last_cit(), sample_pa(), sample_pa_age(), sample_pref(), sample_sbm(), sample_smallworld(), sample_traits_callaway(), sample_tree()

Examples

```
N <- 10000
g <- sample_fitness(5 * N, sample((1:50)^-2, N, replace = TRUE))
degree_distribution(g)
plot(degree_distribution(g, cumulative = TRUE), log = "xy")
```

---

sample_fitness_pl *Scale-free random graphs, from vertex fitness scores*

Description

This function generates a non-growing random graph with expected power-law degree distributions.

Usage

```
sample_fitness_pl(
  no.of.nodes,
  no.of.edges,
  exponent.out,
  exponent.in = -1,
  loops = FALSE,
  multiple = FALSE,
  finite.size.correction = TRUE
)
```

Arguments

no.of.nodes	The number of vertices in the generated graph.
no.of.edges	The number of edges in the generated graph.
exponent.out	Numeric scalar, the power law exponent of the degree distribution. For directed graphs, this specifies the exponent of the out-degree distribution. It must be greater than or equal to 2. If you pass Inf here, you will get back an Erdős-Rényi random network.
exponent.in	Numeric scalar. If negative, the generated graph will be undirected. If greater than or equal to 2, this argument specifies the exponent of the in-degree distribution. If non-negative but less than 2, an error will be generated.
loops	Logical scalar, whether to allow loop edges in the graph.

`multiple` Logical scalar, whether to allow multiple edges in the graph.

`finite.size.correction`
Logical scalar, whether to use the proposed finite size correction of Cho et al., see references below.

Details

This game generates a directed or undirected random graph where the degrees of vertices follow power-law distributions with prescribed exponents. For directed graphs, the exponents of the in- and out-degree distributions may be specified separately.

The game simply uses `sample_fitness()` with appropriately constructed fitness vectors. In particular, the fitness of vertex i is $i^{-\alpha}$, where $\alpha = 1/(\gamma - 1)$ and γ is the exponent given in the arguments.

To remove correlations between in- and out-degrees in case of directed graphs, the in-fitness vector will be shuffled after it has been set up and before `sample_fitness()` is called.

Note that significant finite size effects may be observed for exponents smaller than 3 in the original formulation of the game. This function provides an argument that lets you remove the finite size effects by assuming that the fitness of vertex i is $(i + i_0 - 1)^{-\alpha}$ where i_0 is a constant chosen appropriately to ensure that the maximum degree is less than the square root of the number of edges times the average degree; see the paper of Chung and Lu, and Cho et al for more details.

Value

An igraph graph, directed or undirected.

Related documentation in the C library

`static_power_law_game()`

Author(s)

Tamas Nepusz <ntamas@gmail.com>

References

Goh K-I, Kahng B, Kim D: Universal behaviour of load distribution in scale-free networks. *Phys Rev Lett* 87(27):278701, 2001.

Chung F and Lu L: Connected components in a random graph with given degree sequences. *Annals of Combinatorics* 6, 125-145, 2002.

Cho YS, Kim JS, Park J, Kahng B, Kim D: Percolation transitions in scale-free networks under the Achlioptas process. *Phys Rev Lett* 103:135702, 2009.

See Also

Random graph models (games): `bipartite_gnm()`, `erdos.renyi.game()`, `sample_()`, `sample_bipartite()`, `sample_chung_lu()`, `sample_correlated_gnp()`, `sample_correlated_gnp_pair()`, `sample_degseq()`, `sample_dot_product()`, `sample_fitness()`, `sample_forestfire()`, `sample_gnm()`, `sample_gnp()`, `sample_grg()`, `sample_growing()`, `sample_hierarchical_sbm()`, `sample_islands()`, `sample_k_regular()`,

sample_last_cit(), sample_pa(), sample_pa_age(), sample_pref(), sample_sbm(), sample_smallworld(), sample_traits_callaway(), sample_tree()

Examples

```
g <- sample_fitness_pl(10000, 30000, 2.2, 2.3)
plot(degree_distribution(g, cumulative = TRUE, mode = "out"), log = "xy")
```

---

sample_forestfire *Forest Fire Network Model*

Description

This is a growing network model, which resembles of how the forest fire spreads by igniting trees close by.

Usage

```
sample_forestfire(nodes, fw.prob, bw.factor = 1, ambs = 1, directed = TRUE)
```

Arguments

nodes	The number of vertices in the graph.
fw.prob	The forward burning probability, see details below.
bw.factor	The backward burning ratio. The backward burning probability is calculated as bw.factor*fw.prob.
ambs	The number of ambassador vertices.
directed	Logical scalar, whether to create a directed graph.

Details

The forest fire model intends to reproduce the following network characteristics, observed in real networks:

- Heavy-tailed in-degree distribution.
- Heavy-tailed out-degree distribution.
- Communities.
- Densification power-law. The network is densifying in time, according to a power-law rule.
- Shrinking diameter. The diameter of the network decreases in time.

The network is generated in the following way. One vertex is added at a time. This vertex connects to (cites) ambs vertices already present in the network, chosen uniformly random. Now, for each cited vertex v we do the following procedure:

1. We generate two random number, x and y, that are geometrically distributed with means $p/(1-p)$ and $rp(1-rp)$. (p is fw.prob, r is bw.factor.) The new vertex cites x outgoing neighbors and y incoming neighbors of v, from those which are not yet cited by the new vertex. If there are less than x or y such vertices available then we cite all of them.
2. The same procedure is applied to all the newly cited vertices.

Value

A simple graph, possibly directed if the `directed` argument is `TRUE`.

Related documentation in the C library

`forest_fire_game()`

Note

The version of the model in the published paper is incorrect in the sense that it cannot generate the kind of graphs the authors claim. A corrected version is available from https://www.cs.cmu.edu/~jure/pubs/powergrowth-tkdd.pdf, our implementation is based on this.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Jure Leskovec, Jon Kleinberg and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. *KDD '05: Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, 177–187, 2005.

See Also

`sample_pa()` for the basic preferential attachment model.

Random graph models (games): `bipartite_gnm()`, `erdos.renyi.game()`, `sample_()`, `sample_bipartite()`, `sample_chung_lu()`, `sample_correlated_gnp()`, `sample_correlated_gnp_pair()`, `sample_degseq()`, `sample_dot_product()`, `sample_fitness()`, `sample_fitness_pl()`, `sample_gnm()`, `sample_gnp()`, `sample_grg()`, `sample_growing()`, `sample_hierarchical_sbm()`, `sample_islands()`, `sample_k_regular()`, `sample_last_cit()`, `sample_pa()`, `sample_pa_age()`, `sample_pref()`, `sample_sbm()`, `sample_smallworld()`, `sample_traits_callaway()`, `sample_tree()`

Examples

```
fire <- sample_forestfire(50, fw.prob = 0.37, bw.factor = 0.32 / 0.37)
plot(fire)

g <- sample_forestfire(10000, fw.prob = 0.37, bw.factor = 0.32 / 0.37)
dd1 <- degree_distribution(g, mode = "in")
dd2 <- degree_distribution(g, mode = "out")
# The forest fire model produces graphs with a heavy tail degree distribution.
# Note that some in- or out-degrees are zero which will be excluded from the logarithmic plot.
plot(seq(along.with = dd1) - 1, dd1, log = "xy")
points(seq(along.with = dd2) - 1, dd2, col = 2, pch = 2)
```

`sample_gnm` *Generate random graphs according to the $G(n, m)$ Erdős-Rényi model*

Description

Random graph with a fixed number of edges and vertices.

Usage

```
sample_gnm(n, m, directed = FALSE, loops = FALSE)

gnm(n, m, directed = FALSE, loops = FALSE)
```

Arguments

`n`	The number of vertices in the graph.
`m`	The number of edges in the graph.
`directed`	Logical, whether the graph will be directed, defaults to `FALSE`.
`loops`	Logical, whether to add loop edges, defaults to `FALSE`.

Details

The graph has `n` vertices and `m` edges. The edges are chosen uniformly at random from the set of all vertex pairs. This set includes potential self-connections as well if the `loops` parameter is `TRUE`.

Value

A graph object.

Related documentation in the C library

`erdos_renyi_game_gnm()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Erdős, P. and Rényi, A., On random graphs, *Publicationes Mathematicae* 6, 290–297 (1959).

See Also

Random graph models (games): bipartite_gnm(), erdos.renyi.game(), sample_(), sample_bipartite(), sample_chung_lu(), sample_correlated_gnp(), sample_correlated_gnp_pair(), sample_degseq(), sample_dot_product(), sample_fitness(), sample_fitness_pl(), sample_forestfire(), sample_gnp(), sample_grg(), sample_growing(), sample_hierarchical_sbm(), sample_islands(), sample_k_regular(), sample_last_cit(), sample_pa(), sample_pa_age(), sample_pref(), sample_sbm(), sample_smallworld(), sample_traits_callaway(), sample_tree()

Examples

```
g <- sample_gnm(1000, 1000)
degree_distribution(g)
```

---

sample_gnp *Generate random graphs according to the $G(n,p)$ Erdős-Rényi model*

Description

Every possible edge is created independently with the same probability p. This model is also referred to as a Bernoulli random graph since the connectivity status of vertex pairs follows a Bernoulli distribution.

Usage

```
sample_gnp(n, p, directed = FALSE, loops = FALSE)

gnp(n, p, directed = FALSE, loops = FALSE)
```

Arguments

n	The number of vertices in the graph.
p	The probability for drawing an edge between two arbitrary vertices ($G(n,p)$ graph).
directed	Logical, whether the graph will be directed, defaults to FALSE.
loops	Logical, whether to add loop edges, defaults to FALSE.

Details

The graph has n vertices and each pair of vertices is connected with the same probability p. The loops parameter controls whether self-connections are also considered. This model effectively constrains the average number of edges, $pm_{\max}$, where $m_{\max}$ is the largest possible number of edges, which depends on whether the graph is directed or undirected and whether self-loops are allowed.

Value

A graph object.

Related documentation in the C library

erdos_renyi_game_gnp()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Erdős, P. and Rényi, A., On random graphs, *Publicationes Mathematicae* 6, 290–297 (1959).

See Also

Random graph models (games): bipartite_gnm(), erdos.renyi.game(), sample_(), sample_bipartite(), sample_chung_lu(), sample_correlated_gnp(), sample_correlated_gnp_pair(), sample_degseq(), sample_dot_product(), sample_fitness(), sample_fitness_pl(), sample_forestfire(), sample_gnm(), sample_grg(), sample_growing(), sample_hierarchical_sbm(), sample_islands(), sample_k_regular(), sample_last_cit(), sample_pa(), sample_pa_age(), sample_pref(), sample_sbm(), sample_smallworld(), sample_traits_callaway(), sample_tree()

Examples

```
# Random graph with expected mean degree of 2
g <- sample_gnp(1000, 2 / 1000)
mean(degree(g))
degree_distribution(g)

# Pick a simple graph on 6 vertices uniformly at random
plot(sample_gnp(6, 0.5))
```

---

sample_grg *Geometric random graphs*

Description

Generate a random graph based on the distance of random point on a unit square

Usage

```
sample_grg(nodes, radius, torus = FALSE, coords = FALSE)

grg(nodes, radius, torus = FALSE, coords = FALSE)
```

Arguments

nodes	The number of vertices in the graph.
radius	The radius within which the vertices will be connected by an edge.
torus	Logical constant, whether to use a torus instead of a square.
coords	Logical scalar, whether to add the positions of the vertices as vertex attributes called 'x' and 'y'.

Details

First a number of points are dropped on a unit square, these points correspond to the vertices of the graph to create. Two points will be connected with an undirected edge if they are closer to each other in Euclidean norm than a given radius. If the torus argument is TRUE then a unit area torus is used instead of a square.

Value

A graph object. If coords is TRUE then with vertex attributes 'x' and 'y'.

Related documentation in the C library

vcount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>, first version was written by Keith Briggs (https://keithbriggs.info/).

See Also

Random graph models (games): bipartite_gnm(), erdos.renyi.game(), sample_(), sample_bipartite(), sample_chung_lu(), sample_correlated_gnp(), sample_correlated_gnp_pair(), sample_degseq(), sample_dot_product(), sample_fitness(), sample_fitness_pl(), sample_forestfire(), sample_gnm(), sample_gnp(), sample_growing(), sample_hierarchical_sbm(), sample_islands(), sample_k_regular(), sample_last_cit(), sample_pa(), sample_pa_age(), sample_pref(), sample_sbm(), sample_smallworld(), sample_traits_callaway(), sample_tree()

Examples

```
g <- sample_grg(1000, 0.05, torus = FALSE)
g2 <- sample_grg(1000, 0.05, torus = TRUE)
```

sample_growing *Growing random graph generation*

Description

This function creates a random graph by simulating its stochastic evolution.

Usage

```
sample_growing(n, m = 1, ..., directed = TRUE, citation = FALSE)

growing(n, m = 1, ..., directed = TRUE, citation = FALSE)
```

Arguments

n	Numeric constant, number of vertices in the graph.
m	Numeric constant, number of edges added in each time step.
...	These dots are for future extensions and must be empty.
directed	Logical, whether to create a directed graph.
citation	Logical. If TRUE a citation graph is created, i.e. in each time step the added edges are originating from the new vertex.

Details

This is discrete time step model, in each time step a new vertex is added to the graph and m new edges are created. If citation is FALSE these edges are connecting two uniformly randomly chosen vertices, otherwise the edges are connecting new vertex to uniformly randomly chosen old vertices.

Value

A new graph object.

Related documentation in the C library

growing_random_game()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Random graph models (games): bipartite_gnm(), erdos.renyi.game(), sample_(), sample_bipartite(), sample_chung_lu(), sample_correlated_gnp(), sample_correlated_gnp_pair(), sample_degseq(), sample_dot_product(), sample_fitness(), sample_fitness_pl(), sample_forestfire(), sample_gnm(), sample_gnp(), sample_grg(), sample_hierarchical_sbm(), sample_islands(), sample_k_regular(), sample_last_cit(), sample_pa(), sample_pa_age(), sample_pref(), sample_sbm(), sample_smallworld(), sample_traits_callaway(), sample_tree()

Examples

```
g <- sample_growing(500, citation = FALSE)
g2 <- sample_growing(500, citation = TRUE)
```

`sample_hierarchical_sbm` *Sample the hierarchical stochastic block model*

Description

Sampling from a hierarchical stochastic block model of networks.

Usage

```
sample_hierarchical_sbm(n, m, rho, C, p)

hierarchical_sbm(n, m, rho, C, p)
```

Arguments

`n`	Integer scalar, the number of vertices.
`m`	Integer scalar, the number of vertices per block. `n / m` must be integer. Alternatively, an integer vector of block sizes, if not all the blocks have equal sizes.
`rho`	Numeric vector, the fraction of vertices per cluster, within a block. Must sum up to 1, and `rho * m` must be integer for all elements of rho. Alternatively a list of rho vectors, one for each block, if they are not the same for all blocks.
`C`	A square, symmetric numeric matrix, the Bernoulli rates for the clusters within a block. Its size must mach the size of the `rho` vector. Alternatively, a list of square matrices, if the Bernoulli rates differ in different blocks.
`p`	Numeric scalar, the Bernoulli rate of connections between vertices in different blocks.

Details

The function generates a random graph according to the hierarchical stochastic block model.

Value

An igraph graph.

Related documentation in the C library

`hsbm_game()`, `hsbm_list_game()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Random graph models (games): bipartite_gnm(), erdos.renyi.game(), sample_(), sample_bipartite(), sample_chung_lu(), sample_correlated_gnp(), sample_correlated_gnp_pair(), sample_degseq(), sample_dot_product(), sample_fitness(), sample_fitness_pl(), sample_forestfire(), sample_gnm(), sample_gnp(), sample_grg(), sample_growing(), sample_islands(), sample_k_regular(), sample_last_cit(), sample_pa(), sample_pa_age(), sample_pref(), sample_sbm(), sample_smallworld(), sample_traits_callaway(), sample_tree()

Examples

```
## Ten blocks with three clusters each
C <- matrix(c(
  1, 3 / 4, 0,
  3 / 4, 0, 3 / 4,
  0, 3 / 4, 3 / 4
), nrow = 3)
g <- sample_hierarchical_sbm(100, 10, rho = c(3, 3, 4) / 10, C = C, p = 1 / 20)
g


library("Matrix")
image(g[])
```

---

sample_hrg *Sample from a hierarchical random graph model*

Description

sample_hrg() samples a graph from a given hierarchical random graph model.

Usage

```
sample_hrg(hrg)
```

Arguments

hrg	A hierarchical random graph model.

Value

An igraph graph.

Related documentation in the C library

hrg_game()

See Also

Other hierarchical random graph functions: consensus_tree(), fit_hrg(), hrg(), hrg-methods, hrg_tree(), predict_edges(), print.igraphHRG(), print.igraphHRGConsensus()

---

sample_islands *A graph with subgraphs that are each a random graph.*

Description

Create a number of Erdős-Rényi random graphs with identical parameters, and connect them with the specified number of edges.

Usage

```
sample_islands(islands.n, islands.size, islands.pin, n.inter)
```

Arguments

islands.n	The number of islands in the graph.
islands.size	The size of islands in the graph.
islands.pin	The probability to create each possible edge into each island.
n.inter	The number of edges to create between two islands.

Value

An igraph graph.

Examples

```
g <- sample_islands(3, 10, 5/10, 1)
oc <- cluster_optimal(g)
oc
```

Related documentation in the C library

simple_interconnected_islands_game()

Author(s)

Samuel Thiriot

See Also

sample_gnp()

Random graph models (games): bipartite_gnm(), erdos.renyi.game(), sample_(), sample_bipartite(), sample_chung_lu(), sample_correlated_gnp(), sample_correlated_gnp_pair(), sample_degseq(), sample_dot_product(), sample_fitness(), sample_fitness_pl(), sample_forestfire(), sample_gnm(), sample_gnp(), sample_grg(), sample_growing(), sample_hierarchical_sbm(), sample_k_regular(), sample_last_cit(), sample_pa(), sample_pa_age(), sample_pref(), sample_sbm(), sample_smallworld(), sample_traits_callaway(), sample_tree()

---

sample_k_regular *Create a random regular graph*

Description

Generate a random graph where each vertex has the same degree.

Usage

```
sample_k_regular(no.of.nodes, k, directed = FALSE, multiple = FALSE)
```

Arguments

no.of.nodes	Integer scalar, the number of vertices in the generated graph.
k	Integer scalar, the degree of each vertex in the graph, or the out-degree and in-degree in a directed graph.
directed	Logical scalar, whether to create a directed graph.
multiple	Logical scalar, whether multiple edges are allowed.

Details

This game generates a directed or undirected random graph where the degrees of vertices are equal to a predefined constant k. For undirected graphs, at least one of k and the number of vertices must be even.

The game simply uses sample_degseq() with appropriately constructed degree sequences.

Value

An igraph graph.

Related documentation in the C library

k_regular_game()

Author(s)

Tamas Nepusz <ntamas@gmail.com>

See Also

sample_degseq() for a generator with prescribed degree sequence.

Random graph models (games): bipartite_gnm(), erdos.renyi.game(), sample_(), sample_bipartite(), sample_chung_lu(), sample_correlated_gnp(), sample_correlated_gnp_pair(), sample_degseq(), sample_dot_product(), sample_fitness(), sample_fitness_pl(), sample_forestfire(), sample_gnm(), sample_gnp(), sample_grg(), sample_growing(), sample_hierarchical_sbm(), sample_islands(), sample_last_cit(), sample_pa(), sample_pa_age(), sample_pref(), sample_sbm(), sample_smallworld(), sample_traits_callaway(), sample_tree()

Examples

```
## A simple ring
ring <- sample_k_regular(10, 2)
plot(ring)

## k-regular graphs on 10 vertices, with k=1:9
k10 <- lapply(1:9, sample_k_regular, no.of.nodes = 10)

layout(matrix(1:9, nrow = 3, byrow = TRUE))
sapply(k10, plot, vertex.label = NA)
```

---

sample_last_cit *Random citation graphs*

Description

sample_last_cit() creates a graph, where vertices age, and gain new connections based on how long ago their last citation happened.

Usage

```
sample_last_cit(
  n,
  edges = 1,
  agebins = n/7100,
  pref = (1:(agebins + 1))^-3,
  directed = TRUE
)

last_cit(
  n,
  edges = 1,
  agebins = n/7100,
  pref = (1:(agebins + 1))^-3,
  directed = TRUE
)
```

```
sample_cit_types(
  n,
  edges = 1,
  types = rep(0, n),
  pref = rep(1, length(types)),
  directed = TRUE,
  attr = TRUE
)

cit_types(
  n,
  edges = 1,
  types = rep(0, n),
  pref = rep(1, length(types)),
  directed = TRUE,
  attr = TRUE
)

sample_cit_cit_types(
  n,
  edges = 1,
  types = rep(0, n),
  pref = matrix(1, nrow = length(types), ncol = length(types)),
  directed = TRUE,
  attr = TRUE
)

cit_cit_types(
  n,
  edges = 1,
  types = rep(0, n),
  pref = matrix(1, nrow = length(types), ncol = length(types)),
  directed = TRUE,
  attr = TRUE
)
```

Arguments

n	Number of vertices.
edges	Number of edges per step.
agebins	Number of aging bins.
pref	Vector (`sample_last_cit()` and `sample_cit_types()` or matrix (`sample_cit_cit_types()`) giving the (unnormalized) citation probabilities for the different vertex types.
directed	Logical scalar, whether to generate directed networks.
types	Vector of length 'n', the types of the vertices. Types are numbered from zero.
attr	Logical scalar, whether to add the vertex types to the generated graph as a vertex attribute called 'type'.

Details

`sample_cit_cit_types()` is a stochastic block model where the graph is growing.

`sample_cit_types()` is similarly a growing stochastic block model, but the probability of an edge depends on the (potentially) cited vertex only.

Value

A new graph.

Related documentation in the C library

`lastcit_game()`, `cited_type_game()`, `vcount()`, `citing_cited_type_game()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Random graph models (games): `bipartite_gnm()`, `erdos.renyi.game()`, `sample_()`, `sample_bipartite()`, `sample_chung_lu()`, `sample_correlated_gnp()`, `sample_correlated_gnp_pair()`, `sample_degseq()`, `sample_dot_product()`, `sample_fitness()`, `sample_fitness_pl()`, `sample_forestfire()`, `sample_gnm()`, `sample_gnp()`, `sample_grg()`, `sample_growing()`, `sample_hierarchical_sbm()`, `sample_islands()`, `sample_k_regular()`, `sample_pa()`, `sample_pa_age()`, `sample_pref()`, `sample_sbm()`, `sample_smallworld()`, `sample_traits_callaway()`, `sample_tree()`

---

sample_motifs *Graph motifs*

Description

Graph motifs are small connected induced subgraphs with a well-defined structure. These functions search a graph for various motifs.

Usage

```
sample_motifs(
  graph,
  size = 3,
  cut.prob = rep(0, size),
  sample.size = NULL,
  sample = NULL
)
```

Arguments

graph	Graph object, the input graph.
size	The size of the motif, currently size 3 and 4 are supported in directed graphs and sizes 3-6 in undirected graphs.
cut.prob	Numeric vector giving the probabilities that the search graph is cut at a certain level. Its length should be the same as the size of the motif (the size argument). If NULL, the default, no cuts are made.
sample.size	The number of vertices to use as a starting point for finding motifs. Only used if the sample argument is NULL. The default is ceiling(vcount(graph) / 10) .
sample	If not NULL then it specifies the vertices to use as a starting point for finding motifs.

Details

sample_motifs() estimates the total number of motifs of a given size in a graph based on a sample.

Value

A numeric scalar, an estimate for the total number of motifs in the graph.

Related documentation in the C library

motifs_randesu_estimate(), vcount()

See Also

isomorphism_class()

Other graph motifs: count_motifs(), dyad_census(), motifs()

Examples

```
g <- sample_pa(100)
motifs(g, 3)
count_motifs(g, 3)
sample_motifs(g, 3)
```

---

sample_pa *Generate random graphs using preferential attachment*

Description

Preferential attachment is a family of simple stochastic algorithms for building a graph. Variants include the Barabási-Abert model and the Price model.

Usage

```
sample_pa(
  n,
  power = 1,
  m = NULL,
  out.dist = NULL,
  out.seq = NULL,
  out.pref = FALSE,
  zero.appeal = 1,
  directed = TRUE,
  algorithm = c("psumtree", "psumtree-multiple", "bag"),
  start.graph = NULL
)

pa(
  n,
  power = 1,
  m = NULL,
  out.dist = NULL,
  out.seq = NULL,
  out.pref = FALSE,
  zero.appeal = 1,
  directed = TRUE,
  algorithm = c("psumtree", "psumtree-multiple", "bag"),
  start.graph = NULL
)
```

Arguments

n	Number of vertices.
power	The power of the preferential attachment, the default is one, i.e. linear preferential attachment.
m	Numeric constant, the number of edges to add in each time step, defaults to 1. This argument is only used if both out.dist and out.seq are omitted or NULL.
out.dist	Numeric vector, the distribution of the number of edges to add in each time step. This argument is only used if the out.seq argument is omitted or NULL.
out.seq	Numeric vector giving the number of edges to add in each time step. Its first element is ignored as no edges are added in the first time step.
out.pref	Logical, if true the total degree is used for calculating the citation probability, otherwise the in-degree is used.
zero.appeal	The 'attractiveness' of the vertices with no adjacent edges. See details below.
directed	Whether to create a directed graph.
algorithm	The algorithm to use for the graph generation. psumtree uses a partial prefix-sum tree to generate the graph, this algorithm can handle any power and zero.appeal values and never generates multiple edges. psumtree-multiple also uses a partial prefix-sum tree, but the generation of multiple edges is allowed. Before the

0.6 version igraph used this algorithm if `power` was not one, or `zero.appeal` was not one. `bag` is the algorithm that was previously (before version 0.6) used if `power` was one and `zero.appeal` was one as well. It works by putting the ids of the vertices into a bag (multiset, really), exactly as many times as their (in-)degree, plus once more. Then the required number of cited vertices are drawn from the bag, with replacement. This method might generate multiple edges. It only works if `power` and `zero.appeal` are equal one.

`start.graph` `NULL` or an igraph graph. If a graph, then the supplied graph is used as a starting graph for the preferential attachment algorithm. The graph should have at least one vertex. If a graph is supplied here and the `out.seq` argument is not `NULL`, then it should contain the out degrees of the new vertices only, not the ones in the `start.graph`.

Details

This is a simple stochastic algorithm to generate a graph. It is a discrete time step model and in each time step a single vertex is added.

We start with a single vertex and no edges in the first time step. Then we add one vertex in each time step and the new vertex initiates some edges to old vertices. The probability that an old vertex is chosen is given by

$$P[i] \sim k_i^{\alpha} + a$$

where k_i is the in-degree of vertex i in the current time step (more precisely the number of adjacent edges of i which were not initiated by i itself) and α and a are parameters given by the `power` and `zero.appeal` arguments.

The number of edges initiated in a time step is given by the `m`, `out.dist` and `out.seq` arguments. If `out.seq` is given and not NULL then it gives the number of edges to add in a vector, the first element is ignored, the second is the number of edges to add in the second time step and so on. If `out.seq` is not given or null and `out.dist` is given and not NULL then it is used as a discrete distribution to generate the number of edges in each time step. Its first element is the probability that no edges will be added, the second is the probability that one edge is added, etc. (`out.dist` does not need to sum up to one, it normalized automatically.) `out.dist` should contain non-negative numbers and at east one element should be positive.

If both `out.seq` and `out.dist` are omitted or NULL then `m` will be used, it should be a positive integer constant and `m` edges will be added in each time step.

`sample_pa()` generates a directed graph by default, set `directed` to `FALSE` to generate an undirected graph. Note that even if an undirected graph is generated k_i denotes the number of adjacent edges not initiated by the vertex itself and not the total (in- + out-) degree of the vertex, unless the `out.pref` argument is set to `TRUE`.

Value

A graph object.

Related documentation in the C library

`barabasi_game()`, `vcount()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Barabási, A.-L. and Albert R. 1999. Emergence of scaling in random networks *Science*, 286 509–512.

de Solla Price, D. J. 1965. Networks of Scientific Papers *Science*, 149 510–515.

See Also

Random graph models (games): bipartite_gnm(), erdos.renyi.game(), sample_(), sample_bipartite(), sample_chung_lu(), sample_correlated_gnp(), sample_correlated_gnp_pair(), sample_degseq(), sample_dot_product(), sample_fitness(), sample_fitness_pl(), sample_forestfire(), sample_gnm(), sample_gnp(), sample_grg(), sample_growing(), sample_hierarchical_sbm(), sample_islands(), sample_k_regular(), sample_last_cit(), sample_pa_age(), sample_pref(), sample_sbm(), sample_smallworld(), sample_traits_callaway(), sample_tree()

Examples

```
g <- sample_pa(10000)
degree_distribution(g)
```

---

sample_pa_age *Generate an evolving random graph with preferential attachment and aging*

Description

This function creates a random graph by simulating its evolution. Each time a new vertex is added it creates a number of links to old vertices and the probability that an old vertex is cited depends on its in-degree (preferential attachment) and age.

Usage

```
sample_pa_age(
  n,
  pa.exp,
  aging.exp,
  m = NULL,
  aging.bin = 300,
  out.dist = NULL,
  out.seq = NULL,
  out.pref = FALSE,
  directed = TRUE,
  zero.deg.appeal = 1,
```

```
  zero.age.appeal = 0,
  deg.coef = 1,
  age.coef = 1,
  time.window = NULL
)

pa_age(
  n,
  pa.exp,
  aging.exp,
  m = NULL,
  aging.bin = 300,
  out.dist = NULL,
  out.seq = NULL,
  out.pref = FALSE,
  directed = TRUE,
  zero.deg.appeal = 1,
  zero.age.appeal = 0,
  deg.coef = 1,
  age.coef = 1,
  time.window = NULL
)
```

Arguments

n	The number of vertices in the graph.
pa.exp	The preferential attachment exponent, see the details below.
aging.exp	The exponent of the aging, usually a non-positive number, see details below.
m	The number of edges each new vertex creates (except the very first vertex). This argument is used only if both the out.dist and out.seq arguments are NULL.
aging.bin	The number of bins to use for measuring the age of vertices, see details below.
out.dist	The discrete distribution to generate the number of edges to add in each time step if out.seq is NULL. See details below.
out.seq	The number of edges to add in each time step, a vector containing as many elements as the number of vertices. See details below.
out.pref	Logical constant, whether to include edges not initiated by the vertex as a basis of preferential attachment. See details below.
directed	Logical constant, whether to generate a directed graph. See details below.
zero.deg.appeal	The degree-dependent part of the 'attractiveness' of the vertices with no adjacent edges. See also details below.
zero.age.appeal	The age-dependent part of the 'attrativeness' of the vertices with age zero. It is usually zero, see details below.
deg.coef	The coefficient of the degree-dependent 'attractiveness'. See details below.

`age.coef` The coefficient of the age-dependent part of the 'attractiveness'. See details below.

`time.window` Integer constant, if NULL only adjacent added in the last `time.windows` time steps are counted as a basis of the preferential attachment. See also details below.

Details

This is a discrete time step model of a growing graph. We start with a network containing a single vertex (and no edges) in the first time step. Then in each time step (starting with the second) a new vertex is added and it initiates a number of edges to the old vertices in the network. The probability that an old vertex is connected to is proportional to

$$P[i] \sim (c \cdot k_i^\alpha + a)(d \cdot l_i^\beta + b)$$

.

Here k_i is the in-degree of vertex i in the current time step and l_i is the age of vertex i. The age is simply defined as the number of time steps passed since the vertex is added, with the extension that vertex age is divided to be in `aging.bin` bins.

c, α, a, d, β and b are parameters and they can be set via the following arguments: `pa.exp` (α, mandatory argument), `aging.exp` (β, mandatory argument), `zero.deg.appeal` (a, optional, the default value is 1), `zero.age.appeal` (b, optional, the default is 0), `deg.coef` (c, optional, the default is 1), and `age.coef` (d, optional, the default is 1).

The number of edges initiated in each time step is governed by the `m`, `out.seq` and `out.pref` parameters. If `out.seq` is given then it is interpreted as a vector giving the number of edges to be added in each time step. It should be of length `n` (the number of vertices), and its first element will be ignored. If `out.seq` is not given (or NULL) and `out.dist` is given then it will be used as a discrete probability distribution to generate the number of edges. Its first element gives the probability that zero edges are added at a time step, the second element is the probability that one edge is added, etc. (`out.seq` should contain non-negative numbers, but if they don't sum up to 1, they will be normalized to sum up to 1. This behavior is similar to the `prob` argument of the `sample` command.)

By default a directed graph is generated, but it `directed` is set to `FALSE` then an undirected is created. Even if an undirected graph is generated k_i denotes only the adjacent edges not initiated by the vertex itself except if `out.pref` is set to `TRUE`.

If the `time.window` argument is given (and not NULL) then k_i means only the adjacent edges added in the previous `time.window` time steps.

This function might generate graphs with multiple edges.

Value

A new graph.

Related documentation in the C library

`barabasi_aging_game()`, `recent_degree_aging_game()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Random graph models (games): bipartite_gnm(), erdos.renyi.game(), sample_(), sample_bipartite(), sample_chung_lu(), sample_correlated_gnp(), sample_correlated_gnp_pair(), sample_degseq(), sample_dot_product(), sample_fitness(), sample_fitness_pl(), sample_forestfire(), sample_gnm(), sample_gnp(), sample_grg(), sample_growing(), sample_hierarchical_sbm(), sample_islands(), sample_k_regular(), sample_last_cit(), sample_pa(), sample_pref(), sample_sbm(), sample_smallworld(), sample_traits_callaway(), sample_tree()

Examples

```
# The maximum degree for graph with different aging exponents
g1 <- sample_pa_age(10000, pa.exp = 1, aging.exp = 0, aging.bin = 1000)
g2 <- sample_pa_age(10000, pa.exp = 1, aging.exp = -1, aging.bin = 1000)
g3 <- sample_pa_age(10000, pa.exp = 1, aging.exp = -3, aging.bin = 1000)
max(degree(g1))
max(degree(g2))
max(degree(g3))
```

---

sample_pref *Trait-based random generation*

Description

Generation of random graphs based on different vertex types.

Usage

```
sample_pref(
  nodes,
  types,
  type.dist = rep(1, types),
  fixed.sizes = FALSE,
  pref.matrix = matrix(1, types, types),
  directed = FALSE,
  loops = FALSE
)

pref(
  nodes,
  types,
  type.dist = rep(1, types),
  fixed.sizes = FALSE,
  pref.matrix = matrix(1, types, types),
```

```
  directed = FALSE,
  loops = FALSE
)

sample_asym_pref(
  nodes,
  types,
  type.dist.matrix = matrix(1, types, types),
  pref.matrix = matrix(1, types, types),
  loops = FALSE
)

asym_pref(
  nodes,
  types,
  type.dist.matrix = matrix(1, types, types),
  pref.matrix = matrix(1, types, types),
  loops = FALSE
)
```

Arguments

nodes	The number of vertices in the graphs.
types	The number of different vertex types.
type.dist	The distribution of the vertex types, a numeric vector of length 'types' containing non-negative numbers. The vector will be normed to obtain probabilities.
fixed.sizes	Fix the number of vertices with a given vertex type label. The type.dist argument gives the group sizes (i.e. number of vertices with the different labels) in this case.
pref.matrix	A square matrix giving the preferences of the vertex types. The matrix has 'types' rows and columns. When generating an undirected graph, it must be symmetric.
directed	Logical scalar, whether to create a directed graph.
loops	Logical scalar, whether self-loops are allowed in the graph.
type.dist.matrix	The joint distribution of the in- and out-vertex types.

Details

Both models generate random graphs with given vertex types. For sample_pref() the probability that two vertices will be connected depends on their type and is given by the 'pref.matrix' argument. This matrix should be symmetric to make sense but this is not checked. The distribution of the different vertex types is given by the 'type.dist' vector.

For sample_asym_pref() each vertex has an in-type and an out-type and a directed graph is created. The probability that a directed edge is realized from a vertex with a given out-type to a vertex with a given in-type is given in the 'pref.matrix' argument, which can be asymmetric. The joint distribution for the in- and out-types is given in the 'type.dist.matrix' argument.

The types of the generated vertices can be retrieved from the `type` vertex attribute for `sample_pref()` and from the `intype` and `outtype` vertex attribute for `sample_asym_pref()`.

Value

An igraph graph.

Related documentation in the C library

`preference_game()`, `vcount()`, `asymmetric_preference_game()`

Author(s)

Tamas Nepusz <ntamas@gmail.com> and Gabor Csardi <csardi.gabor@gmail.com> for the R interface

See Also

Random graph models (games): `bipartite_gnm()`, `erdos.renyi.game()`, `sample_()`, `sample_bipartite()`, `sample_chung_lu()`, `sample_correlated_gnp()`, `sample_correlated_gnp_pair()`, `sample_degseq()`, `sample_dot_product()`, `sample_fitness()`, `sample_fitness_pl()`, `sample_forestfire()`, `sample_gnm()`, `sample_gnp()`, `sample_grg()`, `sample_growing()`, `sample_hierarchical_sbm()`, `sample_islands()`, `sample_k_regular()`, `sample_last_cit()`, `sample_pa()`, `sample_pa_age()`, `sample_sbm()`, `sample_smallworld()`, `sample_traits_callaway()`, `sample_tree()`

Examples

```
pf <- matrix(c(1, 0, 0, 1), nrow = 2)
g <- sample_pref(20, 2, pref.matrix = pf)

# example code

tkplot(g, layout = layout_with_fr)


pf <- matrix(c(0, 1, 0, 0), nrow = 2)
g <- sample_asym_pref(20, 2, pref.matrix = pf)

tkplot(g, layout = layout_in_circle)
```

---

sample_sbm *Sample stochastic block model*

Description

Sampling from the stochastic block model of networks

Usage

```
sample_sbm(n, pref.matrix, block.sizes, directed = FALSE, loops = FALSE)

sbm(n, pref.matrix, block.sizes, directed = FALSE, loops = FALSE)
```

Arguments

n	Number of vertices in the graph.
pref.matrix	The matrix giving the Bernoulli rates. This is a $K \times K$ matrix, where K is the number of groups. The probability of creating an edge between vertices from groups i and j is given by element (i, j). For undirected graphs, this matrix must be symmetric.
block.sizes	Numeric vector giving the number of vertices in each group. The sum of the vector must match the number of vertices.
directed	Logical scalar, whether to create a directed graph.
loops	Logical scalar, whether self-loops are allowed in the graph.

Details

This function samples graphs from a stochastic block model by (doing the equivalent of) Bernoulli trials for each potential edge with the probabilities given by the Bernoulli rate matrix, `pref.matrix`. The order of the vertices in the generated graph corresponds to the `block.sizes` argument.

Value

An igraph graph.

Related documentation in the C library

sbm_game()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Faust, K., & Wasserman, S. (1992a). Blockmodels: Interpretation and evaluation. *Social Networks*, 14, 5–61.

See Also

Random graph models (games): bipartite_gnm(), erdos.renyi.game(), sample_(), sample_bipartite(), sample_chung_lu(), sample_correlated_gnp(), sample_correlated_gnp_pair(), sample_degseq(), sample_dot_product(), sample_fitness(), sample_fitness_pl(), sample_forestfire(), sample_gnm(), sample_gnp(), sample_grg(), sample_growing(), sample_hierarchical_sbm(), sample_islands(), sample_k_regular(), sample_last_cit(), sample_pa(), sample_pa_age(), sample_pref(), sample_smallworld(), sample_traits_callaway(), sample_tree()

Examples

```
## Two groups with not only few connection between groups
pm <- cbind(c(.1, .001), c(.001, .05))
g <- sample_sbm(1000, pref.matrix = pm, block.sizes = c(300, 700))
g
```

`sample_seq` *Sampling a random integer sequence*

Description

This function provides a very efficient way to pull an integer random sample sequence from an integer interval.

Usage

```
sample_seq(low, high, length)
```

Arguments

`low`	The lower limit of the interval (inclusive).
`high`	The higher limit of the interval (inclusive).
`length`	The length of the sample.

Details

The algorithm runs in `O(length)` expected time, even if `high-low` is big. It is much faster (but of course less general) than the builtin `sample` function of R.

Value

An increasing numeric vector containing integers, the sample.

Related documentation in the C library

`random_sample()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Jeffrey Scott Vitter: An Efficient Algorithm for Sequential Random Sampling, *ACM Transactions on Mathematical Software*, 13/1, 58–67.

See Also

Other other: convex_hull(), running_mean()

Examples

```
rs <- sample_seq(1, 100000000, 10)
rs
```

---

sample_smallworld *The Watts-Strogatz small-world model*

Description

This function generates networks with the small-world property based on a variant of the Watts-Strogatz model. The network is obtained by first creating a periodic undirected lattice, then rewiring both endpoints of each edge with probability p, while avoiding the creation of multi-edges.

Usage

```
sample_smallworld(dim, size, nei, p, loops = FALSE, multiple = FALSE)

smallworld(dim, size, nei, p, loops = FALSE, multiple = FALSE)
```

Arguments

dim	Integer constant, the dimension of the starting lattice.
size	Integer constant, the size of the lattice along each dimension.
nei	Integer constant, the neighborhood within which the vertices of the lattice will be connected.
p	Real constant between zero and one, the rewiring probability.
loops	Logical scalar, whether loops edges are allowed in the generated graph.
multiple	Logical scalar, whether multiple edges are allowed int the generated graph.

Details

Note that this function might create graphs with loops and/or multiple edges. You can use simplify() to get rid of these.

This process differs from the original model of Watts and Strogatz (see reference) in that it rewires **both** endpoints of edges. Thus in the limit of p=1, we obtain a G(n,m) random graph with the same number of vertices and edges as the original lattice. In comparison, the original Watts-Strogatz model only rewires a single endpoint of each edge, thus the network does not become fully random even for p=1. For appropriate choices of p, both models exhibit the property of simultaneously having short path lengths and high clustering.

Value

A graph object.

Related documentation in the C library

watts_strogatz_game()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Duncan J Watts and Steven H Strogatz: Collective dynamics of 'small world' networks, Nature 393, 440-442, 1998.

See Also

make_lattice(), rewire()

Random graph models (games): bipartite_gnm(), erdos.renyi.game(), sample_(), sample_bipartite(), sample_chung_lu(), sample_correlated_gnp(), sample_correlated_gnp_pair(), sample_degseq(), sample_dot_product(), sample_fitness(), sample_fitness_pl(), sample_forestfire(), sample_gnm(), sample_gnp(), sample_grg(), sample_growing(), sample_hierarchical_sbm(), sample_islands(), sample_k_regular(), sample_last_cit(), sample_pa(), sample_pa_age(), sample_pref(), sample_sbm(), sample_traits_callaway(), sample_tree()

Examples

```
g <- sample_smallworld(1, 100, 5, 0.05)
mean_distance(g)
transitivity(g, type = "average")
```

---

sample_spanning_tree *Samples from the spanning trees of a graph randomly and uniformly*

Description

sample_spanning_tree() picks a spanning tree of an undirected graph randomly and uniformly, using loop-erased random walks.

Usage

```
sample_spanning_tree(graph, vid = 0)
```

Arguments

graph	The input graph to sample from. Edge directions are ignored if the graph is directed.
vid	When the graph is disconnected, this argument specifies how to handle the situation. When the argument is zero (the default), the sampling will be performed component-wise, and the result will be a spanning forest. When the argument contains a vertex ID, only the component containing the given vertex will be processed, and the result will be a spanning tree of the component of the graph.

Value

An edge sequence containing the edges of the spanning tree. Use subgraph_from_edges() to extract the corresponding subgraph.

Related documentation in the C library

random_spanning_tree(), ecount(), edges(), vcount(), get_eids()

See Also

subgraph_from_edges() to extract the tree itself

Other trees: is_forest(), is_tree(), make_from_prufer(), to_prufer()

Examples

```
g <- make_full_graph(10) %du% make_full_graph(5)
edges <- sample_spanning_tree(g)
forest <- subgraph_from_edges(g, edges)
```

---

sample_sphere_surface *Sample vectors uniformly from the surface of a sphere*

Description

Sample finite-dimensional vectors to use as latent position vectors in random dot product graphs

Usage

```
sample_sphere_surface(dim, n = 1, radius = 1, positive = TRUE)
```

Arguments

dim	Integer scalar, the dimension of the random vectors.
n	Integer scalar, the sample size.
radius	Numeric scalar, the radius of the sphere to sample.
positive	Logical scalar, whether to sample from the positive orthant of the sphere.

Details

`sample_sphere_surface()` generates uniform samples from S^{dim-1} (the `(dim-1)`-sphere) with radius `radius`, i.e. the Euclidean norm of the samples equal `radius`.

Value

A `dim` (length of the `alpha` vector for `sample_dirichlet()`) times `n` matrix, whose columns are the sample vectors.

Related documentation in the C library

`sample_sphere_surface()`

See Also

Other latent position vector samplers: `sample_dirichlet()`, `sample_sphere_volume()`

Examples

```
lpvs.sph <- sample_sphere_surface(dim = 10, n = 20, radius = 1)
RDP.graph.3 <- sample_dot_product(lpvs.sph)
vec.norm <- apply(lpvs.sph, 2, function(x) {
  sum(x^2)
})
vec.norm
```

---

sample_sphere_volume *Sample vectors uniformly from the volume of a sphere*

Description

Sample finite-dimensional vectors to use as latent position vectors in random dot product graphs

Usage

```
sample_sphere_volume(dim, n = 1, radius = 1, positive = TRUE)
```

Arguments

`dim`	Integer scalar, the dimension of the random vectors.
`n`	Integer scalar, the sample size.
`radius`	Numeric scalar, the radius of the sphere to sample.
`positive`	Logical scalar, whether to sample from the positive orthant of the sphere.

Details

`sample_sphere_volume()` generates uniform samples from S^{dim-1} (the `(dim-1)`-sphere) i.e. the Euclidean norm of the samples is smaller or equal to `radius`.

Value

A `dim` (length of the `alpha` vector for `sample_dirichlet()`) times `n` matrix, whose columns are the sample vectors.

Related documentation in the C library

`sample_sphere_volume()`

See Also

Other latent position vector samplers: `sample_dirichlet()`, `sample_sphere_surface()`

Examples

```
lpvs.sph.vol <- sample_sphere_volume(dim = 10, n = 20, radius = 1)
RDP.graph.4 <- sample_dot_product(lpvs.sph.vol)
vec.norm <- apply(lpvs.sph.vol, 2, function(x) {
  sum(x^2)
})
vec.norm
```

`sample_traits_callaway` *Graph generation based on different vertex types*

Description

These functions implement evolving network models based on different vertex types.

Usage

```
sample_traits_callaway(
  nodes,
  types,
  edge.per.step = 1,
  type.dist = rep(1, types),
  pref.matrix = matrix(1, types, types),
  directed = FALSE
)

traits_callaway(
  nodes,
  types,
  edge.per.step = 1,
  type.dist = rep(1, types),
  pref.matrix = matrix(1, types, types),
  directed = FALSE
```

```
)

sample_traits(
  nodes,
  types,
  k = 1,
  type.dist = rep(1, types),
  pref.matrix = matrix(1, types, types),
  directed = FALSE
)

traits(
  nodes,
  types,
  k = 1,
  type.dist = rep(1, types),
  pref.matrix = matrix(1, types, types),
  directed = FALSE
)
```

Arguments

nodes	The number of vertices in the graph.
types	The number of different vertex types.
edge.per.step	The number of edges to add to the graph per time step.
type.dist	The distribution of the vertex types. This is assumed to be stationary in time.
pref.matrix	A matrix giving the preferences of the given vertex types. These should be probabilities, i.e. numbers between zero and one.
directed	Logical constant, whether to generate directed graphs.
k	The number of trials per time step, see details below.

Details

For sample_traits_callaway() the simulation goes like this: in each discrete time step a new vertex is added to the graph. The type of this vertex is generated based on type.dist. Then two vertices are selected uniformly randomly from the graph. The probability that they will be connected depends on the types of these vertices and is taken from pref.matrix. Then another two vertices are selected and this is repeated edges.per.step times in each time step.

For sample_traits() the simulation goes like this: a single vertex is added at each time step. This new vertex tries to connect to k vertices in the graph. The probability that such a connection is realized depends on the types of the vertices involved and is taken from pref.matrix.

Value

A new graph object.

Related documentation in the C library

callaway_traits_game(), establishment_game()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Random graph models (games): bipartite_gnm(), erdos.renyi.game(), sample_(), sample_bipartite(), sample_chung_lu(), sample_correlated_gnp(), sample_correlated_gnp_pair(), sample_degseq(), sample_dot_product(), sample_fitness(), sample_fitness_pl(), sample_forestfire(), sample_gnm(), sample_gnp(), sample_grg(), sample_growing(), sample_hierarchical_sbm(), sample_islands(), sample_k_regular(), sample_last_cit(), sample_pa(), sample_pa_age(), sample_pref(), sample_sbm(), sample_smallworld(), sample_tree()

Examples

```
# two types of vertices, they like only themselves
g1 <- sample_traits_callaway(1000, 2, pref.matrix = matrix(c(1, 0, 0, 1), ncol = 2))
g2 <- sample_traits(1000, 2, k = 2, pref.matrix = matrix(c(1, 0, 0, 1), ncol = 2))
```

---

sample_tree *Sample trees randomly and uniformly*

Description

sample_tree() generates a random with a given number of nodes uniform at random from the set of labelled trees.

Usage

```
sample_tree(n, directed = FALSE, method = c("lerw", "prufer"))
```

Arguments

n	The number of nodes in the tree
directed	Whether to create a directed tree. The edges of the tree are oriented away from the root.
method	The algorithm to use to generate the tree. 'prufer' samples Prüfer sequences uniformly and then converts the sampled sequence to a tree. 'lerw' performs a loop-erased random walk on the complete graph to uniformly sampleits spanning trees. (This is also known as Wilson's algorithm). The default is 'lerw'. Note that the method based on Prüfer sequences does not support directed trees at the moment.

Details

In other words, the function generates each possible labelled tree with the given number of nodes with the same probability.

Value

A graph object.

Related documentation in the C library

tree_game()

See Also

Random graph models (games): bipartite_gnm(), erdos.renyi.game(), sample_(), sample_bipartite(), sample_chung_lu(), sample_correlated_gnp(), sample_correlated_gnp_pair(), sample_degseq(), sample_dot_product(), sample_fitness(), sample_fitness_pl(), sample_forestfire(), sample_gnm(), sample_gnp(), sample_grg(), sample_growing(), sample_hierarchical_sbm(), sample_islands(), sample_k_regular(), sample_last_cit(), sample_pa(), sample_pa_age(), sample_pref(), sample_sbm(), sample_smallworld(), sample_traits_callaway()

Examples

```
g <- sample_tree(100, method = "lerw")
```

---

scan_stat *Scan statistics on a time series of graphs*

Description

Calculate scan statistics on a time series of graphs. This is done by calculating the local scan statistics for each graph and each vertex, and then normalizing across the vertices and across the time steps.

Usage

```
scan_stat(graphs, tau = 1, ell = 0, locality = c("us", "them"), ...)
```

Arguments

graphs	A list of igraph graph objects. They must be all directed or all undirected and they must have the same number of vertices.
tau	The number of previous time steps to consider for the time-dependent normalization for individual vertices. In other words, the current locality statistics of each vertex will be compared to this many previous time steps of the same vertex to decide whether it is significantly larger.

`ell`	The number of previous time steps to consider for the aggregated scan statistics. This is essentially a smoothing parameter.
`locality`	Whether to calculate the 'us' or 'them' statistics.
`...`	Extra arguments are passed to `local_scan()`.

Value

A list with entries:

stat The scan statistics in each time step. It is `NA` for the initial `tau + ell` time steps.

arg_max_v The (numeric) vertex ids for the vertex with the largest locality statistics, at each time step. It is `NA` for the initial `tau + ell` time steps.

Related documentation in the C library

`vcount()`, `local_scan_0()`, `local_scan_0_them()`, `local_scan_1_ecount()`, `local_scan_1_ecount_them()`, `local_scan_k_ecount()`, `local_scan_k_ecount_them()`, `local_scan_neighborhood_ecount()`, `induced_subgraph()`, `edges()`, `get_eids()`, `ecount()`

See Also

Other scan statistics: `local_scan()`

Examples

```
## Generate a bunch of SBMs, with the last one being different
num_t <- 20
block_sizes <- c(10, 5, 5)
p_ij <- list(p = 0.1, h = 0.9, q = 0.9)

P0 <- matrix(p_ij$p, 3, 3)
P0[2, 2] <- p_ij$h
PA <- P0
PA[3, 3] <- p_ij$q
num_v <- sum(block_sizes)

tsg <- replicate(num_t - 1, P0, simplify = FALSE) %>%
  append(list(PA)) %>%
  lapply(sample_sbm, n = num_v, block.sizes = block_sizes, directed = TRUE)

scan_stat(graphs = tsg, k = 1, tau = 4, ell = 2)
scan_stat(graphs = tsg, locality = "them", k = 1, tau = 4, ell = 2)
```

`sequential_pal` *Sequential palette*

Description

This is the 'OrRd' palette from https://colorbrewer2.org/. It has at most nine colors.

Usage

```
sequential_pal(n)
```

Arguments

`n`	The number of colors in the palette. The maximum is nine currently.

Details

Use this palette, if vertex colors mark some ordinal quantity, e.g. some centrality measure, or some ordinal vertex covariate, like the age of people, or their seniority level.

Value

A character vector of RGB color codes.

See Also

Other palettes: `categorical_pal()`, `diverging_pal()`, `r_pal()`

Examples

```
library(igraphdata)
data(karate)
karate <- karate %>%
  add_layout_(with_kk()) %>%
  set_vertex_attr("size", value = 10)

V(karate)$color <- scales::dscale(degree(karate) %>% cut(5), sequential_pal)
plot(karate)
```

set_edge_attr *Set edge attributes*

Description

Set edge attributes

Usage

```
set_edge_attr(graph, name, index = E(graph), value)
```

Arguments

graph	The graph
name	The name of the attribute to set.
index	An optional edge sequence to set the attributes of a subset of edges.
value	The new value of the attribute for all (or index) edges. If NULL, the input is returned unchanged.

Value

The graph, with the edge attribute added or set.

Related documentation in the C library

edges(), get_eids(), vcount(), ecount()

See Also

Vertex, edge and graph attributes: delete_edge_attr(), delete_graph_attr(), delete_vertex_attr(), edge_attr(), edge_attr<-(), edge_attr_names(), graph_attr(), graph_attr<-(), graph_attr_names(), igraph-attribute-combination, igraph-dollar, igraph-vs-attributes, set_graph_attr(), set_vertex_attr(), set_vertex_attrs(), vertex_attr(), vertex_attr<-(), vertex_attr_names()

Examples

```
g <- make_ring(10) %>%
  set_edge_attr("label", value = LETTERS[1:10])
g
plot(g)
```

set_graph_attr *Set a graph attribute*

Description

An existing attribute with the same name is overwritten.

Usage

```
set_graph_attr(graph, name, value)
```

Arguments

graph	The graph.
name	The name of the attribute to set.
value	New value of the attribute.

Value

The graph with the new graph attribute added or set.

See Also

Vertex, edge and graph attributes: delete_edge_attr(), delete_graph_attr(), delete_vertex_attr(), edge_attr(), edge_attr<-(), edge_attr_names(), graph_attr(), graph_attr<-(), graph_attr_names(), igraph-attribute-combination, igraph-dollar, igraph-vs-attributes, set_edge_attr(), set_vertex_attr(), set_vertex_attrs(), vertex_attr(), vertex_attr<-(), vertex_attr_names()

Examples

```
g <- make_ring(10) %>%
  set_graph_attr("layout", layout_with_fr)
g
plot(g)
```

set_vertex_attr *Set vertex attributes*

Description

Set vertex attributes

Usage

```
set_vertex_attr(graph, name, index = V(graph), value)
```

Arguments

graph	The graph.
name	The name of the attribute to set.
index	An optional vertex sequence to set the attributes of a subset of vertices.
value	The new value of the attribute for all (or index) vertices. If NULL, the input is returned unchanged.

Value

The graph, with the vertex attribute added or set.

Related documentation in the C library

vcount()

See Also

Vertex, edge and graph attributes: delete_edge_attr(), delete_graph_attr(), delete_vertex_attr(), edge_attr(), edge_attr<-(), edge_attr_names(), graph_attr(), graph_attr<-(), graph_attr_names(), igraph-attribute-combination, igraph-dollar, igraph-vs-attributes, set_edge_attr(), set_graph_attr(), set_vertex_attrs(), vertex_attr(), vertex_attr<-(), vertex_attr_names()

Examples

```
g <- make_ring(10) %>%
  set_vertex_attr("label", value = LETTERS[1:10])
g
plot(g)
```

---

set_vertex_attrs *Set multiple vertex attributes*

Description

Set multiple vertex attributes

Usage

```
set_vertex_attrs(graph, ..., index = V(graph))
```

Arguments

graph	The graph.
...	<dynamic-dots> Named arguments, where the names are the attributes
index	An optional vertex sequence to set the attributes of a subset of vertices.

Value

The graph, with the vertex attributes added or set.

Related documentation in the C library

vcount()

See Also

Vertex, edge and graph attributes: delete_edge_attr(), delete_graph_attr(), delete_vertex_attr(), edge_attr(), edge_attr<-(), edge_attr_names(), graph_attr(), graph_attr<-(), graph_attr_names(), igraph-attribute-combination, igraph-dollar, igraph-vs-attributes, set_edge_attr(), set_graph_attr(), set_vertex_attr(), vertex_attr(), vertex_attr<-(), vertex_attr_names()

Examples

```
g <- make_ring(10)
set_vertex_attrs(g, color = "blue", size = 10, name = LETTERS[1:10])
# use splicing if suplying a list
x <- list(color = "red", name = LETTERS[1:10])
set_vertex_attrs(g, !!!x)
# to set an attribute named "index" use `:=`
set_vertex_attrs(g, color = "blue", index := 10, name = LETTERS[1:10])
```

---

shapes *Various vertex shapes when plotting igraph graphs*

Description

Starting from version 0.5.1 igraph supports different vertex shapes when plotting graphs.

Usage

```
shapes(shape = NULL)

shape_noclip(coords, el, params, end = c("both", "from", "to"))

shape_noplot(coords, v = NULL, params)

add_shape(shape, clip = shape_noclip, plot = shape_noplot, parameters = list())
```

Arguments

shape	Character scalar, name of a vertex shape. If it is NULL for shapes(), then the names of all defined vertex shapes are returned.
coords, el, params, end, v	See parameters of the clipping/plotting functions below.

clip	An R function object, the clipping function.
plot	An R function object, the plotting function.
parameters	Named list, additional plot/vertex/edge parameters. The element named define the new parameters, and the elements themselves define their default values. Vertex parameters should have a prefix 'vertex.', edge parameters a prefix 'edge.'. Other general plotting parameters should have a prefix 'plot.'. See Details below.

Details

In igraph a vertex shape is defined by two functions: 1) provides information about the size of the shape for clipping the edges and 2) plots the shape if requested. These functions are called "shape functions" in the rest of this manual page. The first one is the clipping function and the second is the plotting function.

The clipping function has the following arguments:

coords A matrix with four columns, it contains the coordinates of the vertices for the edge list supplied in the el argument.

el A matrix with two columns, the edges of which some end points will be clipped. It should have the same number of rows as coords.

params This is a function object that can be called to query vertex/edge/plot graphical parameters. The first argument of the function is "vertex", "edge" or "plot" to decide the type of the parameter, the second is a character string giving the name of the parameter. E.g. params("vertex", "size").

end Character string, it gives which end points will be used. Possible values are "both", "from" and "to". If "from" the function is expected to clip the first column in the el edge list, "to" selects the second column, "both" selects both.

The clipping function should return a matrix with the same number of rows as the el arguments. If end is both then the matrix must have four columns, otherwise two. The matrix contains the modified coordinates, with the clipping applied.

The plotting function has the following arguments:

coords The coordinates of the vertices, a matrix with two columns.

v The ids of the vertices to plot. It should match the number of rows in the coords argument.

params The same as for the clipping function, see above.

The return value of the plotting function is not used.

shapes() can be used to list the names of all installed vertex shapes, by calling it without arguments, or setting the shape argument to NULL. If a shape name is given, then the clipping and plotting functions of that shape are returned in a named list.

add_shape() can be used to add new vertex shapes to igraph. For this one must give the clipping and plotting functions of the new shape. It is also possible to list the plot/vertex/edge parameters, in the parameters argument, that the clipping and/or plotting functions can make use of. An example would be a generic regular polygon shape, which can have a parameter for the number of sides.

shape_noclip() is a very simple clipping function that the user can use in their own shape definitions. It does no clipping, the edges will be drawn exactly until the listed vertex position coordinates.

shape_noplot() is a very simple (and probably not very useful) plotting function, that does not plot anything.

Value

shapes() returns a character vector if the shape argument is NULL. It returns a named list with entries named 'clip' and 'plot', both of them R functions.

add_shape() returns TRUE, invisibly.

shape_noclip() returns the appropriate columns of its coords argument.

Examples

```
# all vertex shapes, minus "raster", that might not be available
shapes <- setdiff(shapes(), "")
g <- make_ring(length(shapes))
set.seed(42)
plot(g,
  vertex.shape = shapes, vertex.label = shapes, vertex.label.dist = 1,
  vertex.size = 15, vertex.size2 = 15,
  vertex.pie = lapply(shapes, function(x) if (x == "pie") 2:6 else 0),
  vertex.pie.color = list(heat.colors(5))
)

# add new vertex shape, plot nothing with no clipping
add_shape("nil")
plot(g, vertex.shape = "nil")

#################################################################
# triangle vertex shape
mytriangle <- function(coords, v = NULL, params) {
  vertex.color <- params("vertex", "color")
  if (length(vertex.color) != 1 && !is.null(v)) {
    vertex.color <- vertex.color[v]
  }
  vertex.size <- params("vertex", "size")
  if (length(vertex.size) != 1 && !is.null(v)) {
    vertex.size <- vertex.size[v]
  }

  symbols(
    x = coords[, 1], y = coords[, 2], bg = vertex.color,
    stars = cbind(vertex.size, vertex.size, vertex.size),
    add = TRUE, inches = FALSE
  )
}
# clips as a circle
add_shape("triangle",
  clip = shapes("circle")$clip,
  plot = mytriangle
)
plot(g,
  vertex.shape = "triangle", vertex.color = rainbow(vcount(g)),
```

```
  vertex.size = seq(10, 20, length.out = vcount(g))
)

################################################################
# generic star vertex shape, with a parameter for number of rays
mystar <- function(coords, v = NULL, params) {
  vertex.color <- params("vertex", "color")
  if (length(vertex.color) != 1 && !is.null(v)) {
    vertex.color <- vertex.color[v]
  }
  vertex.size <- params("vertex", "size")
  if (length(vertex.size) != 1 && !is.null(v)) {
    vertex.size <- vertex.size[v]
  }
  norays <- params("vertex", "norays")
  if (length(norays) != 1 && !is.null(v)) {
    norays <- norays[v]
  }

  mapply(coords[, 1], coords[, 2], vertex.color, vertex.size, norays,
    FUN = function(x, y, bg, size, nor) {
      symbols(
        x = x, y = y, bg = bg,
        stars = matrix(c(size, size / 2), nrow = 1, ncol = nor * 2),
        add = TRUE, inches = FALSE
      )
    }
  )
}
# no clipping, edges will be below the vertices anyway
add_shape("star",
  clip = shape_noclip,
  plot = mystar, parameters = list(vertex.norays = 5)
)
plot(g,
  vertex.shape = "star", vertex.color = rainbow(vcount(g)),
  vertex.size = seq(10, 20, length.out = vcount(g))
)
plot(g,
  vertex.shape = "star", vertex.color = rainbow(vcount(g)),
  vertex.size = seq(10, 20, length.out = vcount(g)),
  vertex.norays = rep(4:8, length.out = vcount(g))
)
```

`similarity` *Similarity measures of two vertices*

Description

These functions calculates similarity scores for vertices based on their connection patterns.

Usage

```
similarity(
  graph,
  vids = V(graph),
  mode = c("all", "out", "in", "total"),
  loops = FALSE,
  method = c("jaccard", "dice", "invlogweighted")
)
```

Arguments

graph	The input graph.
vids	The vertex ids for which the similarity is calculated.
mode	The type of neighboring vertices to use for the calculation, possible values: 'out', 'in', 'all'.
loops	Whether to include vertices themselves in the neighbor sets.
method	The method to use.

Details

The Jaccard similarity coefficient of two vertices is the number of common neighbors divided by the number of vertices that are neighbors of at least one of the two vertices being considered. The jaccard method calculates the pairwise Jaccard similarities for some (or all) of the vertices.

The Dice similarity coefficient of two vertices is twice the number of common neighbors divided by the sum of the degrees of the vertices. Methof dice calculates the pairwise Dice similarities for some (or all) of the vertices.

The inverse log-weighted similarity of two vertices is the number of their common neighbors, weighted by the inverse logarithm of their degrees. It is based on the assumption that two vertices should be considered more similar if they share a low-degree common neighbor, since high-degree common neighbors are more likely to appear even by pure chance. Isolated vertices will have zero similarity to any other vertex. Self-similarities are not calculated. See the following paper for more details: Lada A. Adamic and Eytan Adar: Friends and neighbors on the Web. Social Networks, 25(3):211-230, 2003.

Value

A length(vids) by length(vids) numeric matrix containing the similarity scores. This argument is ignored by the invlogweighted method.

Related documentation in the C library

similarity_dice(), similarity_inverse_log_weighted(), similarity_jaccard(), vcount()

Author(s)

Tamas Nepusz <ntamas@gmail.com> and Gabor Csardi <csardi.gabor@gmail.com> for the manual page.

References

Lada A. Adamic and Eytan Adar: Friends and neighbors on the Web. *Social Networks*, 25(3):211-230, 2003.

See Also

Other cocitation: `cocitation()`

Examples

```
g <- make_ring(5)
similarity(g, method = "dice")
similarity(g, method = "jaccard")
```

`simple_cycles` *Finds all simple cycles in a graph.*

Description

[Experimental]

This function lists all simple cycles in a graph within a range of cycle lengths. A cycle is called simple if it has no repeated vertices.

Multi-edges and self-loops are taken into account. Note that typical graphs have exponentially many cycles and the presence of multi-edges exacerbates this combinatorial explosion.

Usage

```
simple_cycles(
  graph,
  mode = c("out", "in", "all", "total"),
  min = NULL,
  max = NULL,
  ...,
  callback = NULL
)
```

Arguments

`graph`	The input graph.
`mode`	Character constant specifying how to handle directed graphs. `out` follows edge directions, `in` follows edges in the reverse direction, and `all` ignores edge directions. Ignored in undirected graphs.
`min`	Lower limit on cycle lengths to consider. `NULL` means no limit.
`max`	Upper limit on cycle lengths to consider. `NULL` means no limit.
`...`	These dots are for future extensions and must be empty.

callback — Optional function to call for each cycle found. If provided, the function should accept two arguments: vertices (integer vector of vertex IDs in the cycle) and edges (integer vector of edge IDs in the cycle). The function should return FALSE to continue the search or TRUE to stop it. If NULL (the default), all cycles are collected and returned as a list.

Important limitation: Callback functions must NOT call any igraph functions (including simple queries like vcount() or ecount()). Doing so will cause R to crash due to nested .Call() state corruption. Extract any needed graph information before calling the function with a callback, or use collector mode (the default) and process results afterward.

Value

If callback is NULL, returns a list with two elements: vertices (list of integer vectors with vertex IDs) and edges (list of integer vectors with edge IDs). If callback is provided, returns NULL invisibly.

If callback is NULL, returns a list with two elements: vertices (list of integer vectors with vertex IDs) and edges (list of integer vectors with edge IDs). If callback is provided, returns NULL invisibly.

Related documentation in the C library

simple_cycles(), simple_cycles_callback(), vcount(), edges(), get_eids(), ecount()

See Also

Graph cycles: feedback_arc_set(), feedback_vertex_set(), find_cycle(), girth(), has_eulerian_path(), is_acyclic(), is_dag()

Examples

```
g <- graph_from_literal(A -+ B -+ C -+ A -+ D -+ E +- F -+ A, E -+ E, A -+ F, simplify = FALSE)
simple_cycles(g)
simple_cycles(g, mode = "all") # ignore edge directions
simple_cycles(g, mode = "all", min = 2, max = 3) # limit cycle lengths
```

---

simplified *Constructor modifier to drop multiple and loop edges*

Description

Constructor modifier to drop multiple and loop edges

Usage

```
simplified()
```

See Also

Constructor modifiers (and related functions): make_(), sample_(), with_edge_(), with_graph_(), with_vertex_(), without_attr(), without_loops(), without_multiples()

Examples

```
sample_(pa(10, m = 3, algorithm = "bag"))
sample_(pa(10, m = 3, algorithm = "bag"), simplified())
```

---

simplify *Simple graphs*

Description

Simple graphs are graphs which do not contain loop and multiple edges.

Usage

```
simplify(
  graph,
  remove.multiple = TRUE,
  remove.loops = TRUE,
  edge.attr.comb = igraph_opt("edge.attr.comb")
)

is_simple(graph)

simplify_and_colorize(graph)
```

Arguments

graph	The graph to work on.
remove.multiple	Logical, whether the multiple edges are to be removed.
remove.loops	Logical, whether the loop edges are to be removed.
edge.attr.comb	Specifies what to do with edge attributes, if remove.multiple=TRUE. In this case many edges might be mapped to a single one in the new graph, and their attributes are combined. Please see attribute.combination() for details on this.

Details

A loop edge is an edge for which the two endpoints are the same vertex. Two edges are multiple edges if they have exactly the same two endpoints (for directed graphs order does matter). A graph is simple is it does not contain loop edges and multiple edges.

is_simple() checks whether a graph is simple.

`simplify()` removes the loop and/or multiple edges from a graph. If both `remove.loops` and `remove.multiple` are `TRUE` the function returns a simple graph.

`simplify_and_colorize()` constructs a new, simple graph from a graph and also sets a `color` attribute on both the vertices and the edges. The colors of the vertices represent the number of self-loops that were originally incident on them, while the colors of the edges represent the multiplicities of the same edges in the original graph. This allows one to take into account the edge multiplicities and the number of loop edges in the VF2 isomorphism algorithm. Other graph, vertex and edge attributes from the original graph are discarded as the primary purpose of this function is to facilitate the usage of multigraphs with the VF2 algorithm.

Value

a new graph object with the edges deleted.

Related documentation in the C library

`simplify()`, `is_simple()`, `simplify_and_colorize()`, `vcount()`, `edges()`, `get_eids()`, `ecount()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

`which_loop()`, `which_multiple()` and `count_multiple()`, `delete_edges()`, `delete_vertices()`

Other functions for manipulating graph structure: `+.igraph()`, `add_edges()`, `add_vertices()`, `complementer()`, `compose()`, `connect()`, `contract()`, `delete_edges()`, `delete_vertices()`, `difference()`, `difference.igraph()`, `disjoint_union()`, `edge()`, `igraph-minus`, `intersection()`, `intersection.igraph()`, `path()`, `permute()`, `rep.igraph()`, `reverse_edges()`, `transitive_closure()`, `union()`, `union.igraph()`, `vertex()`

Examples

```
g <- make_graph(c(1, 2, 1, 2, 3, 3))
is_simple(g)
is_simple(simplify(g, remove.loops = FALSE))
is_simple(simplify(g, remove.multiple = FALSE))
is_simple(simplify(g))
```

---

spectrum *Eigenvalues and eigenvectors of the adjacency matrix of a graph*

Description

Calculate selected eigenvalues and eigenvectors of a (supposedly sparse) graph.

Usage

```
spectrum(
  graph,
 algorithm = c("arpack", "auto", "lapack", "comp_auto", "comp_lapack", "comp_arpack"),
  which = list(),
  options = arpack_defaults()
)
```

Arguments

graph	The input graph, can be directed or undirected.
algorithm	The algorithm to use. Currently only arpack is implemented, which uses the ARPACK solver. See also arpack().
which	A list to specify which eigenvalues and eigenvectors to calculate. By default the leading (i.e. largest magnitude) eigenvalue and the corresponding eigenvector is calculated.
options	Options for the ARPACK solver. See arpack_defaults().

Details

The which argument is a list and it specifies which eigenvalues and corresponding eigenvectors to calculate: There are eight options:

1. Eigenvalues with the largest magnitude. Set pos to LM, and howmany to the number of eigenvalues you want.
2. Eigenvalues with the smallest magnitude. Set pos to SM and howmany to the number of eigenvalues you want.
3. Largest eigenvalues. Set pos to LA and howmany to the number of eigenvalues you want.
4. Smallest eigenvalues. Set pos to SA and howmany to the number of eigenvalues you want.
5. Eigenvalues from both ends of the spectrum. Set pos to BE and howmany to the number of eigenvalues you want. If howmany is odd, then one more eigenvalue is returned from the larger end.
6. Selected eigenvalues. This is not (yet) implemented currently.
7. Eigenvalues in an interval. This is not (yet) implemented.
8. All eigenvalues. This is not implemented yet. The standard eigen function does a better job at this, anyway.

Note that ARPACK might be unstable for graphs with multiple components, e.g. graphs with isolate vertices.

Value

Depends on the algorithm used.

For arpack a list with three entries is returned:

options See the return value for arpack() for a complete description.

values Numeric vector, the eigenvalues.

vectors Numeric matrix, with the eigenvectors as columns.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

as_adjacency_matrix() to create a (sparse) adjacency matrix.

Centrality measures: alpha_centrality(), authority_score(), betweenness(), closeness(), diversity(), eigen_centrality(), harmonic_centrality(), hits_scores(), page_rank(), power_centrality(), strength(), subgraph_centrality()

Examples

```
## Small example graph, leading eigenvector by default
kite <- make_graph("Krackhardt_kite")
spectrum(kite)[c("values", "vectors")]

## Double check
eigen(as_adjacency_matrix(kite, sparse = FALSE))$vectors[, 1]

## Should be the same as 'eigen_centrality' (but rescaled)
cor(eigen_centrality(kite)$vector, spectrum(kite)$vectors)

## Smallest eigenvalues
spectrum(kite, which = list(pos = "SM", howmany = 2))$values
```

---

split_join_distance *Split-join distance of two community structures*

Description

The split-join distance between partitions A and B is the sum of the projection distance of A from B and the projection distance of B from A. The projection distance is an asymmetric measure and it is defined as follows:

Usage

```
split_join_distance(comm1, comm2)
```

Arguments

comm1	The first community structure.
comm2	The second community structure.

Details

First, each set in partition A is evaluated against all sets in partition B. For each set in partition A, the best matching set in partition B is found and the overlap size is calculated. (Matching is quantified by the size of the overlap between the two sets). Then, the maximal overlap sizes for each set in A are summed together and subtracted from the number of elements in A.

The split-join distance will be returned as two numbers, the first is the projection distance of the first partition from the second, while the second number is the projection distance of the second partition from the first. This makes it easier to detect whether a partition is a subpartition of the other, since in this case, the corresponding distance will be zero.

Value

Two integer numbers, see details below.

Related documentation in the C library

`split_join_distance()`

References

van Dongen S: Performance criteria for graph clustering and Markov cluster experiments. Technical Report INS-R0012, National Research Institute for Mathematics and Computer Science in the Netherlands, Amsterdam, May 2000.

See Also

Community detection: `as_membership()`, `cluster_edge_betweenness()`, `cluster_fast_greedy()`, `cluster_fluid_communities()`, `cluster_infomap()`, `cluster_label_prop()`, `cluster_leading_eigen()`, `cluster_leiden()`, `cluster_louvain()`, `cluster_optimal()`, `cluster_spinglass()`, `cluster_walktrap()`, `compare()`, `groups()`, `make_clusters()`, `membership()`, `modularity.igraph()`, `plot_dendrogram()`, `voronoi_cells()`

---

st_cuts *List all (s,t)-cuts of a graph*

Description

List all (s,t)-cuts in a directed graph.

Usage

```
st_cuts(graph, source, target)
```

Arguments

`graph`	The input graph. It must be directed.
`source`	The source vertex.
`target`	The target vertex.

Details

Given a G directed graph and two, different and non-ajacent vertices, s and t, an (s, t)-cut is a set of edges, such that after removing these edges from G there is no directed path from s to t.

Value

A list with entries:

cuts A list of numeric vectors containing edge ids. Each vector is an (s, t)-cut.

partition1s A list of numeric vectors containing vertex ids, they correspond to the edge cuts. Each vertex set is a generator of the corresponding cut, i.e. in the graph $G = (V, E)$, the vertex set X and its complementer $V - X$, generates the cut that contains exactly the edges that go from X to $V - X$.

Related documentation in the C library

all_st_cuts(), vcount(), edges(), get_eids(), ecount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

JS Provan and DR Shier: A Paradigm for listing (s,t)-cuts in graphs, *Algorithmica* 15, 351–372, 1996.

See Also

Other flow: dominator_tree(), edge_connectivity(), is_min_separator(), is_separator(), max_flow(), min_cut(), min_separators(), min_st_separators(), st_min_cuts(), vertex_connectivity()

Examples

```
# A very simple graph
g <- graph_from_literal(a -+ b -+ c -+ d -+ e)
st_cuts(g, source = "a", target = "e")

# A somewhat more difficult graph
g2 <- graph_from_literal(
  s --+ a:b, a:b --+ t,
  a --+ 1:2:3, 1:2:3 --+ b
)
st_cuts(g2, source = "s", target = "t")
```

st_min_cuts *List all minimum (s,t)-cuts of a graph*

Description

Listing all minimum (s,t)-cuts of a directed graph, for given s and t.

Usage

```
st_min_cuts(graph, source, target, capacity = NULL)
```

Arguments

graph	The input graph. It must be directed.
source	The id of the source vertex.
target	The id of the target vertex.
capacity	Numeric vector giving the edge capacities. If this is NULL and the graph has a weight edge attribute, then this attribute defines the edge capacities. For forcing unit edge capacities, even for graphs that have a weight edge attribute, supply NA here.

Details

Given a G directed graph and two, different and non-ajacent vertices, s and t, an (s,t)-cut is a set of edges, such that after removing these edges from G there is no directed path from s to t.

The size of an (s,t)-cut is defined as the sum of the capacities (or weights) in the cut. For unweighted (=equally weighted) graphs, this is simply the number of edges.

An (s,t)-cut is minimum if it is of the smallest possible size.

Value

A list with entries:

value Numeric scalar, the size of the minimum cut(s).

cuts A list of numeric vectors containing edge ids. Each vector is a minimum (s,t)-cut.

partition1s A list of numeric vectors containing vertex ids, they correspond to the edge cuts. Each vertex set is a generator of the corresponding cut, i.e. in the graph $G=(V,E)$, the vertex set X and its complementer $V-X$, generates the cut that contains exactly the edges that go from X to $V-X$.

Related documentation in the C library

all_st_mincuts(), vcount(), edges(), get_eids(), ecount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

JS Provan and DR Shier: A Paradigm for listing (s,t)-cuts in graphs, *Algorithmica* 15, 351–372, 1996.

See Also

Other flow: `dominator_tree()`, `edge_connectivity()`, `is_min_separator()`, `is_separator()`, `max_flow()`, `min_cut()`, `min_separators()`, `min_st_separators()`, `st_cuts()`, `vertex_connectivity()`

Examples

```
# A difficult graph, from the Provan-Shier paper
g <- graph_from_literal(
  s --+ a:b, a:b --+ t,
  a --+ 1:2:3:4:5, 1:2:3:4:5 --+ b
)
st_min_cuts(g, source = "s", target = "t")
```

---

stochastic_matrix *Stochastic matrix of a graph*

Description

Retrieves the stochastic matrix of a graph of class `igraph`.

Usage

```
stochastic_matrix(
  graph,
  column.wise = FALSE,
  sparse = igraph_opt("sparsematrices")
)
```

Arguments

`graph`	The input graph. Must be of class `igraph`.
`column.wise`	If `FALSE`, then the rows of the stochastic matrix sum up to one; otherwise it is the columns.
`sparse`	Logical scalar, whether to return a sparse matrix. The `Matrix` package is needed for sparse matrices.

Details

Let M be an $n \times n$ adjacency matrix with real non-negative entries. Let us define $D = \text{diag}(\sum_i M_{1i}, \ldots, \sum_i M_{ni})$ The (row) stochastic matrix is defined as

$$W = D^{-1} M,$$

where it is assumed that D is non-singular. Column stochastic matrices are defined in a symmetric way.

Value

A regular matrix or a matrix of class `Matrix` if a `sparse` argument was `TRUE`.

Related documentation in the C library

`get_stochastic()`, `get_stochastic_sparse()`, `vcount()`, `edges()`, `get_eids()`, `ecount()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

`as_adjacency_matrix()`

Examples

```
library(Matrix)
## g is a large sparse graph
g <- sample_pa(n = 10^5, power = 2, directed = FALSE)
W <- stochastic_matrix(g, sparse = TRUE)

## a dense matrix here would probably not fit in the memory
class(W)

## may not be exactly 1, due to numerical errors
max(abs(rowSums(W)) - 1)
```

---

strength *Strength or weighted vertex degree*

Description

Summing up the edge weights of the adjacent edges for each vertex.

Usage

```
strength(
  graph,
  vids = V(graph),
  mode = c("all", "out", "in", "total"),
  loops = TRUE,
  weights = NULL
)
```

Arguments

graph	The input graph.
vids	The vertices for which the strength will be calculated.
mode	Character string, "out" for out-degree, "in" for in-degree or "all" for the sum of the two. For undirected graphs this argument is ignored.
loops	Logical; whether the loop edges are also counted.
weights	Weight vector. If the graph has a `weight` edge attribute, then this is used by default. If the graph does not have a `weight` edge attribute and this argument is `NULL`, then a `degree()` is called. If this is `NA`, then no edge weights are used (even if the graph has a `weight` edge attribute).

Value

A numeric vector giving the strength of the vertices.

Related documentation in the C library

`strength()`, `vcount()`, `edges()`, `get_eids()`, `ecount()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Alain Barrat, Marc Barthelemy, Romualdo Pastor-Satorras, Alessandro Vespignani: The architecture of complex weighted networks, Proc. Natl. Acad. Sci. USA 101, 3747 (2004)

See Also

`degree()` for the unweighted version.

Centrality measures: `alpha_centrality()`, `authority_score()`, `betweenness()`, `closeness()`, `diversity()`, `eigen_centrality()`, `harmonic_centrality()`, `hits_scores()`, `page_rank()`, `power_centrality()`, `spectrum()`, `subgraph_centrality()`

Examples

```
g <- make_star(10)
E(g)$weight <- seq(ecount(g))
strength(g)
strength(g, mode = "out")
strength(g, mode = "in")

# No weights
g <- make_ring(10)
strength(g)
```

subcomponent *In- or out- component of a vertex*

Description

Finds all vertices reachable from a given vertex, or the opposite: all vertices from which a given vertex is reachable via a directed path.

Usage

```
subcomponent(graph, v, mode = c("all", "out", "in"))
```

Arguments

graph	The graph to analyze.
v	The vertex to start the search from.
mode	Character string, either "in", "out" or "all". If "in" all vertices from which v is reachable are listed. If "out" all vertices reachable from v are returned. If "all" returns the union of these. It is ignored for undirected graphs.

Details

A breadth-first search is conducted starting from vertex v.

Value

Numeric vector, the ids of the vertices in the same component as v.

Related documentation in the C library

vcount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

components()

Other structural.properties: bfs(), component_distribution(), connect(), constraint(), coreness(), degree(), dfs(), distance_table(), edge_density(), feedback_arc_set(), feedback_vertex_set(), girth(), is_acyclic(), is_dag(), is_matching(), k_shortest_paths(), knn(), reciprocity(), subgraph(), topo_sort(), transitivity(), unfold_tree(), which_multiple(), which_mutual()

Examples

```
g <- sample_gnp(100, 1 / 200)
subcomponent(g, 1, "in")
subcomponent(g, 1, "out")
subcomponent(g, 1, "all")
```

---

subgraph *Subgraph of a graph*

Description

subgraph() creates a subgraph of a graph, containing only the specified vertices and all the edges among them.

Usage

```
subgraph(graph, vids)

induced_subgraph(
  graph,
  vids,
  impl = c("auto", "copy_and_delete", "create_from_scratch")
)

subgraph_from_edges(graph, eids, delete.vertices = TRUE)
```

Arguments

graph	The original graph.
vids	Numeric vector, the vertices of the original graph which will form the subgraph.
impl	Character scalar, to choose between two implementation of the subgraph calculation. 'copy_and_delete' copies the graph first, and then deletes the vertices and edges that are not included in the result graph. 'create_from_scratch' searches for all vertices and edges that must be kept and then uses them to create the graph from scratch. 'auto' chooses between the two implementations automatically, using heuristics based on the size of the original and the result graph.
eids	The edge ids of the edges that will be kept in the result graph.
delete.vertices	Logical scalar, whether to remove vertices that do not have any adjacent edges in eids.

Details

induced_subgraph() calculates the induced subgraph of a set of vertices in a graph. This means that exactly the specified vertices and all the edges between them will be kept in the result graph.

subgraph_from_edges() calculates the subgraph of a graph. For this function one can specify the vertices and edges to keep. This function will be renamed to subgraph() in the next major version of igraph.

The subgraph() function currently does the same as induced_subgraph() (assuming 'auto' as the impl argument), but this behaviour is deprecated. In the next major version, subgraph() will overtake the functionality of subgraph_from_edges().

Value

A new graph object.

Related documentation in the C library

induced_subgraph(), vcount(), subgraph_from_edges(), get_eids(), edges(), ecount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Other structural.properties: bfs(), component_distribution(), connect(), constraint(), coreness(), degree(), dfs(), distance_table(), edge_density(), feedback_arc_set(), feedback_vertex_set(), girth(), is_acyclic(), is_dag(), is_matching(), k_shortest_paths(), knn(), reciprocity(), subcomponent(), topo_sort(), transitivity(), unfold_tree(), which_multiple(), which_mutual()

Examples

```
g <- make_ring(10)
g2 <- induced_subgraph(g, 1:7)
g3 <- subgraph_from_edges(g, 1:5)
```

---

subgraph_centrality *Find subgraph centrality scores of network positions*

Description

Subgraph centrality of a vertex measures the number of subgraphs a vertex participates in, weighting them according to their size.

Usage

```
subgraph_centrality(graph, diag = FALSE)
```

Arguments

graph	The input graph. It will be treated as undirected.
diag	Boolean scalar, whether to include the diagonal of the adjacency matrix in the analysis. Giving FALSE here effectively eliminates the loops edges from the graph before the calculation.

Details

The subgraph centrality of a vertex is defined as the number of closed walks originating at the vertex, where longer walks are downweighted by the factorial of their length.

Currently the calculation is performed by explicitly calculating all eigenvalues and eigenvectors of the adjacency matrix of the graph. This effectively means that the measure can only be calculated for small graphs.

Value

A numeric vector, the subgraph centrality scores of the vertices.

Related documentation in the C library

is_directed(), get_adjacency(), get_adjacency_sparse(), vcount(), edges(), get_eids(), ecount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com> based on the Matlab code by Ernesto Estrada

References

Ernesto Estrada, Juan A. Rodriguez-Velazquez: Subgraph centrality in Complex Networks. *Physical Review E* 71, 056103 (2005).

See Also

eigen_centrality(), page_rank()

Centrality measures: alpha_centrality(), authority_score(), betweenness(), closeness(), diversity(), eigen_centrality(), harmonic_centrality(), hits_scores(), page_rank(), power_centrality(), spectrum(), strength()

Examples

```
g <- sample_pa(100, m = 4, dir = FALSE)
sc <- subgraph_centrality(g)
cor(degree(g), sc)
```

subgraph_isomorphic *Decide if a graph is subgraph isomorphic to another one*

Description

Decide if a graph is subgraph isomorphic to another one

Usage

```
subgraph_isomorphic(pattern, target, method = c("auto", "lad", "vf2"), ...)

is_subgraph_isomorphic_to(
  pattern,
  target,
  method = c("auto", "lad", "vf2"),
  ...
)
```

Arguments

pattern	The smaller graph, it might be directed or undirected. Undirected graphs are treated as directed graphs with mutual edges.
target	The bigger graph, it might be directed or undirected. Undirected graphs are treated as directed graphs with mutual edges.
method	The method to use. Possible values: 'auto', 'lad', 'vf2'. See their details below.
...	Additional arguments, passed to the various methods.

Value

Logical scalar, TRUE if the pattern is isomorphic to a (possibly induced) subgraph of target.

'auto' method

This method currently selects 'lad', always, as it seems to be superior on most graphs.

'lad' method

This is the LAD algorithm by Solnon, see the reference below. It has the following extra arguments:

domains If not NULL, then it specifies matching restrictions. It must be a list of target vertex sets, given as numeric vertex ids or symbolic vertex names. The length of the list must be vcount(pattern) and for each vertex in pattern it gives the allowed matching vertices in target. Defaults to NULL.

induced Logical scalar, whether to search for an induced subgraph. It is FALSE by default.

time.limit The processor time limit for the computation, in seconds. It defaults to Inf, which means no limit.

'vf2' method

This method uses the VF2 algorithm by Cordella, Foggia et al., see references below. It supports vertex and edge colors and have the following extra arguments:

vertex.color1, vertex.color2 Optional integer vectors giving the colors of the vertices for colored graph isomorphism. If they are not given, but the graph has a "color" vertex attribute, then it will be used. If you want to ignore these attributes, then supply `NULL` for both of these arguments. See also examples below.

edge.color1, edge.color2 Optional integer vectors giving the colors of the edges for edge-colored (sub)graph isomorphism. If they are not given, but the graph has a "color" edge attribute, then it will be used. If you want to ignore these attributes, then supply `NULL` for both of these arguments.

Related documentation in the C library

`subisomorphic_vf2()`, `vcount()`, `edges()`, `get_eids()`, `ecount()`

References

LP Cordella, P Foggia, C Sansone, and M Vento: An improved algorithm for matching large graphs, *Proc. of the 3rd IAPR TC-15 Workshop on Graphbased Representations in Pattern Recognition*, 149–159, 2001.

C. Solnon: AllDifferent-based Filtering for Subgraph Isomorphism, *Artificial Intelligence* 174(12-13):850–864, 2010.

See Also

Other graph isomorphism: `canonical_permutation()`, `count_isomorphisms()`, `count_subgraph_isomorphisms()`, `graph_from_isomorphism_class()`, `isomorphic()`, `isomorphism_class()`, `isomorphisms()`, `subgraph_isomorphisms()`

Examples

```
# A LAD example
pattern <- make_graph(
  ~ 1:2:3:4:5,
  1 - 2:5, 2 - 1:5:3, 3 - 2:4, 4 - 3:5, 5 - 4:2:1
)
target <- make_graph(
  ~ 1:2:3:4:5:6:7:8:9,
  1 - 2:5:7, 2 - 1:5:3, 3 - 2:4, 4 - 3:5:6:8:9,
  5 - 1:2:4:6:7, 6 - 7:5:4:9, 7 - 1:5:6,
  8 - 4:9, 9 - 6:4:8
)
domains <- list(
  `1` = c(1, 3, 9), `2` = c(5, 6, 7, 8), `3` = c(2, 4, 6, 7, 8, 9),
  `4` = c(1, 3, 9), `5` = c(2, 4, 8, 9)
)
subgraph_isomorphisms(pattern, target)
subgraph_isomorphisms(pattern, target, induced = TRUE)
```

```
subgraph_isomorphisms(pattern, target, domains = domains)

# Directed LAD example
pattern <- make_graph(~ 1:2:3, 1 -+ 2:3)
dring <- make_ring(10, directed = TRUE)
subgraph_isomorphic(pattern, dring)
```

`subgraph_isomorphisms` *All isomorphic mappings between a graph and subgraphs of another graph*

Description

All isomorphic mappings between a graph and subgraphs of another graph

Usage

```
subgraph_isomorphisms(
  pattern,
  target,
  method = c("lad", "vf2"),
  ...,
  callback = NULL
)
```

Arguments

`pattern`	The smaller graph, it might be directed or undirected. Undirected graphs are treated as directed graphs with mutual edges.
`target`	The bigger graph, it might be directed or undirected. Undirected graphs are treated as directed graphs with mutual edges.
`method`	The method to use. Possible values: 'auto', 'lad', 'vf2'. See their details below.
`...`	Additional arguments, passed to the various methods.
`callback`	Optional callback function to call for each subisomorphism found. If provided, the function should accept two arguments: `map12` (integer vector mapping vertex IDs from pattern to target, 1-based indexing) and `map21` (integer vector mapping vertex IDs from target to pattern, 1-based indexing). The function should return `FALSE` to continue the search or `TRUE` to stop it. If `NULL` (the default), all subisomorphisms are collected and returned as a list. Only supported for `method = "vf2"`. **Important limitation:** Callback functions must NOT call any igraph functions (including simple queries like `vcount()` or `ecount()`). Doing so will cause R to crash due to reentrancy issues. Extract any needed graph information before calling the function with a callback, or use collector mode (the default) and process results afterward.

Value

If `callback` is `NULL`, returns a list of vertex sequences, corresponding to all mappings from the pattern graph to the target graph. If `callback` is provided, returns `NULL` invisibly.

'lad' method

This is the LAD algorithm by Solnon, see the reference below. It has the following extra arguments:

domains If not `NULL`, then it specifies matching restrictions. It must be a list of `target` vertex sets, given as numeric vertex ids or symbolic vertex names. The length of the list must be `vcount(pattern)` and for each vertex in `pattern` it gives the allowed matching vertices in `target`. Defaults to `NULL`.

induced Logical scalar, whether to search for an induced subgraph. It is `FALSE` by default.

time.limit The processor time limit for the computation, in seconds. It defaults to `Inf`, which means no limit.

'vf2' method

This method uses the VF2 algorithm by Cordella, Foggia et al., see references below. It supports vertex and edge colors and have the following extra arguments:

vertex.color1, vertex.color2 Optional integer vectors giving the colors of the vertices for colored graph isomorphism. If they are not given, but the graph has a "color" vertex attribute, then it will be used. If you want to ignore these attributes, then supply `NULL` for both of these arguments. See also examples below.

edge.color1, edge.color2 Optional integer vectors giving the colors of the edges for edge-colored (sub)graph isomorphism. If they are not given, but the graph has a "color" edge attribute, then it will be used. If you want to ignore these attributes, then supply `NULL` for both of these arguments.

Related documentation in the C library

`get_subisomorphisms_vf2_callback()`, `get_subisomorphisms_vf2()`, `vcount()`, `edges()`, `get_eids()`, `ecount()`

See Also

Other graph isomorphism: `canonical_permutation()`, `count_isomorphisms()`, `count_subgraph_isomorphisms()`, `graph_from_isomorphism_class()`, `isomorphic()`, `isomorphism_class()`, `isomorphisms()`, `subgraph_isomorphic()`

tail_of *Tails of the edge(s) in a graph*

Description

For undirected graphs, head and tail is not defined. In this case `tail_of()` returns vertices incident to the supplied edges, and `head_of()` returns the other end(s) of the edge(s).

Usage

```
tail_of(graph, es)
```

Arguments

`graph`	The input graph.
`es`	The edges to query.

Value

A vertex sequence with the tail(s) of the edge(s).

Related documentation in the C library

`edges()`, `vcount()`, `get_eids()`, `ecount()`

See Also

Other structural queries: `[.igraph()`, `[[.igraph()`, `adjacent_vertices()`, `are_adjacent()`, `ends()`, `get_edge_ids()`, `gorder()`, `gsize()`, `head_of()`, `incident()`, `incident_edges()`, `is_directed()`, `neighbors()`

time_bins *SIR model on graphs*

Description

Run simulations for an SIR (susceptible-infected-recovered) model, on a graph

Usage

```
time_bins(x, middle = TRUE)

## S3 method for class 'sir'
time_bins(x, middle = TRUE)

## S3 method for class 'sir'
median(x, na.rm = FALSE, ...)

## S3 method for class 'sir'
quantile(x, comp = c("NI", "NS", "NR"), prob, ...)

sir(graph, beta, gamma, no.sim = 100)
```

Arguments

x	A sir object, returned by the sir() function.
middle	Logical scalar, whether to return the middle of the time bins, or the boundaries.
na.rm	Logical scalar, whether to ignore NA values. sir objects do not contain any NA values currently, so this argument is effectively ignored.
...	Additional arguments, ignored currently.
comp	Character scalar. The component to calculate the quantile of. NI is infected agents, NS is susceptibles, NR stands for recovered.
prob	Numeric vector of probabilities, in [0,1], they specify the quantiles to calculate.
graph	The graph to run the model on. If directed, then edge directions are ignored and a warning is given.
beta	Non-negative scalar. The rate of infection of an individual that is susceptible and has a single infected neighbor. The infection rate of a susceptible individual with n infected neighbors is n times beta. Formally this is the rate parameter of an exponential distribution.
gamma	Positive scalar. The rate of recovery of an infected individual. Formally, this is the rate parameter of an exponential distribution.
no.sim	Integer scalar, the number simulation runs to perform.

Details

The SIR model is a simple model from epidemiology. The individuals of the population might be in three states: susceptible, infected and recovered. Recovered people are assumed to be immune to the disease. Susceptibles become infected with a rate that depends on their number of infected neighbors. Infected people become recovered with a constant rate.

The function sir() simulates the model. This function runs multiple simulations, all starting with a single uniformly randomly chosen infected individual. A simulation is stopped when no infected individuals are left.

Function time_bins() bins the simulation steps, using the Freedman-Diaconis heuristics to determine the bin width.

Function median and quantile calculate the median and quantiles of the results, respectively, in bins calculated with time_bins().

Value

For sir() the results are returned in an object of class 'sir', which is a list, with one element for each simulation. Each simulation is itself a list with the following elements. They are all numeric vectors, with equal length:

times The times of the events.

NS The number of susceptibles in the population, over time.

NI The number of infected individuals in the population, over time.

NR The number of recovered individuals in the population, over time.

Function time_bins() returns a numeric vector, the middle or the boundaries of the time bins, depending on the middle argument.

median returns a list of three named numeric vectors, NS, NI and NR. The names within the vectors are created from the time bins.

quantile returns the same vector as median (but only one, the one requested) if only one quantile is requested. If multiple quantiles are requested, then a list of these vectors is returned, one for each quantile.

Related documentation in the C library

sir()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>. Eric Kolaczyk (https://kolaczyk.github.io/) wrote the initial version in R.

References

Bailey, Norman T. J. (1975). The mathematical theory of infectious diseases and its applications (2nd ed.). London: Griffin.

See Also

plot.sir() to conveniently plot the results

Processes on graphs: plot.sir()

Examples

```
g <- sample_gnm(100, 100)
sm <- sir(g, beta = 5, gamma = 1)
plot(sm)
```

tkplot *Interactive plotting of graphs*

Description

tkplot() and its companion functions serve as an interactive graph drawing facility. Not all parameters of the plot can be changed interactively right now though, e.g. the colors of vertices, edges, and also others have to be pre-defined.

Usage

```
tkplot(graph, canvas.width = 450, canvas.height = 450, ...)

tk_close(tkp.id, window.close = TRUE)

tk_off()

tk_fit(tkp.id, width = NULL, height = NULL)

tk_center(tkp.id)

tk_reshape(tkp.id, newlayout, ..., params)

tk_postscript(tkp.id)

tk_coords(tkp.id, norm = FALSE)

tk_set_coords(tkp.id, coords)

tk_rotate(tkp.id, degree = NULL, rad = NULL)

tk_canvas(tkp.id)
```

Arguments

graph	The graph to plot.
canvas.width, canvas.height	The size of the tkplot drawing area.
...	Additional plotting parameters. See igraph.plotting for the complete list.
tkp.id	The id of the tkplot window to close/reshape/etc.
window.close	Leave this on the default value.
width	The width of the rectangle for generating new coordinates.
height	The height of the rectangle for generating new coordinates.
newlayout	The new layout, see the layout parameter of tkplot.
params	Extra parameters in a list, to pass to the layout function.

norm	Logical, should we norm the coordinates.
coords	Two-column numeric matrix, the new coordinates of the vertices, in absolute coordinates.
degree	The degree to rotate the plot.
rad	The degree to rotate the plot, in radian.

Details

tkplot() is an interactive graph drawing facility. It is not very well developed at this stage, but it should be still useful.

It's handling should be quite straightforward most of the time, here are some remarks and hints.

There are different popup menus, activated by the right mouse button, for vertices and edges. Both operate on the current selection if the vertex/edge under the cursor is part of the selection and operate on the vertex/edge under the cursor if it is not.

One selection can be active at a time, either a vertex or an edge selection. A vertex/edge can be added to a selection by holding the control key while clicking on it with the left mouse button. Doing this again deselect the vertex/edge.

Selections can be made also from the "Select" menu. The "Select some vertices" dialog allows to give an expression for the vertices to be selected: this can be a list of numeric R expessions separated by commas, like 1,2:10,12,14,15 for example. Similarly in the "Select some edges" dialog two such lists can be given and all edges connecting a vertex in the first list to one in the second list will be selected.

In the color dialog a color name like 'orange' or RGB notation can also be used.

The tkplot() command creates a new Tk window with the graphical representation of graph. The command returns an integer number, the tkplot id. The other commands utilize this id to be able to query or manipulate the plot.

tk_close() closes the Tk plot with id tkp.id.

tk_off() closes all Tk plots.

tk_fit() fits the plot to the given rectangle (width and height), if some of these are NULL the actual physical width od height of the plot window is used.

tk_reshape() applies a new layout to the plot, its optional parameters will be collected to a list analogous to layout.par.

tk_postscript() creates a dialog window for saving the plot in postscript format.

tk_canvas() returns the Tk canvas object that belongs to a graph plot. The canvas can be directly manipulated then, e.g. labels can be added, it could be saved to a file programmatically, etc. See an example below.

tk_coords() returns the coordinates of the vertices in a matrix. Each row corresponds to one vertex.

tk_set_coords() sets the coordinates of the vertices. A two-column matrix specifies the new positions, with each row corresponding to a single vertex.

tk_center() shifts the figure to the center of its plot window.

tk_rotate() rotates the figure, its parameter can be given either in degrees or in radians.

tkplot.center tkplot.rotate

Value

`tkplot()` returns an integer, the id of the plot, this can be used to manipulate it from the command line.

`tk_canvas()` returns `tkwin` object, the Tk canvas.

`tk_coords()` returns a matrix with the coordinates.

`tk_close()`, `tk_off()`, `tk_fit()`, `tk_reshape()`, `tk_postscript()`, `tk_center()` and `tk_rotate()` return `NULL` invisibly.

Examples

```
g <- make_ring(10)
tkplot(g)

## Saving a tkplot() to a file programmatically
g <- make_star(10, center=10)
E(g)$width <- sample(1:10, ecount(g), replace=TRUE)
lay <- layout_nicely(g)

id <- tkplot(g, layout=lay)
canvas <- tk_canvas(id)
tcltk::tkpostscript(canvas, file="/tmp/output.eps")
tk_close(id)

## Setting the coordinates and adding a title label
g <- make_ring(10)
id <- tkplot(make_ring(10), canvas.width=450, canvas.height=500)

canvas <- tk_canvas(id)
padding <- 20
coords <- norm_coords(layout_in_circle(g), 0+padding, 450-padding,
                      50+padding, 500-padding)
tk_set_coords(id, coords)

width <- as.numeric(tkcget(canvas, "-width"))
height <- as.numeric(tkcget(canvas, "-height"))
tkcreate(canvas, "text", width/2, 25, text="My title",
         justify="center", font=tcltk::tkfont.create(family="helvetica",
         size=20,weight="bold"))
```

Related documentation in the C library

`vcount()`, `is_directed()`, `get_edgelist()`, `edges()`, `get_eids()`, `ecount()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

plot.igraph(), layout()

---

to_prufer *Convert a tree graph to its Prüfer sequence*

Description

to_prufer() converts a tree graph into its Prüfer sequence.

Usage

```
to_prufer(graph)
```

Arguments

graph	The graph to convert to a Prüfer sequence

Details

The Prüfer sequence of a tree graph with n labeled vertices is a sequence of n-2 numbers, constructed as follows. If the graph has more than two vertices, find a vertex with degree one, remove it from the tree and add the label of the vertex that it was connected to to the sequence. Repeat until there are only two vertices in the remaining graph.

Value

The Prüfer sequence of the graph, represented as a numeric vector of vertex IDs in the sequence.

Related documentation in the C library

to_prufer()

See Also

make_from_prufer() to construct a graph from its Prüfer sequence

Other trees: is_forest(), is_tree(), make_from_prufer(), sample_spanning_tree()

Examples

```
g <- make_tree(13, 3)
to_prufer(g)
```

topo_sort *Topological sorting of vertices in a graph*

Description

A topological sorting of a directed acyclic graph is a linear ordering of its nodes where each node comes before all nodes to which it has edges.

Usage

```
topo_sort(graph, mode = c("out", "all", "in"))
```

Arguments

graph	The input graph, should be directed
mode	Specifies how to use the direction of the edges. For "out", the sorting order ensures that each node comes before all nodes to which it has edges, so nodes with no incoming edges go first. For "in", it is quite the opposite: each node comes before all nodes from which it receives edges. Nodes with no outgoing edges go first.

Details

Every DAG has at least one topological sort, and may have many. This function returns a possible topological sort among them. If the graph is not acyclic (it has at least one cycle), a partial topological sort is returned and a warning is issued.

Value

A vertex sequence (by default, but see the `return.vs.es` option of `igraph_options()`) containing vertices in topologically sorted order.

Related documentation in the C library

`topological_sorting()`, `vcount()`

Author(s)

Tamas Nepusz <ntamas@gmail.com> and Gabor Csardi <csardi.gabor@gmail.com> for the R interface

See Also

Other structural.properties: `bfs()`, `component_distribution()`, `connect()`, `constraint()`, `coreness()`, `degree()`, `dfs()`, `distance_table()`, `edge_density()`, `feedback_arc_set()`, `feedback_vertex_set()`, `girth()`, `is_acyclic()`, `is_dag()`, `is_matching()`, `k_shortest_paths()`, `knn()`, `reciprocity()`, `subcomponent()`, `subgraph()`, `transitivity()`, `unfold_tree()`, `which_multiple()`, `which_mutual()`

Examples

```
g <- sample_pa(100)
topo_sort(g)
```

`transitive_closure` *Transitive closure of a graph*

Description

[Experimental]

Computes the transitive closure of a graph. The resulting graph will have an edge from vertex i to vertex j if j is reachable from i in the original graph.

The transitive closure of a graph is a new graph where there is an edge between any two vertices if there is a path between them in the original graph. For directed graphs, an edge from i to j is added if there is a directed path from i to j. For undirected graphs, this is equivalent to connecting all vertices that are in the same connected component.

Usage

```
transitive_closure(graph)
```

Arguments

`graph`	The input graph. It can be directed or undirected.

Value

A new graph object representing the transitive closure. The returned graph will have the same directedness as the input.

Related documentation in the C library

`transitive_closure()`

Author(s)

Fabio Zanini <fabio.zanini@unsw.edu.au>

See Also

`distances()`, `are_adjacent()`

Other functions for manipulating graph structure: `+.igraph()`, `add_edges()`, `add_vertices()`, `complementer()`, `compose()`, `connect()`, `contract()`, `delete_edges()`, `delete_vertices()`, `difference()`, `difference.igraph()`, `disjoint_union()`, `edge()`, `igraph-minus`, `intersection()`, `intersection.igraph()`, `path()`, `permute()`, `rep.igraph()`, `reverse_edges()`, `simplify()`, `union()`, `union.igraph()`, `vertex()`

Examples

```
# Directed graph
g <- make_graph(c(1, 2, 2, 3, 3, 4))
tc <- transitive_closure(g)
# The closure has edges 1->2, 1->3, 1->4, 2->3, 2->4, 3->4
print_all(tc)

# Undirected graph - connects all vertices in same component
g2 <- make_graph(c(1, 2, 3, 4), directed = FALSE)
tc2 <- transitive_closure(g2)
# Full graph on vertices 1, 2 and full graph on vertices 3, 4
print_all(tc2)
```

---

transitivity *Transitivity of a graph*

Description

Transitivity measures the probability that the adjacent vertices of a vertex are connected. This is sometimes also called the clustering coefficient.

Usage

```
transitivity(
  graph,
  type = c("undirected", "global", "globalundirected", "localundirected", "local",
    "average", "localaverage", "localaverageundirected", "barrat", "weighted"),
  vids = NULL,
  weights = NULL,
  isolates = c("NaN", "zero")
)
```

Arguments

graph	The graph to analyze.
type	The type of the transitivity to calculate. Possible values:

"global" The global transitivity of an undirected graph. This is simply the ratio of the count of triangles and connected triples in the graph. In directed graphs, edge directions are ignored.

"local" The local transitivity of an undirected graph. It is calculated for each vertex given in the `vids` argument. The local transitivity of a vertex is the ratio of the count of triangles connected to the vertex and the triples centered on the vertex. In directed graphs, edge directions are ignored.

"undirected" This is the same as `global`.

"globalundirected" This is the same as `global`.

"localundirected" This is the same as `local`.

"barrat" The weighted transitivity as defined by A. Barrat. See details below.

"weighted" The same as `barrat`.

`vids` The vertex ids for the local transitivity will be calculated. This will be ignored for global transitivity types. The default value is `NULL`, in this case all vertices are considered. It is slightly faster to supply `NULL` here than `V(graph)`.

`weights` Optional weights for weighted transitivity. It is ignored for other transitivity measures. If it is `NULL` (the default) and the graph has a `weight` edge attribute, then it is used automatically.

`isolates` Character scalar, for local versions of transitivity, it defines how to treat vertices with degree zero and one. If it is 'NaN' then their local transitivity is reported as `NaN` and they are not included in the averaging, for the transitivity types that calculate an average. If there are no vertices with degree two or higher, then the averaging will still result `NaN`. If it is 'zero', then we report 0 transitivity for them, and they are included in the averaging, if an average is calculated. For the global transitivity, it controls how to handle graphs with no connected triplets: `NaN` or zero will be returned according to the respective setting.

Details

Note that there are essentially two classes of transitivity measures, one is a vertex-level, the other a graph level property.

There are several generalizations of transitivity to weighted graphs, here we use the definition by A. Barrat, this is a local vertex-level quantity, its formula is

$$C_i^w = \frac{1}{s_i(k_i-1)} \sum_{j,h} \frac{w_{ij}+w_{ih}}{2} a_{ij}a_{ih}a_{jh}$$

s_i is the strength of vertex i, see `strength()`, a_{ij} are elements of the adjacency matrix, k_i is the vertex degree, w_{ij} are the weights.

This formula gives back the normal not-weighted local transitivity if all the edge weights are the same.

The `barrat` type of transitivity does not work for graphs with multiple and/or loop edges. If you want to calculate it for a directed graph, call `as_undirected()` with the `collapse` mode first.

Value

For 'global' a single number, or `NaN` if there are no connected triples in the graph.

For 'local' a vector of transitivity scores, one for each vertex in 'vids'.

Related documentation in the C library

`transitivity_undirected()`, `transitivity_local_undirected()`, `transitivity_avglocal_undirected()`, `transitivity_barrat()`, `vcount()`, `edges()`, `get_eids()`, `ecount()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Wasserman, S., and Faust, K. (1994). *Social Network Analysis: Methods and Applications.* Cambridge: Cambridge University Press.

Alain Barrat, Marc Barthelemy, Romualdo Pastor-Satorras, Alessandro Vespignani: The architecture of complex weighted networks, Proc. Natl. Acad. Sci. USA 101, 3747 (2004)

See Also

Other structural.properties: bfs(), component_distribution(), connect(), constraint(), coreness(), degree(), dfs(), distance_table(), edge_density(), feedback_arc_set(), feedback_vertex_set(), girth(), is_acyclic(), is_dag(), is_matching(), k_shortest_paths(), knn(), reciprocity(), subcomponent(), subgraph(), topo_sort(), unfold_tree(), which_multiple(), which_mutual()

Examples

```
g <- make_ring(10)
transitivity(g)
g2 <- sample_gnp(1000, 10 / 1000)
transitivity(g2) # this is about 10/1000

# Weighted version, the figure from the Barrat paper
gw <- graph_from_literal(A - B:C:D:E, B - C:D, C - D)
E(gw)$weight <- 1
E(gw)[V(gw)[name == "A"] %--% V(gw)[name == "E"]]$weight <- 5
transitivity(gw, vids = "A", type = "local")
transitivity(gw, vids = "A", type = "weighted")

# Weighted reduces to "local" if weights are the same
gw2 <- sample_gnp(1000, 10 / 1000)
E(gw2)$weight <- 1
t1 <- transitivity(gw2, type = "local")
t2 <- transitivity(gw2, type = "weighted")
all(is.na(t1) == is.na(t2))
all(na.omit(t1 == t2))
```

---

triad_census *Triad census, subgraphs with three vertices*

Description

This function counts the different induced subgraphs of three vertices in a graph.

Usage

```
triad_census(graph)
```

Arguments

graph	The input graph, it should be directed. An undirected graph results a warning, and undefined results.

Details

Triad census was defined by David and Leinhardt (see References below). Every triple of vertices (A, B, C) are classified into the 16 possible states:

003 A,B,C, the empty graph.

012 A->B, C, the graph with a single directed edge.

102 A<->B, C, the graph with a mutual connection between two vertices.

021D A<-B->C, the out-star.

021U A->B<-C, the in-star.

021C A->B->C, directed line.

111D A<->B<-C.

111U A<->B->C.

030T A->B<-C, A->C.

030C A<-B<-C, A->C.

201 A<->B<->C.

120D A<-B->C, A<->C.

120U A->B<-C, A<->C.

120C A->B->C, A<->C.

210 A->B<->C, A<->C.

300 A<->B<->C, A<->C, the complete graph.

This functions uses the RANDESU motif finder algorithm to find and count the subgraphs, see `motifs()`.

Value

A numeric vector, the subgraph counts, in the order given in the above description.

Related documentation in the C library

`triad_census()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

See also Davis, J.A. and Leinhardt, S. (1972). The Structure of Positive Interpersonal Relations in Small Groups. In J. Berger (Ed.), Sociological Theories in Progress, Volume 2, 218-251. Boston: Houghton Mifflin.

See Also

dyad_census() for classifying binary relationships, motifs() for the underlying implementation.

Examples

```
g <- sample_gnm(15, 45, directed = TRUE)
triad_census(g)
```

---

triangles *Find triangles in graphs*

Description

Count how many triangles a vertex is part of, in a graph, or just list the triangles of a graph.

Usage

```
triangles(graph)

count_triangles(graph, vids = V(graph))
```

Arguments

graph	The input graph. It might be directed, but edge directions are ignored.
vids	The vertices to query, all of them by default. This might be a vector of numeric ids, or a character vector of symbolic vertex names for named graphs.

Details

triangles() lists all triangles of a graph. For efficiency, all triangles are returned in a single vector. The first three vertices belong to the first triangle, etc.

count_triangles() counts how many triangles a vertex is part of.

Value

For triangles() a numeric vector of vertex ids, the first three vertices belong to the first triangle found, etc.

For count_triangles() a numeric vector, the number of triangles for all vertices queried.

Related documentation in the C library

list_triangles(), vcount(), count_adjacent_triangles()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

transitivity()

Examples

```
## A small graph
kite <- make_graph("Krackhardt_Kite")
plot(kite)
matrix(triangles(kite), nrow = 3)

## Adjacenct triangles
atri <- count_triangles(kite)
plot(kite, vertex.label = atri)

## Always true
sum(count_triangles(kite)) == length(triangles(kite))

## Should match, local transitivity is the
## number of adjacent triangles divided by the number
## of adjacency triples
transitivity(kite, type = "local")
count_triangles(kite) / (degree(kite) * (degree(kite) - 1) / 2)
```

---

unfold_tree *Convert a general graph into a forest*

Description

Perform a breadth-first search on a graph and convert it into a tree or forest by replicating vertices that were found more than once.

Usage

```
unfold_tree(graph, mode = c("all", "out", "in", "total"), roots)
```

Arguments

graph	The input graph, it can be either directed or undirected.
mode	Character string, defined the types of the paths used for the breadth-first search. "out" follows the outgoing, "in" the incoming edges, "all" and "total" both of them. This argument is ignored for undirected graphs.
roots	A vector giving the vertices from which the breadth-first search is performed. Typically it contains one vertex per component.

Details

A forest is a graph, whose components are trees.

The roots vector can be calculated by simply doing a topological sort in all components of the graph, see the examples below.

Value

A list with two components:

tree The result, an igraph object, a tree or a forest.

vertex_index A numeric vector, it gives a mapping from the vertices of the new graph to the vertices of the old graph.

Related documentation in the C library

unfold_tree(), vcount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Other structural.properties: bfs(), component_distribution(), connect(), constraint(), coreness(), degree(), dfs(), distance_table(), edge_density(), feedback_arc_set(), feedback_vertex_set(), girth(), is_acyclic(), is_dag(), is_matching(), k_shortest_paths(), knn(), reciprocity(), subcomponent(), subgraph(), topo_sort(), transitivity(), which_multiple(), which_mutual()

Examples

```
g <- make_tree(10) %du% make_tree(10)
V(g)$id <- seq_len(vcount(g)) - 1
roots <- sapply(decompose(g), function(x) {
  V(x)$id[topo_sort(x)[1] + 1]
})
tree <- unfold_tree(g, roots = roots)
```

---

union *Union of two or more sets*

Description

This is an S3 generic function. See methods("union") for the actual implementations for various S3 classes. Initially it is implemented for igraph graphs and igraph vertex and edge sequences. See union.igraph(), and union.igraph.vs().

Usage

```
union(...)
```

Arguments

...	Arguments, their number and interpretation depends on the function that implements union().

Value

Depends on the function that implements this method.

See Also

Other functions for manipulating graph structure: `+.igraph()`, `add_edges()`, `add_vertices()`, `complementer()`, `compose()`, `connect()`, `contract()`, `delete_edges()`, `delete_vertices()`, `difference()`, `difference.igraph()`, `disjoint_union()`, `edge()`, `igraph-minus`, `intersection()`, `intersection.igraph()`, `path()`, `permute()`, `rep.igraph()`, `reverse_edges()`, `simplify()`, `transitive_closure()`, `union.igraph()`, `vertex()`

---

union.igraph *Union of graphs*

Description

The union of two or more graphs are created. The graphs may have identical or overlapping vertex sets.

Usage

```
## S3 method for class 'igraph'
union(..., byname = "auto")
```

Arguments

`...`	Graph objects or lists of graph objects.
`byname`	A logical scalar, or the character scalar `auto`. Whether to perform the operation based on symbolic vertex names. If it is `auto`, that means `TRUE` if all graphs are named and `FALSE` otherwise. A warning is generated if `auto` and some (but not all) graphs are named.

Details

`union()` creates the union of two or more graphs. Edges which are included in at least one graph will be part of the new graph. This function can be also used via the `%u%` operator.

If the `byname` argument is `TRUE` (or `auto` and all graphs are named), then the operation is performed on symbolic vertex names instead of the internal numeric vertex ids.

`union()` keeps the attributes of all graphs. All graph, vertex and edge attributes are copied to the result. If an attribute is present in multiple graphs and would result a name clash, then this attribute is renamed by adding suffixes: _1, _2, etc.

The `name` vertex attribute is treated specially if the operation is performed based on symbolic vertex names. In this case `name` must be present in all graphs, and it is not renamed in the result graph.

An error is generated if some input graphs are directed and others are undirected.

Value

A new graph object.

Related documentation in the C library

vcount(), permute_vertices(), edges(), get_eids(), ecount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

Other functions for manipulating graph structure: +.igraph(), add_edges(), add_vertices(), complementer(), compose(), connect(), contract(), delete_edges(), delete_vertices(), difference(), difference.igraph(), disjoint_union(), edge(), igraph-minus, intersection(), intersection.igraph(), path(), permute(), rep.igraph(), reverse_edges(), simplify(), transitive_closure(), union(), vertex()

Examples

```
## Union of two social networks with overlapping sets of actors
net1 <- graph_from_literal(
  D - A:B:F:G, A - C - F - A, B - E - G - B, A - B, F - G,
  H - F:G, H - I - J
)
net2 <- graph_from_literal(D - A:F:Y, B - A - X - F - H - Z, F - Y)
print_all(net1 %u% net2)
```

---

union.igraph.es *Union of edge sequences*

Description

Union of edge sequences

Usage

```
## S3 method for class 'igraph.es'
union(...)
```

Arguments

...	The edge sequences to take the union of.

Details

They must belong to the same graph. Note that this function has 'set' semantics and the multiplicity of edges is lost in the result. (This is to match the behavior of the based unique function.)

Value

An edge sequence that contains all edges in the given sequences, exactly once.

See Also

Other vertex and edge sequence operations: c.igraph.es(), c.igraph.vs(), difference.igraph.es(), difference.igraph.vs(), igraph-es-indexing, igraph-es-indexing2, igraph-vs-indexing, igraph-vs-indexing2, intersection.igraph.es(), intersection.igraph.vs(), rev.igraph.es(), rev.igraph.vs(), union.igraph.vs(), unique.igraph.es(), unique.igraph.vs()

Examples

```
g <- make_(ring(10), with_vertex_(name = LETTERS[1:10]))
union(E(g)[1:6], E(g)[5:9], E(g)["A|J"])
```

---

union.igraph.vs *Union of vertex sequences*

Description

Union of vertex sequences

Usage

```
## S3 method for class 'igraph.vs'
union(...)
```

Arguments

...	The vertex sequences to take the union of.

Details

They must belong to the same graph. Note that this function has 'set' semantics and the multiplicity of vertices is lost in the result. (This is to match the behavior of the based unique function.)

Value

A vertex sequence that contains all vertices in the given sequences, exactly once.

See Also

Other vertex and edge sequence operations: c.igraph.es(), c.igraph.vs(), difference.igraph.es(), difference.igraph.vs(), igraph-es-indexing, igraph-es-indexing2, igraph-vs-indexing, igraph-vs-indexing2, intersection.igraph.es(), intersection.igraph.vs(), rev.igraph.es(), rev.igraph.vs(), union.igraph.es(), unique.igraph.es(), unique.igraph.vs()

Examples

```
g <- make_(ring(10), with_vertex_(name = LETTERS[1:10]))
union(V(g)[1:6], V(g)[5:10])
```

`unique.igraph.es` *Remove duplicate edges from an edge sequence*

Description

Remove duplicate edges from an edge sequence

Usage

```
## S3 method for class 'igraph.es'
unique(x, incomparables = FALSE, ...)
```

Arguments

`x`	An edge sequence.
`incomparables`	a vector of values that cannot be compared. Passed to base function `duplicated`. See details there.
`...`	Passed to base function `duplicated()`.

Value

An edge sequence with the duplicate vertices removed.

See Also

Other vertex and edge sequence operations: `c.igraph.es()`, `c.igraph.vs()`, `difference.igraph.es()`, `difference.igraph.vs()`, `igraph-es-indexing`, `igraph-es-indexing2`, `igraph-vs-indexing`, `igraph-vs-indexing2`, `intersection.igraph.es()`, `intersection.igraph.vs()`, `rev.igraph.es()`, `rev.igraph.vs()`, `union.igraph.es()`, `union.igraph.vs()`, `unique.igraph.vs()`

Examples

```
g <- make_(ring(10), with_vertex_(name = LETTERS[1:10]))
E(g)[1, 1:5, 1:10, 5:10]
E(g)[1, 1:5, 1:10, 5:10] %>% unique()
```

unique.igraph.vs *Remove duplicate vertices from a vertex sequence*

Description

Remove duplicate vertices from a vertex sequence

Usage

```
## S3 method for class 'igraph.vs'
unique(x, incomparables = FALSE, ...)
```

Arguments

x	A vertex sequence.
incomparables	a vector of values that cannot be compared. Passed to base function duplicated. See details there.
...	Passed to base function duplicated().

Value

A vertex sequence with the duplicate vertices removed.

See Also

Other vertex and edge sequence operations: c.igraph.es(), c.igraph.vs(), difference.igraph.es(), difference.igraph.vs(), igraph-es-indexing, igraph-es-indexing2, igraph-vs-indexing, igraph-vs-indexing2, intersection.igraph.es(), intersection.igraph.vs(), rev.igraph.es(), rev.igraph.vs(), union.igraph.es(), union.igraph.vs(), unique.igraph.es()

Examples

```
g <- make_(ring(10), with_vertex_(name = LETTERS[1:10]))
V(g)[1, 1:5, 1:10, 5:10]
V(g)[1, 1:5, 1:10, 5:10] %>% unique()
```

upgrade_graph *igraph data structure versions*

Description

igraph's internal data representation changes sometimes between versions. This means that it is not possible to use igraph objects that were created (and possibly saved to a file) with an older igraph version.

Usage

```
upgrade_graph(graph)
```

Arguments

graph	The input graph.

Details

graph_version() queries the current data format, or the data format of a possibly older igraph graph.

upgrade_graph() can convert an older data format to the current one.

Value

The graph in the current format.

Related documentation in the C library

vcount()

See Also

graph_version to check the current data format version or the version of a graph.

Other versions: graph_version()

V *Vertices of a graph*

Description

Create a vertex sequence (vs) containing all vertices of a graph.

Usage

```
V(graph)
```

Arguments

graph	The graph

Details

A vertex sequence is just what the name says it is: a sequence of vertices. Vertex sequences are usually used as igraph function arguments that refer to vertices of a graph.

A vertex sequence is tied to the graph it refers to: it really denoted the specific vertices of that graph, and cannot be used together with another graph.

At the implementation level, a vertex sequence is simply a vector containing numeric vertex ids, but it has a special class attribute which makes it possible to perform graph specific operations on it, like selecting a subset of the vertices based on graph structure, or vertex attributes.

A vertex sequence is most often created by the `V()` function. The result of this includes all vertices in increasing vertex id order. A vertex sequence can be indexed by a numeric vector, just like a regular R vector. See `[.igraph.vs` and additional links to other vertex sequence operations below.

Value

A vertex sequence containing all vertices, in the order of their numeric vertex ids.

Indexing vertex sequences

Vertex sequences mostly behave like regular vectors, but there are some additional indexing operations that are specific for them; e.g. selecting vertices based on graph structure, or based on vertex attributes. See `[.igraph.vs` for details.

Querying or setting attributes

Vertex sequences can be used to query or set attributes for the vertices in the sequence. See `$.igraph.vs()` for details.

Related documentation in the C library

`vcount()`

See Also

Other vertex and edge sequences: `E()`, `as_ids()`, `igraph-es-attributes`, `igraph-es-indexing`, `igraph-es-indexing2`, `igraph-vs-attributes`, `igraph-vs-indexing`, `igraph-vs-indexing2`, `print.igraph.es()`, `print.igraph.vs()`

Examples

```
# Vertex ids of an unnamed graph
g <- make_ring(10)
V(g)

# Vertex ids of a named graph
g2 <- make_ring(10) %>%
  set_vertex_attr("name", value = letters[1:10])
V(g2)
```

vertex *Helper function for adding and deleting vertices*

Description

This is a helper function that simplifies adding and deleting vertices to/from graphs.

Usage

```
vertex(...)

vertices(...)
```

Arguments

...	See details below.

Details

`vertices()` is an alias for `vertex()`.

When adding vertices via `+`, all unnamed arguments are interpreted as vertex names of the new vertices. Named arguments are interpreted as vertex attributes for the new vertices.

When deleting vertices via `-`, all arguments of `vertex()` (or `vertices()`) are concatenated via `c()` and passed to `delete_vertices()`.

Value

A special object that can be used with together with igraph graphs and the plus and minus operators.

See Also

Other functions for manipulating graph structure: `+.igraph()`, `add_edges()`, `add_vertices()`, `complementer()`, `compose()`, `connect()`, `contract()`, `delete_edges()`, `delete_vertices()`, `difference()`, `difference.igraph()`, `disjoint_union()`, `edge()`, `igraph-minus`, `intersection()`, `intersection.igraph()`, `path()`, `permute()`, `rep.igraph()`, `reverse_edges()`, `simplify()`, `transitive_closure()`, `union()`, `union.igraph()`

Examples

```
g <- make_(ring(10), with_vertex_(name = LETTERS[1:10])) +
  vertices("X", "Y")
g
plot(g)
```

vertex.shape.pie *Using pie charts as vertices in graph plots*

Description

More complex vertex images can be used to express addtional information about vertices. E.g. pie charts can be used as vertices, to denote vertex classes, fuzzy classification of vertices, etc.

Details

The vertex shape 'pie' makes igraph draw a pie chart for every vertex. There are some extra graphical vertex parameters that specify how the pie charts will look like:

pie Numeric vector, gives the sizes of the pie slices.

pie.color A list of color vectors to use for the pies. If it is a list of a single vector, then this is used for all pies. It the color vector is shorter than the number of areas in a pie, then it is recycled.

pie.angle The slope of shading lines, given as an angle in degrees (counter-clockwise).

pie.density The density of the shading lines, in lines per inch. Non-positive values inhibit the drawing of shading lines.

pie.lty The line type of the border of the slices.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

igraph.plotting(), plot.igraph()

Examples

```
g <- make_ring(10)
values <- lapply(1:10, function(x) sample(1:10, 3))


plot(g,
  vertex.shape = "pie", vertex.pie = values,
  vertex.pie.color = list(heat.colors(5)),
  vertex.size = seq(10, 30, length.out = 10), vertex.label = NA
)
```

vertex_attr *Query vertex attributes of a graph*

Description

Query vertex attributes of a graph

Usage

```
vertex_attr(graph, name, index = V(graph))
```

Arguments

graph	The graph.
name	Name of the attribute to query. If missing, then all vertex attributes are returned in a list.
index	An optional vertex sequence to query the attribute only for these vertices.

Value

The value of the vertex attribute, or the list of all vertex attributes, if name is missing.

Related documentation in the C library

vcount()

See Also

Vertex, edge and graph attributes: delete_edge_attr(), delete_graph_attr(), delete_vertex_attr(), edge_attr(), edge_attr<-(), edge_attr_names(), graph_attr(), graph_attr<-(), graph_attr_names(), igraph-attribute-combination, igraph-dollar, igraph-vs-attributes, set_edge_attr(), set_graph_attr(), set_vertex_attr(), set_vertex_attrs(), vertex_attr<-(), vertex_attr_names()

Examples

```
g <- make_ring(10) %>%
  set_vertex_attr("color", value = "red") %>%
  set_vertex_attr("label", value = letters[1:10])
vertex_attr(g, "label")
vertex_attr(g)
plot(g)
```

vertex_attr_names *List names of vertex attributes*

Description

List names of vertex attributes

Usage

```
vertex_attr_names(graph)
```

Arguments

graph	The graph.

Value

Character vector, the names of the vertex attributes.

See Also

Vertex, edge and graph attributes: delete_edge_attr(), delete_graph_attr(), delete_vertex_attr(), edge_attr(), edge_attr<-(), edge_attr_names(), graph_attr(), graph_attr<-(), graph_attr_names(), igraph-attribute-combination, igraph-dollar, igraph-vs-attributes, set_edge_attr(), set_graph_attr(), set_vertex_attr(), set_vertex_attrs(), vertex_attr(), vertex_attr<-()

Examples

```
g <- make_ring(10) %>%
  set_vertex_attr("name", value = LETTERS[1:10]) %>%
  set_vertex_attr("color", value = rep("green", 10))
vertex_attr_names(g)
plot(g)
```

vertex_attr<- *Set one or more vertex attributes*

Description

Set one or more vertex attributes

Usage

```
vertex_attr(graph, name, index = V(graph)) <- value
```

Arguments

graph	The graph.
name	The name of the vertex attribute to set. If missing, then value must be a named list, and its entries are set as vertex attributes.
index	An optional vertex sequence to set the attributes of a subset of vertices.
value	The new value of the attribute(s) for all (or index) vertices.

Value

The graph, with the vertex attribute(s) added or set.

See Also

Vertex, edge and graph attributes: delete_edge_attr(), delete_graph_attr(), delete_vertex_attr(), edge_attr(), edge_attr<-(), edge_attr_names(), graph_attr(), graph_attr<-(), graph_attr_names(), igraph-attribute-combination, igraph-dollar, igraph-vs-attributes, set_edge_attr(), set_graph_attr(), set_vertex_attr(), set_vertex_attrs(), vertex_attr(), vertex_attr_names()

Examples

```
g <- make_ring(10)
vertex_attr(g) <- list(
  name = LETTERS[1:10],
  color = rep("yellow", gorder(g))
)
vertex_attr(g, "label") <- V(g)$name
g
plot(g)
```

---

vertex_connectivity *Vertex connectivity*

Description

The vertex connectivity of a graph or two vertices, this is recently also called group cohesion.

Usage

```
vertex_connectivity(graph, source = NULL, target = NULL, checks = TRUE)

vertex_disjoint_paths(graph, source = NULL, target = NULL)

## S3 method for class 'igraph'
cohesion(x, checks = TRUE, ...)
```

Arguments

graph, x	The input graph.
source	The id of the source vertex, for vertex_connectivity() it can be NULL, see details below.
target	The id of the target vertex, for vertex_connectivity() it can be NULL, see details below.
checks	Logical constant. Whether to check that the graph is connected and also the degree of the vertices. If the graph is not (strongly) connected then the connectivity is obviously zero. Otherwise if the minimum degree is one then the vertex connectivity is also one. It is a good idea to perform these checks, as they can be done quickly compared to the connectivity calculation itself. They were suggested by Peter McMahan, thanks Peter.
...	Additional arguments passed to methods. Not used by vertex_connectivity() directly but may be used by other methods that implement cohesion().

Details

The vertex connectivity of two vertices (source and target) in a graph is the minimum number of vertices that must be deleted to eliminate all (directed) paths from source to target. vertex_connectivity() calculates this quantity if both the source and target arguments are given and they're not NULL.

The vertex connectivity of a pair is the same as the number of different (i.e. node-independent) paths from source to target, assuming no direct edges between them.

The vertex connectivity of a graph is the minimum vertex connectivity of all (ordered) pairs of vertices in the graph. In other words this is the minimum number of vertices needed to remove to make the graph not strongly connected. (If the graph is not strongly connected then this is zero.) vertex_connectivity() calculates this quantity if neither the source nor target arguments are given. (I.e. they are both NULL.)

A set of vertex disjoint directed paths from source to vertex is a set of directed paths between them whose vertices do not contain common vertices (apart from source and target). The maximum number of vertex disjoint paths between two vertices is the same as their vertex connectivity in most cases (if the two vertices are not connected by an edge).

The cohesion of a graph (as defined by White and Harary, see references), is the vertex connectivity of the graph. This is calculated by cohesion().

These three functions essentially calculate the same measure(s), more precisely vertex_connectivity() is the most general, the other two are included only for the ease of using more descriptive function names.

Value

A scalar real value.

Related documentation in the C library

st_vertex_connectivity(), vertex_connectivity(), vcount(), vertex_disjoint_paths(), cohesion()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

White, Douglas R and Frank Harary 2001. The Cohesiveness of Blocks In Social Networks: Node Connectivity and Conditional Density. *Sociological Methodology* 31 (1) : 305-359.

See Also

Other flow: dominator_tree(), edge_connectivity(), is_min_separator(), is_separator(), max_flow(), min_cut(), min_separators(), min_st_separators(), st_cuts(), st_min_cuts()

Examples

```
g <- sample_pa(100, m = 1)
g <- delete_edges(g, E(g)[100 %--% 1])
g2 <- sample_pa(100, m = 5)
g2 <- delete_edges(g2, E(g2)[100 %--% 1])
vertex_connectivity(g, 100, 1)
vertex_connectivity(g2, 100, 1)
vertex_disjoint_paths(g2, 100, 1)

g <- sample_gnp(50, 5 / 50)
g <- as_directed(g)
g <- induced_subgraph(g, subcomponent(g, 1))
cohesion(g)
```

---

voronoi_cells *Voronoi partitioning of a graph*

Description

[Experimental]

This function partitions the vertices of a graph based on a set of generator vertices. Each vertex is assigned to the generator vertex from (or to) which it is closest.

groups() may be used on the output of this function.

Usage

```
voronoi_cells(
  graph,
  generators,
  ...,
  weights = NULL,
  mode = c("out", "in", "all", "total"),
  tiebreaker = c("random", "first", "last")
)
```

Arguments

graph	The graph to partition into Voronoi cells.
generators	The generator vertices of the Voronoi cells.
...	These dots are for future extensions and must be empty.
weights	Possibly a numeric vector giving edge weights. If this is NULL and the graph has a weight edge attribute, then the attribute is used. If this is NA then no weights are used (even if the graph has a weight attribute). In a weighted graph, the length of a path is the sum of the weights of its constituent edges.
mode	Character string. In directed graphs, whether to compute distances from generator vertices to other vertices ("out"), to generator vertices from other vertices ("in"), or ignore edge directions entirely ("all"). Ignored in undirected graphs.
tiebreaker	Character string that specifies what to do when a vertex is at the same distance from multiple generators. "random" assigns a minimal-distance generator randomly, "first" takes the first one, and "last" takes the last one.

Value

A named list with two components:

membership numeric vector giving the cluster id to which each vertex belongs.

distances numeric vector giving the distance of each vertex from its generator

Related documentation in the C library

voronoi(), edges(), vcount(), get_eids(), ecount()

See Also

distances()

Community detection: as_membership(), cluster_edge_betweenness(), cluster_fast_greedy(), cluster_fluid_communities(), cluster_infomap(), cluster_label_prop(), cluster_leading_eigen(), cluster_leiden(), cluster_louvain(), cluster_optimal(), cluster_spinglass(), cluster_walktrap(), compare(), groups(), make_clusters(), membership(), modularity.igraph(), plot_dendrogram(), split_join_distance()

Examples

```
g <- make_lattice(c(10, 10))
clu <- voronoi_cells(g, c(25, 43, 67))
groups(clu)
plot(g, vertex.color = clu$membership)
```

weighted_cliques *Functions to find weighted cliques, i.e. vertex-weighted complete subgraphs in a graph*

Description

These functions find all, the largest or all the maximal weighted cliques in an undirected graph. The weight of a clique is the sum of the weights of its vertices.

Usage

```
weighted_cliques(
  graph,
  vertex.weights = NULL,
  min.weight = 0,
  max.weight = 0,
  maximal = FALSE
)
```

Arguments

graph	The input graph, directed graphs will be considered as undirected ones, multiple edges and loops are ignored.
vertex.weights	Vertex weight vector. If the graph has a weight vertex attribute, then this is used by default. If the graph does not have a weight vertex attribute and this argument is NULL, then every vertex is assumed to have a weight of 1. Note that the current implementation of the weighted clique finder supports positive integer weights only.
min.weight	Numeric constant, lower limit on the weight of the cliques to find. NULL means no limit, i.e. it is the same as 0.
max.weight	Numeric constant, upper limit on the weight of the cliques to find. NULL means no limit.
maximal	Specifies whether to look for all weighted cliques (FALSE) or only the maximal ones (TRUE).

Details

weighted_cliques() finds all complete subgraphs in the input graph, obeying the weight limitations given in the min and max arguments.

largest_weighted_cliques() finds all largest weighted cliques in the input graph. A clique is largest if there is no other clique whose total weight is larger than the weight of this clique.

weighted_clique_num() calculates the weight of the largest weighted clique(s).

Value

`weighted_cliques()` and `largest_weighted_cliques()` return a list containing numeric vectors of vertex IDs. Each list element is a weighted clique, i.e. a vertex sequence of class igraph.vs.

`weighted_clique_num()` returns an integer scalar.

Related documentation in the C library

`weighted_cliques()`, `vcount()`

Author(s)

Tamas Nepusz <ntamas@gmail.com> and Gabor Csardi <csardi.gabor@gmail.com>

See Also

Other cliques: `cliques()`, `is_complete()`, `ivs()`

Examples

```
g <- make_graph("zachary")
V(g)$weight <- 1
V(g)[c(1, 2, 3, 4, 14)]$weight <- 3
weighted_cliques(g)
weighted_cliques(g, maximal = TRUE)
largest_weighted_cliques(g)
weighted_clique_num(g)
```

---

which_multiple *Find the multiple or loop edges in a graph*

Description

A loop edge is an edge from a vertex to itself. An edge is a multiple edge if it has exactly the same head and tail vertices as another edge. A graph without multiple and loop edges is called a simple graph.

Usage

```
which_multiple(graph, eids = E(graph))

any_multiple(graph)

count_multiple(graph, eids = E(graph))

which_loop(graph, eids = E(graph))

any_loop(graph)

count_loops(graph)
```

Arguments

graph	The input graph.
eids	The edges to which the query is restricted. By default this is all edges in the graph.

Details

any_loop() decides whether the graph has any loop edges.

which_loop() decides whether the edges of the graph are loop edges.

count_loops() counts the total number of loop edges in the graph.

any_multiple() decides whether the graph has any multiple edges.

which_multiple() decides whether the edges of the graph are multiple edges.

count_multiple() counts the multiplicity of each edge of a graph.

Note that the semantics for which_multiple() and count_multiple() is different. which_multiple() gives TRUE for all occurrences of a multiple edge except for one. I.e. if there are three i-j edges in the graph then which_multiple() returns TRUE for only two of them while count_multiple() returns '3' for all three.

See the examples for getting rid of multiple edges while keeping their original multiplicity as an edge attribute.

Value

any_loop() and any_multiple() return a logical scalar. which_loop() and which_multiple() return a logical vector. count_loops() returns a numeric scalar with the total number of loop edges. count_multiple() returns a numeric vector.

Related documentation in the C library

is_multiple(), edges(), get_eids(), vcount(), ecount(), has_multiple(), count_multiple(), is_loop(), has_loop(), count_loops()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

simplify() to eliminate loop and multiple edges.

Other structural.properties: bfs(), component_distribution(), connect(), constraint(), coreness(), degree(), dfs(), distance_table(), edge_density(), feedback_arc_set(), feedback_vertex_set(), girth(), is_acyclic(), is_dag(), is_matching(), k_shortest_paths(), knn(), reciprocity(), subcomponent(), subgraph(), topo_sort(), transitivity(), unfold_tree(), which_mutual()

Examples

```
# Loops
g <- make_graph(c(1, 1, 2, 2, 3, 3, 4, 5))
any_loop(g)
which_loop(g)
count_loops(g)

# Multiple edges
g <- sample_pa(10, m = 3, algorithm = "bag")
any_multiple(g)
which_multiple(g)
count_multiple(g)
which_multiple(simplify(g))
all(count_multiple(simplify(g)) == 1)

# Direction of the edge is important
which_multiple(make_graph(c(1, 2, 2, 1)))
which_multiple(make_graph(c(1, 2, 2, 1), dir = FALSE))

# Remove multiple edges but keep multiplicity
g <- sample_pa(10, m = 3, algorithm = "bag")
E(g)$weight <- count_multiple(g)
g <- simplify(g, edge.attr.comb = list(weight = "min"))
any(which_multiple(g))
E(g)$weight
```

---

which_mutual *Find mutual edges in a directed graph*

Description

This function checks the reciprocal pair of the supplied edges.

Usage

```
which_mutual(graph, eids = E(graph), loops = TRUE)
```

Arguments

graph	The input graph.
eids	Edge sequence, the edges that will be probed. By default is includes all edges in the order of their ids.
loops	Logical, whether to consider directed self-loops to be mutual.

Details

In a directed graph an (A,B) edge is mutual if the graph also includes a (B,A) directed edge.

Note that multi-graphs are not handled properly, i.e. if the graph contains two copies of (A,B) and one copy of (B,A), then these three edges are considered to be mutual.

Undirected graphs contain only mutual edges by definition.

Value

A logical vector of the same length as the number of edges supplied.

Related documentation in the C library

`is_mutual()`, `edges()`, `get_eids()`, `vcount()`, `ecount()`

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

`reciprocity()`, `dyad_census()` if you just want some statistics about mutual edges.

Other structural.properties: `bfs()`, `component_distribution()`, `connect()`, `constraint()`, `coreness()`, `degree()`, `dfs()`, `distance_table()`, `edge_density()`, `feedback_arc_set()`, `feedback_vertex_set()`, `girth()`, `is_acyclic()`, `is_dag()`, `is_matching()`, `k_shortest_paths()`, `knn()`, `reciprocity()`, `subcomponent()`, `subgraph()`, `topo_sort()`, `transitivity()`, `unfold_tree()`, `which_multiple()`

Examples

```
g <- sample_gnm(10, 50, directed = TRUE)
reciprocity(g)
dyad_census(g)
which_mutual(g)
sum(which_mutual(g)) / 2 == dyad_census(g)$mut
```

---

with_edge_ *Constructor modifier to add edge attributes*

Description

Constructor modifier to add edge attributes

Usage

```
with_edge_(...)
```

Arguments

...	The attributes to add. They must be named.

See Also

Constructor modifiers (and related functions): make_(), sample_(), simplified(), with_graph_(), with_vertex_(), without_attr(), without_loops(), without_multiples()

Examples

```
make_(
  ring(10),
  with_edge_(
    color = "red",
    weight = rep(1:2, 5)
  )
) %>%
  plot()
```

---

with_graph_ *Constructor modifier to add graph attributes*

Description

Constructor modifier to add graph attributes

Usage

```
with_graph_(...)
```

Arguments

...	The attributes to add. They must be named.

See Also

Constructor modifiers (and related functions): make_(), sample_(), simplified(), with_edge_(), with_vertex_(), without_attr(), without_loops(), without_multiples()

Examples

```
make_(ring(10), with_graph_(name = "10-ring"))
```

with_igraph_opt *Run code with a temporary igraph options setting*

Description

Run code with a temporary igraph options setting

Usage

```
with_igraph_opt(options, code)
```

Arguments

options	A named list of the options to change.
code	The code to run.

Value

The result of the code.

See Also

Other igraph options: igraph_options()

Examples

```
with_igraph_opt(
  list(sparsematrices = FALSE),
  make_ring(10)[]
)
igraph_opt("sparsematrices")
```

with_vertex_ *Constructor modifier to add vertex attributes*

Description

Constructor modifier to add vertex attributes

Usage

```
with_vertex_(...)
```

Arguments

...	The attributes to add. They must be named.

See Also

Constructor modifiers (and related functions): make_(), sample_(), simplified(), with_edge_(), with_graph_(), without_attr(), without_loops(), without_multiples()

Examples

```
make_(
  ring(10),
  with_vertex_(
    color = "#7fcdbb",
    frame.color = "#7fcdbb",
    name = LETTERS[1:10]
  )
) %>%
  plot()
```

---

without_attr *Construtor modifier to remove all attributes from a graph*

Description

Construtor modifier to remove all attributes from a graph

Usage

```
without_attr()
```

See Also

Constructor modifiers (and related functions): make_(), sample_(), simplified(), with_edge_(), with_graph_(), with_vertex_(), without_loops(), without_multiples()

Examples

```
g1 <- make_ring(10)
g1

g2 <- make_(ring(10), without_attr())
g2
```

without_loops *Constructor modifier to drop loop edges*

Description

Constructor modifier to drop loop edges

Usage

```
without_loops()
```

See Also

Constructor modifiers (and related functions): make_(), sample_(), simplified(), with_edge_(), with_graph_(), with_vertex_(), without_attr(), without_multiples()

Examples

```
# An artificial example
make_(full_graph(5, loops = TRUE))
make_(full_graph(5, loops = TRUE), without_loops())
```

without_multiples *Constructor modifier to drop multiple edges*

Description

Constructor modifier to drop multiple edges

Usage

```
without_multiples()
```

See Also

Constructor modifiers (and related functions): make_(), sample_(), simplified(), with_edge_(), with_graph_(), with_vertex_(), without_attr(), without_loops()

Examples

```
sample_(pa(10, m = 3, algorithm = "bag"))
sample_(pa(10, m = 3, algorithm = "bag"), without_multiples())
```

write_graph *Writing the graph to a file in some format*

Description

write_graph() is a general function for exporting graphs to foreign file formats. The recommended formats for data exchange are GraphML and GML.

Usage

```
write_graph(
  graph,
  file,
 format = c("edgelist", "pajek", "ncol", "lgl", "graphml", "dimacs", "gml", "dot",
    "leda"),
  ...
)
```

Arguments

graph	The graph to export.
file	A connection or a string giving the file name to write the graph to.
format	Character string giving the file format. Right now pajek, graphml, dot, gml, edgelist, lgl, ncol, leda and dimacs are implemented. As of igraph 0.4 this argument is case insensitive.
...	Other, format specific arguments, see below.

Value

A 'NULL", invisibly.

Edge list format

The edgelist format is a simple text file, with one edge per line, the two zero-based numerical vertex IDs separated by a space character. Note that vertices are indexed starting with zero. The file is sorted by the first and the second column. This format has no additional arguments.

NCOL format

This format is a plain text edge list in which vertices are referred to by name rather than numerical ID. Edge weights may be optionally written. Additional parameters:

names The name of a vertex attribute to take vertex names from or NULL to use zero-based numerical IDs.

weights The name of an edge attribute to take edge weights from or NULL to omit edge weights.

Pajek format

The `pajek` format is provided for interoperability with the Pajek software only. Since the format does not have a formal specification, it is not recommended for general data exchange or archival.

LGL format

The .lgl format is used by the Large Graph Layout visualization software (https://lgl.sourceforge.net), it can describe undirected optionally weighted graphs.

names The name of a vertex attribute to use for vertex names, or NULL to use numeric IDs.

weights The name of an edge attribute to use for edge weights, or NULL to omit weights.

isolates Logical, whether to include isolated vertices in the file. Default is FALSE.

DIMACS format

This is a line-oriented text file (ASCII) format. The first character of each line defines the type of the line. If the first character is c the line is a comment line and it is ignored. There is one problem line (p in the file), it must appear before any node and arc descriptor lines. The problem line has three fields separated by spaces: the problem type (max or edge), the number of vertices, and number of edges in the graph. In MAX problems, exactly two node identification lines are expected (n), one for the source, and one for the target vertex. These have two fields: the ID of the vertex and the type of the vertex, either s (= source) or t (= target). Arc lines start with a and have three fields: the source vertex, the target vertex and the edge capacity. In EDGE problems, there may be a node line (n) for each node. It specifies the node index and an integer node label. Nodes for which no explicit label was specified will use their index as label. In EDGE problems, each edge is specified as an edge line (e).

source Numeric ID of the source vertex.

target Numeric ID of the target vertex.

capacity The name of an edge attribute to use for edge capacities, or NULL to use the "capacity" attribute if it exists.

GML format

GML is a quite general textual format.

id Optional numeric vertex IDs to use.

creator Optional string specifying the creator of the file.

GraphML format

GraphML is an XML-based file format for representing various types of graphs. When a numerical attribute value is NaN, it will be omitted from the file. This function assumes that non-ASCII characters in attribute names and string attribute values are UTF-8 encoded. If this is not the case, the resulting XML file will be invalid. Control characters, i.e. character codes up to and including 31 (with the exception of tab, cr and lf), are not allowed.

prefixAttr Logical, whether to prefix attribute names to ensure uniqueness across vertex/edge/graph attributes. Default is TRUE.

LEDA format

This function writes a graph to an output stream in LEDA format. See https://www.algorithmic-solutions.info/leda_guide/graphs/leda_native_graph_fileformat.html. The support for the LEDA format is very basic at the moment; igraph writes only the LEDA graph section which supports one selected vertex and edge attribute and no layout information or visual attributes.

vertex.attr Name of vertex attribute to include in the file.

edge.attr Name of edge attribute to include in the file.

DOT format

DOT is the format used by the widely known GraphViz software, see https://www.graphviz.org for details. The grammar of the DOT format can be found here: https://www.graphviz.org/doc/info/lang.html. This is only a preliminary implementation, no visualization information is written. This format is meant solely for interoperability with Graphviz. It is not recommended for data exchange or archival.

Related documentation in the C library

write_graph_edgelist(), write_graph_pajek(), write_graph_graphml(), write_graph_gml(), write_graph_dot(), write_graph_leda(), edges(), get_eids(), vcount(), ecount()

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Adai AT, Date SV, Wieland S, Marcotte EM. LGL: creating a map of protein function with an algorithm for visualizing very large biological networks. *J Mol Biol.* 2004 Jun 25;340(1):179-90.

See Also

read_graph()

Foreign format readers: graph_from_graphdb(), read_graph()

Examples

```
g <- make_ring(10)
file <- tempfile(fileext = ".txt")
write_graph(g, file, "edgelist")
if (!interactive()) {
  unlink(file)
}
```

Index

∗ **Edge list**
graph_from_edgelist, 209
∗ **Empty graph.**
make_empty_graph, 330
∗ **Full graph**
make_full_graph, 334
∗ **Graph Atlas.**
graph_from_atlas, 207
∗ **Lattice**
make_lattice, 340
∗ **Star graph**
make_star, 344
∗ **Trees.**
make_tree, 345
∗ **Wheel graph**
make_wheel, 347
∗ **adjacency**
graph_from_adjacency_matrix, 202
∗ **arpack**
arpack_defaults, 27
∗ **attributes**
delete_edge_attr, 140
delete_graph_attr, 141
delete_vertex_attr, 142
edge_attr, 171
edge_attr<-, 172
edge_attr_names, 172
graph_attr, 197
graph_attr<-, 199
graph_attr_names, 198
igraph-attribute-combination, 232
igraph-dollar, 234
igraph-vs-attributes, 241
set_edge_attr, 472
set_graph_attr, 473
set_vertex_attr, 473
set_vertex_attrs, 474
vertex_attr, 525
vertex_attr<-, 526
vertex_attr_names, 526
∗ **biadjacency**
as_data_frame, 39
graph_from_biadjacency_matrix, 208
∗ **bipartite**
bipartite_mapping, 61
bipartite_projection, 62
is_bipartite, 258
make_bipartite_graph, 325
∗ **centrality**
alpha_centrality, 24
authority_score, 50
betweenness, 53
closeness, 82
diversity, 161
eigen_centrality, 177
harmonic_centrality, 224
hits_scores, 228
page_rank, 371
power_centrality, 390
spectrum, 483
strength, 490
subgraph_centrality, 494
∗ **centralization related**
centr_betw, 69
centr_betw_tmax, 70
centr_clo, 71
centr_clo_tmax, 72
centr_degree, 73
centr_degree_tmax, 74
centr_eigen, 75
centr_eigen_tmax, 76
centralize, 77
∗ **chordal**
is_chordal, 258
max_cardinality, 349
∗ **cliques**
cliques, 79
is_complete, 260

ivs, 279
weighted_cliques, 531
* **cocitation**
cocitation, 106
similarity, 478
* **cohesive.blocks**
cohesive_blocks, 107
* **coloring**
greedy_vertex_coloring, 221
* **community**
as_membership, 47
cluster_edge_betweenness, 84
cluster_fast_greedy, 86
cluster_fluid_communities, 88
cluster_infomap, 89
cluster_label_prop, 91
cluster_leading_eigen, 93
cluster_leiden, 95
cluster_louvain, 98
cluster_optimal, 100
cluster_spinglass, 101
cluster_walktrap, 104
compare, 112
groups, 222
make_clusters, 328
membership, 352
modularity.igraph, 363
plot_dendrogram, 386
split_join_distance, 485
voronoi_cells, 529
* **components**
articulation_points, 31
biconnected_components, 58
component_distribution, 115
count_reachable, 133
decompose, 137
is_biconnected, 257
* **console**
console, 123
* **constructor modifiers**
make_, 323
sample_, 418
simplified, 481
with_edge_, 535
with_graph_, 536
with_vertex_, 537
without_attr, 538
without_loops, 539
without_multiples, 539
* **conversion**
as.matrix.igraph, 33
as_adj_list, 34
as_adjacency_matrix, 36
as_biadjacency_matrix, 37
as_data_frame, 39
as_directed, 41
as_edgelist, 43
as_graphnel, 44
as_long_data_frame, 46
graph_from_adj_list, 201
graph_from_graphnel, 212
* **cycles**
feedback_arc_set, 184
feedback_vertex_set, 185
find_cycle, 186
girth, 192
has_eulerian_path, 225
is_acyclic, 256
is_dag, 261
simple_cycles, 480
* **datagen**
sample_seq, 461
* **deterministic constructors**
graph_from_atlas, 207
graph_from_edgelist, 209
graph_from_literal, 215
make_, 323
make_chordal_ring, 326
make_circulant, 327
make_empty_graph, 330
make_full_citation_graph, 333
make_full_graph, 334
make_full_multipartite, 335
make_graph, 336
make_lattice, 340
make_ring, 343
make_star, 344
make_tree, 345
make_turan, 346
make_wheel, 347
* **efficiency**
global_efficiency, 194
* **embedding**
dim_select, 152
embed_adjacency_matrix, 179
embed_laplacian_matrix, 181

∗ **env-and-data**
dot-data, 164
∗ **fit**
fit_power_law, 188
∗ **flow**
dominator_tree, 163
edge_connectivity, 173
is_min_separator, 269
is_separator, 272
max_flow, 351
min_cut, 358
min_separators, 360
min_st_separators, 361
st_cuts, 486
st_min_cuts, 488
vertex_connectivity, 527
∗ **foreign**
graph_from_graphdb, 210
read_graph, 405
write_graph, 540
∗ **functions for manipulating graph structure**
+.igraph, 15
add_edges, 18
add_vertices, 20
complementer, 114
compose, 118
connect, 119
contract, 125
delete_edges, 140
delete_vertices, 143
difference, 148
difference.igraph, 149
disjoint_union, 154
edge, 170
igraph-minus, 240
intersection, 251
intersection.igraph, 251
path, 373
permute, 374
rep.igraph, 412
reverse_edges, 415
simplify, 482
transitive_closure, 508
union, 515
union.igraph, 516
vertex, 523
∗ **games**
bipartite_gnm, 59
sample_, 418
sample_bipartite, 419
sample_chung_lu, 421
sample_correlated_gnp, 424
sample_correlated_gnp_pair, 426
sample_degseq, 427
sample_dot_product, 432
sample_fitness, 433
sample_fitness_pl, 435
sample_forestfire, 437
sample_gnm, 439
sample_gnp, 440
sample_grg, 441
sample_growing, 443
sample_hierarchical_sbm, 444
sample_islands, 446
sample_k_regular, 447
sample_last_cit, 448
sample_pa, 451
sample_pa_age, 454
sample_pref, 457
sample_sbm, 459
sample_smallworld, 462
sample_traits_callaway, 466
sample_tree, 468
∗ **glet**
graphlet_basis, 219
∗ **graph automorphism**
automorphism_group, 51
count_automorphisms, 129
∗ **graph isomorphism**
canonical_permutation, 66
count_isomorphisms, 131
count_subgraph_isomorphisms, 134
graph_from_isomorphism_class, 213
isomorphic, 275
isomorphism_class, 277
isomorphisms, 278
subgraph_isomorphic, 496
subgraph_isomorphisms, 498
∗ **graph layouts**
add_layout_, 19
component_wise, 117
layout_, 287
layout_as_bipartite, 289
layout_as_star, 290
layout_as_tree, 292

layout_in_circle, 294
layout_nicely, 296
layout_on_grid, 297
layout_on_sphere, 299
layout_randomly, 300
layout_with_dh, 301
layout_with_fr, 306
layout_with_gem, 308
layout_with_graphopt, 310
layout_with_kk, 311
layout_with_lgl, 314
layout_with_mds, 315
layout_with_sugiyama, 317
merge_coords, 356
norm_coords, 369
normalize, 370
∗ **graph motifs**
count_motifs, 132
dyad_census, 165
motifs, 365
sample_motifs, 450
∗ **graphical degree sequences**
is_degseq, 262
is_graphical, 265
∗ **graphs**
all_simple_paths, 22
alpha_centrality, 24
arpack_defaults, 27
articulation_points, 31
as.igraph, 32
as_adj_list, 34
as_biadjacency_matrix, 37
as_data_frame, 39
as_directed, 41
as_edgelist, 43
assortativity, 48
automorphism_group, 51
betweenness, 53
bfs, 55
biconnected_components, 58
bipartite_mapping, 61
bipartite_projection, 62
canonical_permutation, 66
cliques, 79
closeness, 82
cluster_edge_betweenness, 84
cluster_fast_greedy, 86
cluster_fluid_communities, 88
cluster_infomap, 89
cluster_label_prop, 91
cluster_leading_eigen, 93
cluster_leiden, 95
cluster_louvain, 98
cluster_optimal, 100
cluster_spinglass, 101
cluster_walktrap, 104
cocitation, 106
cohesive_blocks, 107
compare, 112
complementer, 114
component_distribution, 115
compose, 118
connect, 119
console, 123
constraint, 124
contract, 125
convex_hull, 127
coreness, 128
count_automorphisms, 129
count_reachable, 133
curve_multiple, 136
decompose, 137
degree, 138
dfs, 144
diameter, 147
difference.igraph, 149
dim_select, 152
disjoint_union, 154
distance_table, 155
diversity, 161
dominator_tree, 163
dyad_census, 165
each_edge, 167
edge_connectivity, 173
edge_density, 175
eigen_centrality, 177
embed_adjacency_matrix, 179
embed_laplacian_matrix, 181
feedback_arc_set, 184
feedback_vertex_set, 185
find_cycle, 186
fit_power_law, 188
girth, 192
global_efficiency, 194
graph_from_adj_list, 201
graph_from_adjacency_matrix, 202

graph_from_biadjacency_matrix, 208
graph_from_graphdb, 210
graph_from_lcf, 214
greedy_vertex_coloring, 221
harmonic_centrality, 224
has_eulerian_path, 225
igraph-attribute-combination, 232
igraph_options, 246
intersection.igraph, 251
is_acyclic, 256
is_biconnected, 257
is_chordal, 258
is_complete, 260
is_dag, 261
is_degseq, 262
is_forest, 264
is_graphical, 265
is_igraph, 266
is_named, 270
is_tree, 273
is_weighted, 274
ivs, 279
k_shortest_paths, 281
keeping_degseq, 282
knn, 283
laplacian_matrix, 285
layout_as_bipartite, 289
layout_as_star, 290
layout_as_tree, 292
layout_in_circle, 294
layout_nicely, 296
layout_on_grid, 297
layout_on_sphere, 299
layout_randomly, 300
layout_with_drl, 303
layout_with_fr, 306
layout_with_gem, 308
layout_with_graphopt, 310
layout_with_kk, 311
layout_with_lgl, 314
layout_with_mds, 315
layout_with_sugiyama, 317
make_bipartite_graph, 325
make_de_bruijn_graph, 329
make_from_prufer, 331
make_full_bipartite_graph, 332
make_kautz_graph, 339
make_line_graph, 342
match_vertices, 348
max_cardinality, 349
membership, 352
merge_coords, 356
min_st_separators, 361
modularity.igraph, 363
mst, 366
norm_coords, 369
page_rank, 371
permute, 374
plot.common, 375
plot.igraph, 382
plot.sir, 384
plot_dendrogram, 386
plot_dendrogram.igraphHRG, 388
power_centrality, 390
print.igraph, 394
read_graph, 405
realize_bipartite_degseq, 408
realize_degseq, 409
reciprocity, 411
rglplot, 416
sample_bipartite, 419
sample_correlated_gnp_pair, 426
sample_degseq, 427
sample_dot_product, 432
sample_fitness, 433
sample_fitness_pl, 435
sample_forestfire, 437
sample_gnm, 439
sample_gnp, 440
sample_grg, 441
sample_growing, 443
sample_hierarchical_sbm, 444
sample_islands, 446
sample_k_regular, 447
sample_last_cit, 448
sample_pa, 451
sample_pa_age, 454
sample_pref, 457
sample_sbm, 459
sample_smallworld, 462
sample_traits_callaway, 466
sample_tree, 468
similarity, 478
simple_cycles, 480
simplify, 482
spectrum, 483

st_cuts, 486
st_min_cuts, 488
stochastic_matrix, 489
strength, 490
subcomponent, 492
subgraph, 493
subgraph_centrality, 494
time_bins, 500
tkplot, 503
to_prufer, 506
topo_sort, 507
transitive_closure, 508
transitivity, 509
triad_census, 511
triangles, 513
unfold_tree, 514
union.igraph, 516
vertex.shape.pie, 524
vertex_connectivity, 527
weighted_cliques, 531
which_multiple, 532
which_mutual, 534
write_graph, 540
* **graph**
sample_spanning_tree, 463
* **hierarchical random graph functions**
consensus_tree, 122
fit_hrg, 187
hrg, 229
hrg-methods, 230
hrg_tree, 231
predict_edges, 392
print.igraphHRG, 399
print.igraphHRGConsensus, 400
sample_hrg, 445
* **igraph options**
igraph_options, 246
with_igraph_opt, 537
* **isomorphism**
simplify, 482
* **latent position vector samplers**
sample_dirichlet, 431
sample_sphere_surface, 464
sample_sphere_volume, 465
* **layout modifiers**
component_wise, 117
layout_modifier, 295
normalize, 370
* **layout_drl**
layout_with_drl, 303
* **low-level operations**
invalidate_cache, 255
* **manip**
running_mean, 417
* **minimum.spanning.tree**
mst, 366
* **motifs**
triad_census, 511
* **other**
convex_hull, 127
running_mean, 417
sample_seq, 461
* **palettes**
categorical_pal, 68
diverging_pal, 160
r_pal, 402
sequential_pal, 471
* **paths**
all_simple_paths, 22
diameter, 147
distance_table, 155
eccentricity, 168
graph_center, 200
radius, 402
* **plot.common**
curve_multiple, 136
* **plot.shapes**
shapes, 475
* **plot**
plot.igraph, 382
rglplot, 416
* **printer callbacks**
is_printer_callback, 271
printer_callback, 401
* **print**
print.igraph, 394
* **processes**
plot.sir, 384
time_bins, 500
* **random_walk**
random_walk, 403
* **rewiring functions**
each_edge, 167
keeping_degseq, 282
rewire, 416
* **scan statistics**

local_scan, 321
scan_stat, 469
* **sgm**
match_vertices, 348
* **similarity**
similarity, 478
* **simple**
simplify, 482
* **structural queries**
[.igraph, 10
[[.igraph, 13
adjacent_vertices, 21
are_adjacent, 26
ends, 183
get_edge_ids, 191
gorder, 196
gsize, 223
head_of, 226
incident, 248
incident_edges, 249
is_directed, 263
neighbors, 368
tail_of, 500
* **structural.properties**
bfs, 55
component_distribution, 115
connect, 119
constraint, 124
coreness, 128
degree, 138
dfs, 144
distance_table, 155
edge_density, 175
feedback_arc_set, 184
feedback_vertex_set, 185
girth, 192
is_acyclic, 256
is_dag, 261
is_matching, 267
k_shortest_paths, 281
knn, 283
reciprocity, 411
subcomponent, 492
subgraph, 493
topo_sort, 507
transitivity, 509
unfold_tree, 514
which_multiple, 532
which_mutual, 534
* **tkplot**
tkplot, 503
* **trees**
is_forest, 264
is_tree, 273
make_from_prufer, 331
sample_spanning_tree, 463
to_prufer, 506
* **triangles**
triangles, 513
* **versions**
graph_version, 218
upgrade_graph, 520
* **vertex and edge sequence operations**
c.igraph.es, 64
c.igraph.vs, 65
difference.igraph.es, 150
difference.igraph.vs, 151
igraph-es-indexing, 236
igraph-es-indexing2, 238
igraph-vs-indexing, 243
igraph-vs-indexing2, 245
intersection.igraph.es, 253
intersection.igraph.vs, 254
rev.igraph.es, 413
rev.igraph.vs, 414
union.igraph.es, 517
union.igraph.vs, 518
unique.igraph.es, 519
unique.igraph.vs, 520
* **vertex and edge sequences**
as_ids, 45
E, 166
igraph-es-attributes, 235
igraph-es-indexing, 236
igraph-es-indexing2, 238
igraph-vs-attributes, 241
igraph-vs-indexing, 243
igraph-vs-indexing2, 245
print.igraph.es, 396
print.igraph.vs, 397
V, 521
*.igraph *(*rep.igraph*)*, 412
+.igraph, 15
+.igraph(), *18*, *20*, *114*, *119*, *121*, *126*, *141*, *143*, *149*, *150*, *155*, *170*, *241*, *251*, *252*, *373*, *375*, *413*, *415*, *483*, *508*,

516, *517*, *523*
-.igraph *(*igraph-minus*)*, 240
.data *(*dot-data*)*, 164
.env *(*dot-data*)*, 164
[.igraph, 10
[.igraph(), *14*, *21*, *26*, *183*, *192*, *196*, *223*, *227*, *249*, *250*, *263*, *368*, *500*
[.igraph.es, *166*, *239*
[.igraph.es *(*igraph-es-indexing*)*, 236
[.igraph.vs, *246*, *522*
[.igraph.vs *(*igraph-vs-indexing*)*, 243
[<-.igraph.es *(*igraph-es-attributes*)*, 235
[<-.igraph.vs *(*igraph-vs-attributes*)*, 241
[[.igraph, 13
[[.igraph(), *13*, *21*, *26*, *183*, *192*, *196*, *223*, *227*, *249*, *250*, *263*, *368*, *500*
[[.igraph.es *(*igraph-es-indexing2*)*, 238
[[.igraph.vs *(*igraph-vs-indexing2*)*, 245
[[<-.igraph.es *(*igraph-es-attributes*)*, 235
[[<-.igraph.vs *(*igraph-vs-attributes*)*, 241
$.igraph *(*igraph-dollar*)*, 234
$.igraph.es *(*igraph-es-attributes*)*, 235
$.igraph.es(), *166*
$.igraph.vs *(*igraph-vs-attributes*)*, 241
$.igraph.vs(), *522*
$<-.igraph *(*igraph-dollar*)*, 234
$<-.igraph.es *(*igraph-es-attributes*)*, 235
$<-.igraph.vs *(*igraph-vs-attributes*)*, 241
%--% *(*igraph-es-indexing*)*, 236
%->% *(*igraph-es-indexing*)*, 236
%<-% *(*igraph-es-indexing*)*, 236
%c% *(*compose*)*, 118
%du% *(*disjoint_union*)*, 154
%m% *(*difference.igraph*)*, 149
%s% *(*intersection.igraph*)*, 251
%u% *(*union.igraph*)*, 516
%>%, 15

add_edges, 18
add_edges(), *17*, *20*, *114*, *119*, *121*, *126*, *141*, *143*, *149*, *150*, *155*, *170*, *241*, *251*, *252*, *373*, *375*, *413*, *415*, *483*, *508*, *516*, *517*, *523*
add_layout_, 19
add_layout_(), *117*, *287*, *288*, *290*, *291*, *293*, *294*, *297–300*, *302*, *308*, *310*, *311*, *313*, *315*, *316*, *318*, *357*, *370*
add_shape *(*shapes*)*, 475
add_vertices, 20
add_vertices(), *17*, *18*, *114*, *119*, *121*, *126*, *141*, *143*, *149*, *150*, *155*, *170*, *241*, *251*, *252*, *373*, *375*, *413*, *415*, *483*, *508*, *516*, *517*, *523*
adhesion *(*edge_connectivity*)*, 173
adjacent_vertices, 21
adjacent_vertices(), *13*, *14*, *26*, *183*, *192*, *196*, *223*, *227*, *249*, *250*, *263*, *368*, *500*
algorithm *(*membership*)*, 352
align_layout, 22
all_shortest_paths *(*distance_table*)*, 155
all_shortest_paths(), *282*
all_simple_paths, 22
all_simple_paths(), *148*, *159*, *169*, *200*, *403*
alpha_centrality, 24
alpha_centrality(), *51*, *55*, *83*, *162*, *178*, *225*, *229*, *372*, *392*, *485*, *491*, *495*
any_loop *(*which_multiple*)*, 532
any_multiple *(*which_multiple*)*, 532
ape::as.phylo(), *355*
are_adjacent, 26
are_adjacent(), *13*, *14*, *21*, *183*, *192*, *196*, *223*, *227*, *249*, *250*, *263*, *368*, *500*, *508*
arpack *(*arpack_defaults*)*, 27
arpack(), *28*, *50*, *94*, *177*, *178*, *180*, *182*, *228*, *229*, *372*, *484*
arpack-options *(*arpack_defaults*)*, 27
arpack.unpack.complex *(*arpack_defaults*)*, 27
arpack_defaults, 27
arpack_defaults(), *179*, *182*, *484*
articulation_points, 31
articulation_points(), *59*, *116*, *134*, *138*, *257*
as.dendrogram(), *95*, *355*, *356*, *387*, *389*
as.dendrogram.communities *(*membership*)*, 352
as.factor(), *49*
as.hclust.communities *(*membership*)*, 352

as.igraph, 32
as.integer(), *49*
as.matrix.igraph, 33
as.matrix.igraph(), *35*, *37, 38*, *40*, *42*, *44, 45*, *47*, *202*, *213*
as_adj_edge_list *(*as_adj_list*)*, 34
as_adj_list, 34
as_adj_list(), *34*, *37, 38*, *40*, *42*, *44, 45*, *47*, *201, 202*, *213*
as_adjacency_matrix, *33*, 36
as_adjacency_matrix(), *33–35*, *38*, *40*, *42*, *44, 45*, *47*, *202*, *213*, *485*, *490*
as_biadjacency_matrix, 37
as_biadjacency_matrix(), *34, 35*, *37*, *40*, *42*, *44, 45*, *47*, *202*, *213*
as_bipartite *(*layout_as_bipartite*)*, 289
as_data_frame, 39
as_data_frame(), *34, 35*, *37, 38*, *42*, *44, 45*, *47*, *202*, *209*, *213*
as_directed, 41
as_directed(), *34, 35*, *37, 38*, *40*, *44, 45*, *47*, *202*, *213*
as_edgelist, 43
as_edgelist(), *33–35*, *37, 38*, *40*, *42*, *45*, *47*, *202*, *213*
as_graphnel, 44
as_graphnel(), *34, 35*, *37, 38*, *40*, *42*, *44*, *47*, *202*, *213*
as_ids, 45
as_ids(), *167*, *236*, *238, 239*, *242*, *244*, *246*, *397, 398*, *522*
as_long_data_frame, 46
as_long_data_frame(), *34, 35*, *37, 38*, *40*, *42*, *44, 45*, *202*, *213*
as_membership, 47
as_membership(), *85*, *87*, *89, 90*, *92*, *95*, *97*, *99*, *101*, *104*, *106*, *113*, *222*, *329*, *356*, *365*, *388*, *486*, *530*
as_star *(*layout_as_star*)*, 290
as_tree *(*layout_as_tree*)*, 292
as_undirected *(*as_directed*)*, 41
as_undirected(), *510*
assortativity, 48
assortativity_degree *(*assortativity*)*, 48
assortativity_nominal *(*assortativity*)*, 48
asym_pref *(*sample_pref*)*, 457
atlas *(*graph_from_atlas*)*, 207
attribute.combination *(*igraph-attribute-combination*)*, 232
attribute.combination(), *42*, *126*, *247, 248*, *482*
attributes *(*graph_attr_names*)*, 198
authority_score, 50
authority_score(), *25*, *55*, *83*, *162*, *178*, *225*, *229*, *372*, *392*, *485*, *491*, *495*
automorphism_group, 51
automorphism_group(), *130*
average_local_efficiency *(*global_efficiency*)*, 194

base::options(), *396*
base::save(), *376*
betweenness, 53
betweenness(), *25*, *51*, *83*, *162*, *178*, *225*, *229*, *372*, *392*, *485*, *491*, *495*
bfs, 55
bfs(), *116*, *122*, *125*, *129*, *139*, *146*, *159*, *176*, *185, 186*, *193*, *256*, *261*, *268*, *282*, *284*, *412*, *492*, *494*, *507*, *511*, *515*, *533*, *535*
bibcoupling *(*cocitation*)*, 106
biconnected_components, 58
biconnected_components(), *32*, *116*, *134*, *138*, *257*
bipartite *(*sample_bipartite*)*, 419
bipartite_gnm, 59
bipartite_gnm(), *419*, *421*, *424, 425*, *427, 428*, *433*, *435, 436*, *438*, *440–443*, *445*, *447, 448*, *450*, *454*, *457*, *459, 460*, *463*, *468, 469*
bipartite_gnp *(*bipartite_gnm*)*, 59
bipartite_graph *(*make_bipartite_graph*)*, 325
bipartite_mapping, 61
bipartite_mapping(), *63*, *258*, *326*
bipartite_projection, 62
bipartite_projection(), *62*, *258*, *326*
bipartite_projection_size *(*bipartite_projection*)*, 62
blocks *(*cohesive_blocks*)*, 107
bridges *(*articulation_points*)*, 31

c(), *233*
c.igraph.es, 64

c.igraph.es(), *65*, *151, 152*, *238, 239*, *244*, *246*, *253, 254*, *414*, *518–520*
c.igraph.vs, 65
c.igraph.vs(), *64*, *151, 152*, *238, 239*, *244*, *246*, *253, 254*, *414*, *518–520*
canonical_permutation, 66
canonical_permutation(), *52*, *130, 131*, *135*, *214*, *276*, *278, 279*, *374, 375*, *497*, *499*
categorical_pal, 68, *380*
categorical_pal(), *160*, *402*, *471*
centr_betw, 69
centr_betw(), *70–74*, *76–78*
centr_betw_tmax, 70
centr_betw_tmax(), *69*, *71–74*, *76–78*
centr_clo, 71
centr_clo(), *69, 70*, *72–74*, *76–78*
centr_clo_tmax, 72
centr_clo_tmax(), *69–71*, *73, 74*, *76–78*
centr_degree, 73
centr_degree(), *69–72*, *74*, *76–78*
centr_degree_tmax, 74
centr_degree_tmax(), *69–73*, *76–78*
centr_eigen, 75
centr_eigen(), *69–74*, *77, 78*
centr_eigen_tmax, 76
centr_eigen_tmax(), *69–74*, *76*, *78*
centralization *(*centralize*)*, 77
centralize, 77
centralize(), *69–77*
chordal_ring *(*make_chordal_ring*)*, 326
chung_lu *(*sample_chung_lu*)*, 421
circulant *(*make_circulant*)*, 327
cit_cit_types *(*sample_last_cit*)*, 448
cit_types *(*sample_last_cit*)*, 448
clique_num *(*cliques*)*, 79
clique_size_counts *(*cliques*)*, 79
cliques, 79
cliques(), *260*, *280*, *532*
closeness, 82
closeness(), *25*, *51*, *55*, *162*, *178*, *225*, *229*, *372*, *392*, *485*, *491*, *495*
cluster_edge_betweenness, 84
cluster_edge_betweenness(), *47*, *87*, *89, 90*, *92*, *95*, *97*, *99*, *101*, *104*, *106*, *113*, *222*, *329*, *356*, *365*, *388*, *486*, *530*
cluster_fast_greedy, 86
cluster_fast_greedy(), *47*, *85*, *89, 90*, *92*, *95*, *97*, *99*, *101*, *104*, *106*, *113*, *222*, *329*, *356*, *365*, *388*, *486*, *530*
cluster_fluid_communities, 88
cluster_fluid_communities(), *47*, *85*, *87*, *90*, *92*, *95*, *97*, *99*, *101*, *104*, *106*, *113*, *222*, *329*, *356*, *365*, *388*, *486*, *530*
cluster_infomap, 89
cluster_infomap(), *47*, *85*, *87*, *89*, *92*, *95*, *97*, *99*, *101*, *104*, *106*, *113*, *222*, *329*, *355, 356*, *365*, *388*, *486*, *530*
cluster_label_prop, 91
cluster_label_prop(), *47*, *85*, *87*, *89, 90*, *95*, *97*, *99*, *101*, *104*, *106*, *113*, *222*, *329*, *356*, *365*, *388*, *486*, *530*
cluster_leading_eigen, 93
cluster_leading_eigen(), *30*, *47*, *85*, *87*, *89, 90*, *92*, *97*, *99*, *101*, *104*, *106*, *113*, *222*, *329*, *355, 356*, *365*, *388*, *486*, *530*
cluster_leiden, 95
cluster_leiden(), *47*, *85*, *87*, *89, 90*, *92*, *95*, *99*, *101*, *104*, *106*, *113*, *222*, *329*, *356*, *365*, *388*, *486*, *530*
cluster_louvain, 98
cluster_louvain(), *47*, *85*, *87*, *89, 90*, *92*, *95*, *97*, *101*, *104*, *106*, *113*, *222*, *329*, *356*, *365*, *388*, *486*, *530*
cluster_optimal, 100
cluster_optimal(), *47*, *85*, *87*, *89, 90*, *92*, *95*, *97*, *99*, *104*, *106*, *113*, *222*, *329*, *356*, *365*, *388*, *486*, *530*
cluster_spinglass, 101
cluster_spinglass(), *47*, *85*, *87*, *89, 90*, *92*, *95*, *97*, *99*, *101*, *103*, *106*, *113*, *222*, *329*, *356*, *365*, *388*, *486*, *530*
cluster_walktrap, 104
cluster_walktrap(), *47*, *85*, *87*, *89, 90*, *92*, *95*, *97*, *99*, *101*, *104*, *113*, *222*, *329*, *356*, *365*, *388*, *486*, *530*
cocitation, 106
cocitation(), *480*
code_len *(*membership*)*, 352
cohesion *(*vertex_connectivity*)*, 527
cohesion(), *111*
cohesion.cohesiveBlocks *(*cohesive_blocks*)*, 107

cohesive_blocks, 107
cohesiveBlocks (cohesive_blocks), 107
communities (membership), 352
communities(), 85, 87–90, 92, 97, 99–101, 103–106, 112, 113, 222, 386
compare, 112
compare(), 47, 85, 87, 89, 90, 92, 95, 97, 99, 101, 104, 106, 222, 329, 356, 365, 388, 486, 530
complementer, 114
complementer(), 17, 18, 20, 119, 121, 126, 141, 143, 149, 150, 155, 170, 241, 251, 252, 373, 375, 413, 415, 483, 508, 516, 517, 523
component_distribution, 115
component_distribution(), 32, 57, 59, 122, 125, 129, 134, 138, 139, 146, 159, 176, 185, 186, 193, 256, 257, 261, 268, 282, 284, 412, 492, 494, 507, 511, 515, 533, 535
component_wise, 117
component_wise(), 19, 288, 290, 291, 293–295, 297–300, 302, 308, 310, 311, 313, 315, 316, 318, 357, 370
components (component_distribution), 115
components(), 32, 59, 104, 134, 138, 222, 367, 492
compose, 118
compose(), 17, 18, 20, 114, 121, 126, 141, 143, 149, 150, 155, 170, 241, 251, 252, 373, 375, 413, 415, 483, 508, 516, 517, 523
connect, 119
connect(), 17, 18, 20, 57, 114, 116, 119, 125, 126, 129, 139, 141, 143, 146, 149, 150, 155, 159, 170, 176, 185, 186, 193, 241, 251, 252, 256, 261, 268, 282, 284, 373, 375, 412, 413, 415, 483, 492, 494, 507, 508, 511, 515–517, 523, 533, 535
consensus_tree, 122
consensus_tree(), 32, 188, 230, 231, 394, 400, 446
console, 123
constraint, 124
constraint(), 57, 116, 122, 129, 139, 146, 159, 176, 185, 186, 193, 256, 261, 268, 282, 284, 412, 492, 494, 507, 511, 515, 533, 535
contract, 125
contract(), 17, 18, 20, 114, 119, 121, 141, 143, 149, 150, 155, 170, 232, 241, 251, 252, 373, 375, 413, 415, 483, 508, 516, 517, 523
convex_hull, 127
convex_hull(), 418, 462
coreness, 128
coreness(), 57, 116, 122, 125, 139, 146, 159, 176, 185, 186, 193, 256, 261, 268, 282, 284, 412, 492, 494, 507, 511, 515, 533, 535
count_automorphisms, 129
count_automorphisms(), 52
count_components (component_distribution), 115
count_isomorphisms, 131
count_isomorphisms(), 67, 135, 214, 276, 278, 279, 497, 499
count_loops (which_multiple), 532
count_max_cliques (cliques), 79
count_motifs, 132
count_motifs(), 165, 366, 451
count_multiple (which_multiple), 532
count_multiple(), 483
count_reachable, 133
count_reachable(), 32, 59, 116, 138, 257
count_subgraph_isomorphisms, 134
count_subgraph_isomorphisms(), 67, 131, 214, 276, 278, 279, 497, 499
count_triangles (triangles), 513
crossing (membership), 352
curve_multiple, 136
curve_multiple(), 379
cut_at (membership), 352

de_bruijn_graph (make_de_bruijn_graph), 329
decompose, 137
decompose(), 32, 59, 116, 134, 257
degree, 138
degree(), 55, 57, 116, 122, 125, 129, 146, 159, 176, 185, 186, 193, 256, 261, 268, 282, 284, 372, 412, 491, 492, 494, 507, 511, 515, 533, 535
degree_distribution (degree), 138
degseq (sample_degseq), 427
delete_edge_attr, 140

delete_edge_attr(), *142, 143*, *171–173*, *198, 199*, *233, 234*, *242*, *472–475*, *525–527*
delete_edges, 140
delete_edges(), *17, 18*, *20*, *114*, *119*, *121*, *126*, *143*, *149, 150*, *155*, *170*, *241*, *251, 252*, *373*, *375*, *413*, *415*, *483*, *508*, *516, 517*, *523*
delete_graph_attr, 141
delete_graph_attr(), *140*, *143*, *171–173*, *198, 199*, *233, 234*, *242*, *472–475*, *525–527*
delete_vertex_attr, 142
delete_vertex_attr(), *140*, *142*, *171–173*, *198, 199*, *233, 234*, *242*, *472–475*, *525–527*
delete_vertices, 143
delete_vertices(), *17, 18*, *20*, *114*, *119*, *121*, *126*, *141*, *149, 150*, *155*, *170*, *241*, *251, 252*, *373*, *375*, *413*, *415*, *483*, *508*, *516, 517*, *523*
dendrogram, *356*
dev.capabilities, *377*
dfs, 144
dfs(), *57*, *116*, *122*, *125*, *129*, *139*, *159*, *176*, *185, 186*, *193*, *256*, *261*, *268*, *282*, *284*, *412*, *492*, *494*, *507*, *511*, *515*, *533*, *535*
diameter, 147
diameter(), *23*, *159*, *169*, *200*, *403*
difference, 148
difference(), *17, 18*, *20*, *114*, *119*, *121*, *126*, *141*, *143*, *150*, *155*, *170*, *240, 241*, *251, 252*, *373*, *375*, *413*, *415*, *483*, *508*, *516, 517*, *523*
difference.igraph, 149
difference.igraph(), *17, 18*, *20*, *114*, *119*, *121*, *126*, *141*, *143*, *148, 149*, *155*, *170*, *241*, *251, 252*, *373*, *375*, *413*, *415*, *483*, *508*, *516, 517*, *523*
difference.igraph.es, 150
difference.igraph.es(), *64, 65*, *152*, *238, 239*, *244*, *246*, *253, 254*, *414*, *518–520*
difference.igraph.vs, 151
difference.igraph.vs(), *64, 65*, *148*, *151*, *238, 239*, *244*, *246*, *253, 254*, *414*, *518–520*
dim_select, 152
dim_select(), *180*, *182*
directed_graph *(*make_graph*)*, 336
disjoint_union, 154
disjoint_union(), *16–18*, *20*, *114*, *119*, *121*, *126*, *141*, *143*, *149, 150*, *170*, *241*, *251, 252*, *357*, *373*, *375*, *413*, *415*, *483*, *508*, *516, 517*, *523*
distance_table, 155
distance_table(), *23*, *57*, *116*, *122*, *125*, *129*, *139*, *146*, *148*, *169*, *176*, *185, 186*, *193*, *200*, *256*, *261*, *268*, *282*, *284*, *403*, *412*, *492*, *494*, *507*, *511*, *515*, *533*, *535*
distances, *403*
distances *(*distance_table*)*, 155
distances(), *148*, *169*, *508*, *530*
diverging_pal, 160
diverging_pal(), *68*, *402*, *471*
diversity, 161
diversity(), *25*, *51*, *55*, *83*, *178*, *225*, *229*, *372*, *392*, *485*, *491*, *495*
DL *(*read_graph*)*, 405
dominator_tree, 163
dominator_tree(), *175*, *269*, *272*, *352*, *359, 360*, *363*, *487*, *489*, *529*
dot-data, 164
dot-env *(*dot-data*)*, 164
dot_product *(*sample_dot_product*)*, 432
drl_defaults *(*layout_with_drl*)*, 303
dyad_census, 165
dyad_census(), *133*, *366*, *451*, *513*, *535*

E, 166
E(), *46*, *235, 236*, *238–240*, *242*, *244*, *246*, *397, 398*, *522*
E<- *(*igraph-es-attributes*)*, 235
each_edge, 167
each_edge(), *283*, *416*
eccentricity, 168
eccentricity(), *23*, *148*, *159*, *200*, *403*
ecount *(*gsize*)*, 223
ecount(), *176*
edge, 170
edge(), *16–18*, *20*, *114*, *119*, *121*, *126*, *141*, *143*, *149, 150*, *155*, *240, 241*, *251, 252*, *373*, *375*, *413*, *415*, *483*, *508*, *516, 517*, *523*
edge.attributes *(*edge_attr*)*, 171

edge.attributes<- (edge_attr<-), 172
edge.betweenness.estimate (betweenness), 53
edge_attr, 171
edge_attr(), 140, 142, 143, 172, 173, 198, 199, 233–235, 242, 472–475, 525–527
edge_attr<-, 172
edge_attr_names, 172
edge_attr_names(), 140, 142, 143, 171, 173, 198, 199, 233, 234, 242, 472–475, 525–527
edge_betweenness (betweenness), 53
edge_betweenness(), 85
edge_connectivity, 173
edge_connectivity(), 32, 164, 269, 272, 352, 359, 360, 363, 487, 489, 529
edge_density, 175
edge_density(), 57, 116, 122, 125, 129, 139, 146, 159, 185, 186, 193, 256, 261, 268, 282, 284, 412, 492, 494, 507, 511, 515, 533, 535
edge_disjoint_paths (edge_connectivity), 173
edges (edge), 170
edges(), 16, 240
ego (connect), 119
ego_graph (connect), 119
ego_size (connect), 119
eigen_centrality, 177
eigen_centrality(), 25, 28, 30, 51, 55, 75, 83, 162, 225, 229, 372, 392, 485, 491, 495
embed_adjacency_matrix, 179
embed_adjacency_matrix(), 153, 182
embed_laplacian_matrix, 181
embed_laplacian_matrix(), 153, 180
empty_graph (make_empty_graph), 330
ends, 183
ends(), 13, 14, 21, 26, 192, 196, 223, 227, 249, 250, 263, 368, 500
erdos.renyi.game(), 60, 419, 421, 424, 425, 427, 428, 433, 435, 436, 438, 440–443, 445, 447, 448, 450, 454, 457, 459, 460, 463, 468, 469
eulerian_cycle (has_eulerian_path), 225
eulerian_path (has_eulerian_path), 225
export_pajek (cohesive_blocks), 107
farthest_vertices (diameter), 147
feedback_arc_set, 184
feedback_arc_set(), 57, 116, 122, 125, 129, 139, 146, 159, 176, 186, 193, 226, 256, 261, 268, 282, 284, 412, 481, 492, 494, 507, 511, 515, 533, 535
feedback_vertex_set, 185
feedback_vertex_set(), 57, 116, 122, 125, 129, 139, 146, 159, 176, 185, 186, 193, 226, 256, 261, 268, 282, 284, 412, 481, 492, 494, 507, 511, 515, 533, 535
find_cycle, 186
find_cycle(), 185, 186, 193, 226, 256, 261, 481
fit_hrg, 187
fit_hrg(), 32, 123, 230, 231, 388, 394, 400, 446
fit_power_law, 188
from_adjacency (graph_from_adjacency_matrix), 202
from_data_frame (as_data_frame), 39
from_edgelist (graph_from_edgelist), 209
from_literal (graph_from_literal), 215
from_prufer (make_from_prufer), 331
full_bipartite_graph (make_full_bipartite_graph), 332
full_citation_graph (make_full_citation_graph), 333
full_graph (make_full_graph), 334
full_multipartite (make_full_multipartite), 335

get.edges (ends), 183
get_diameter (diameter), 147
get_edge_ids, 191
get_edge_ids(), 13, 14, 21, 26, 183, 196, 223, 227, 249, 250, 263, 368, 500
getOption(), 248
girth, 192
girth(), 57, 116, 122, 125, 129, 139, 146, 159, 176, 185, 186, 226, 256, 261, 268, 282, 284, 412, 481, 492, 494, 507, 511, 515, 533, 535
global_efficiency, 194
GML (read_graph), 405
gnm (sample_gnm), 439

gnp *(*sample_gnp*)*, 440
gorder, 196
gorder(), *13, 14*, *21*, *26*, *183*, *192*, *223*, *227*, *249, 250*, *263*, *368*, *500*
graph.attributes *(*graph_attr*)*, 197
graph.attributes<- *(*graph_attr<-*)*, 199
graph.count.isomorphisms.vf2 *(*count_isomorphisms*)*, 131
graph.count.subisomorphisms.vf2 *(*count_subgraph_isomorphisms*)*, 134
graph.get.isomorphisms.vf2 *(*isomorphisms*)*, 278
graph.get.subisomorphisms.vf2 *(*subgraph_isomorphisms*)*, 498
graph.isoclass *(*isomorphism_class*)*, 277
graph.isomorphic *(*isomorphic*)*, 275
graph.subisomorphic.lad *(*subgraph_isomorphic*)*, 496
graph.subisomorphic.vf2 *(*subgraph_isomorphic*)*, 496
graph_, 197
graph_attr, 197
graph_attr(), *140*, *142, 143*, *171–173*, *198, 199*, *233, 234*, *242*, *472–475*, *525–527*
graph_attr<-, 199
graph_attr_names, 198
graph_attr_names(), *140*, *142, 143*, *171–173*, *198, 199*, *233, 234*, *242*, *472–475*, *525–527*
graph_center, 200
graph_center(), *23*, *148*, *159*, *169*, *403*
graph_from_adj_list, 201
graph_from_adj_list(), *34, 35*, *37, 38*, *40*, *42*, *44, 45*, *47*, *213*
graph_from_adjacency_matrix, 202
graph_from_adjacency_matrix(), *37*, *44, 45*, *213*
graph_from_atlas, 207
graph_from_atlas(), *210*, *217*, *324*, *327, 328*, *331*, *333–335*, *339*, *341*, *343–346*, *348*
graph_from_biadjacency_matrix, 208
graph_from_biadjacency_matrix(), *38*, *40*
graph_from_data_frame *(*as_data_frame*)*, 39
graph_from_edgelist, 209
graph_from_edgelist(), *207*, *217*, *324*, *327, 328*, *331*, *333–335*, *339*, *341*, *343–346*, *348*
graph_from_graphdb, 210
graph_from_graphdb(), *408*, *542*
graph_from_graphnel, 212
graph_from_graphnel(), *34, 35*, *37, 38*, *40*, *42*, *44, 45*, *47*, *202*
graph_from_isomorphism_class, 213
graph_from_isomorphism_class(), *67*, *131*, *135*, *276*, *278, 279*, *497*, *499*
graph_from_lcf, 214
graph_from_literal, 215
graph_from_literal(), *40*, *204*, *207*, *210*, *324*, *327, 328*, *331*, *333–336*, *339*, *341*, *343–346*, *348*
graph_id, 217
graph_id(), *395*
graph_version, 218
graph_version(), *521*
graphics::par(), *379*
graphics::plot(), *375*
graphics::text(), *377, 378*
graphics::xspline(), *383*
graphlet_basis, 219
graphlet_proj *(*graphlet_basis*)*, 219
graphlets *(*graphlet_basis*)*, 219
GraphML *(*read_graph*)*, 405
graphs_from_cohesive_blocks *(*cohesive_blocks*)*, 107
grDevices::palette(), *377*
greedy_vertex_coloring, 221
grg *(*sample_grg*)*, 441
groups, 222
groups(), *47*, *85*, *87*, *89, 90*, *92*, *95*, *97*, *99*, *101*, *104*, *106*, *113*, *116*, *329*, *356*, *365*, *388*, *486*, *529, 530*
growing *(*sample_growing*)*, 443
gsize, 223
gsize(), *13, 14*, *21*, *26*, *183*, *192*, *196*, *227*, *249, 250*, *263*, *368*, *500*

harmonic_centrality, 224
harmonic_centrality(), *25*, *51*, *55*, *83*, *162*, *178*, *229*, *372*, *392*, *485*, *491*, *495*
has_eulerian_cycle *(*has_eulerian_path*)*, 225
has_eulerian_path, 225

has_eulerian_path(), *185, 186*, *193*, *256*, *261*, *481*
head(), *233*
head_of, 226
head_of(), *13, 14*, *21*, *26*, *183*, *192*, *196*, *223*, *249, 250*, *263*, *368*, *500*
head_print, 227
hierarchical_sbm *(*sample_hierarchical_sbm*)*, 444
hierarchy *(*cohesive_blocks*)*, 107
hits_scores, 228
hits_scores(), *25*, *51*, *55*, *83*, *162*, *178*, *225*, *372*, *392*, *485*, *491*, *495*
hrg, 229
hrg(), *123*, *188*, *230, 231*, *394*, *400*, *446*
hrg-methods, 230
hrg_tree, 231
hrg_tree(), *123*, *188*, *230*, *394*, *400*, *446*
hub_score *(*authority_score*)*, 50
hub_score(), *30*

identical_graphs, 231
igraph-attribute-combination, 232
igraph-dollar, 234
igraph-es-attributes, 235
igraph-es-indexing, 236
igraph-es-indexing2, 238
igraph-minus, 240
igraph-vs-attributes, 241
igraph-vs-indexing, 243
igraph-vs-indexing2, 245
igraph.drl.coarsen *(*layout_with_drl*)*, 303
igraph.drl.coarsest *(*layout_with_drl*)*, 303
igraph.drl.default *(*layout_with_drl*)*, 303
igraph.drl.final *(*layout_with_drl*)*, 303
igraph.drl.refine *(*layout_with_drl*)*, 303
igraph.eigen.default *(*spectrum*)*, 483
igraph.plotting, *136*, *355*, *383*, *417*, *503*
igraph.plotting *(*plot.common*)*, 375
igraph.plotting(), *524*
igraph.vertex.shapes *(*shapes*)*, 475
igraph.vs, *81*, *532*
igraph_opt *(*igraph_options*)*, 246
igraph_opt(), *395*
igraph_options, 246
igraph_options(), *124*, *184, 185*, *233*, *376*, *378*, *381*, *507*, *537*
in_circle *(*layout_in_circle*)*, 294
incident, 248
incident(), *13, 14*, *21*, *26*, *183*, *192*, *196*, *223*, *227*, *249, 250*, *263*, *368*, *500*
incident_edges, 249
incident_edges(), *13, 14*, *21*, *26*, *183*, *192*, *196*, *223*, *227*, *249*, *263*, *368*, *500*
indent_print, 250
independence_number *(*ivs*)*, 279
induced_subgraph *(*subgraph*)*, 493
intersection, 251
intersection(), *17, 18*, *20*, *114*, *119*, *121*, *126*, *141*, *143*, *149, 150*, *155*, *170*, *241*, *252*, *373*, *375*, *413*, *415*, *483*, *508*, *516, 517*, *523*
intersection.igraph, 251
intersection.igraph(), *17, 18*, *20*, *114*, *119*, *121*, *126*, *141*, *143*, *149, 150*, *155*, *170*, *241*, *251*, *373*, *375*, *413*, *415*, *483*, *508*, *516, 517*, *523*
intersection.igraph.es, 253
intersection.igraph.es(), *64, 65*, *151, 152*, *238, 239*, *244*, *246*, *254*, *414*, *518–520*
intersection.igraph.vs, 254
intersection.igraph.vs(), *64, 65*, *151, 152*, *238, 239*, *244*, *246*, *251*, *253*, *414*, *518–520*
invalidate_cache, 255
is_acyclic, 256
is_acyclic(), *57*, *116*, *122*, *125*, *129*, *139*, *146*, *159*, *176*, *185, 186*, *193*, *226*, *261*, *268*, *282*, *284*, *412*, *481*, *492*, *494*, *507*, *511*, *515*, *533*, *535*
is_biconnected, 257
is_biconnected(), *32*, *59*, *116*, *134*, *138*
is_bipartite, 258
is_bipartite(), *62, 63*, *326*
is_chordal, 258
is_chordal(), *350*
is_clique *(*cliques*)*, 79
is_complete, 260
is_complete(), *81*, *280*, *532*
is_connected *(*component_distribution*)*, 115
is_connected(), *32*, *59*, *134*, *138*, *257*

is_dag, *255*, 261
is_dag(), *57*, *116*, *122*, *125*, *129*, *139*, *146*, *159*, *176*, *185*, *186*, *193*, *226*, *256*, *268*, *282*, *284*, *412*, *481*, *492*, *494*, *507*, *511*, *515*, *533*, *535*
is_degseq, 262
is_degseq(), *266*
is_directed, 263
is_directed(), *13*, *14*, *21*, *26*, *183*, *192*, *196*, *223*, *227*, *249*, *250*, *368*, *500*
is_forest, 264
is_forest(), *256*, *273*, *331*, *464*, *506*
is_graphical, 265
is_graphical(), *263*
is_hierarchical *(*membership*)*, 352
is_igraph, 266
is_isomorphic_to *(*isomorphic*)*, 275
is_ivs *(*ivs*)*, 279
is_matching, 267
is_matching(), *57*, *116*, *122*, *125*, *129*, *139*, *146*, *159*, *176*, *185*, *186*, *193*, *256*, *261*, *282*, *284*, *412*, *492*, *494*, *507*, *511*, *515*, *533*, *535*
is_max_matching *(*is_matching*)*, 267
is_min_separator, 269
is_min_separator(), *164*, *175*, *272*, *352*, *359*, *360*, *363*, *487*, *489*, *529*
is_named, 270
is_printer_callback, 271
is_printer_callback(), *401*
is_separator, 272
is_separator(), *164*, *175*, *269*, *352*, *359*, *360*, *363*, *487*, *489*, *529*
is_simple, *255*
is_simple *(*simplify*)*, 482
is_simple(), *108*
is_subgraph_isomorphic_to *(*subgraph_isomorphic*)*, 496
is_tree, 273
is_tree(), *264*, *331*, *464*, *506*
is_weighted, 274
isomorphic, 275
isomorphic(), *67*, *131*, *135*, *212*, *214*, *278*, *279*, *497*, *499*
isomorphism_class, 277
isomorphism_class(), *67*, *131*, *133*, *135*, *214*, *276*, *279*, *366*, *451*, *497*, *499*
isomorphisms, 278
isomorphisms(), *67*, *131*, *135*, *214*, *276*, *278*, *497*, *499*
ivs, 279
ivs(), *81*, *260*, *532*
ivs_size *(*ivs*)*, 279

k_shortest_paths, 281
k_shortest_paths(), *57*, *116*, *122*, *125*, *129*, *139*, *146*, *159*, *176*, *185*, *186*, *193*, *256*, *261*, *268*, *284*, *412*, *492*, *494*, *507*, *511*, *515*, *533*, *535*
kautz_graph *(*make_kautz_graph*)*, 339
keeping_degseq, 282
keeping_degseq(), *168*, *416*
knn, 283
knn(), *57*, *116*, *122*, *125*, *129*, *139*, *146*, *159*, *176*, *185*, *186*, *193*, *256*, *261*, *268*, *282*, *412*, *492*, *494*, *507*, *511*, *515*, *533*, *535*

laplacian_matrix, 285
largest_cliques *(*cliques*)*, 79
largest_component *(*component_distribution*)*, 115
largest_ivs *(*ivs*)*, 279
largest_weighted_cliques *(*cliques*)*, 79
last_cit *(*sample_last_cit*)*, 448
lattice *(*make_lattice*)*, 340
layout *(*layout_*)*, 287
layout(), *291*, *298*, *306*, *316*, *357*, *383*, *506*
layout_, 287
layout_(), *19*, *117*, *290*, *291*, *293–295*, *297–300*, *302*, *308*, *310*, *311*, *313*, *315*, *316*, *318*, *357*, *370*
layout_as_bipartite, 289
layout_as_bipartite(), *19*, *117*, *288*, *291*, *293*, *294*, *297–300*, *302*, *308*, *310*, *311*, *313*, *315*, *316*, *318*, *357*, *370*
layout_as_star, 290
layout_as_star(), *19*, *117*, *288*, *290*, *293*, *294*, *297–300*, *302*, *308*, *310*, *311*, *313*, *315*, *316*, *318*, *357*, *370*
layout_as_tree, 292
layout_as_tree(), *19*, *117*, *288*, *290*, *291*, *294*, *297–300*, *302*, *308*, *310*, *311*, *313*, *315*, *316*, *318*, *357*, *370*
layout_components *(*merge_coords*)*, 356
layout_in_circle, 294

layout_in_circle(), *19*, *117*, *288*, *290, 291*, *293*, *297–300*, *302*, *308*, *310, 311*, *313*, *315, 316*, *318*, *357*, *370*
layout_modifier, 295
layout_modifier(), *117*, *370*
layout_nicely, 296
layout_nicely(), *19*, *117*, *288*, *290, 291*, *293, 294*, *298–300*, *302*, *308*, *310, 311*, *313*, *315, 316*, *318*, *357*, *370*
layout_on_grid, 297
layout_on_grid(), *19*, *117*, *288*, *290, 291*, *293, 294*, *297*, *299, 300*, *302*, *308*, *310, 311*, *313*, *315, 316*, *318*, *357*, *370*
layout_on_sphere, 299
layout_on_sphere(), *19*, *117*, *288*, *290, 291*, *293, 294*, *297, 298*, *300*, *302*, *308*, *310, 311*, *313*, *315, 316*, *318*, *357*, *370*
layout_randomly, 300
layout_randomly(), *19*, *117*, *288*, *290, 291*, *293, 294*, *297–299*, *302*, *308*, *310, 311*, *313*, *315, 316*, *318*, *357*, *370*
layout_with_dh, 301
layout_with_dh(), *19*, *117*, *288*, *290, 291*, *293, 294*, *297–300*, *308*, *310, 311*, *313*, *315, 316*, *318*, *357*, *370*
layout_with_drl, 303
layout_with_drl(), *291*, *308*, *313*
layout_with_fr, 306
layout_with_fr(), *19*, *22*, *117*, *288*, *290, 291*, *293, 294*, *297–300*, *302*, *310–313*, *315, 316*, *318*, *357*, *370*
layout_with_gem, 308
layout_with_gem(), *19*, *117*, *288*, *290, 291*, *293, 294*, *297–300*, *302*, *308*, *311*, *313*, *315, 316*, *318*, *357*, *370*
layout_with_graphopt, 310
layout_with_graphopt(), *19*, *117*, *288*, *290, 291*, *293, 294*, *297–300*, *302*, *308*, *310*, *313*, *315, 316*, *318*, *357*, *370*
layout_with_kk, 311
layout_with_kk(), *19*, *117*, *288*, *290, 291*, *293, 294*, *297–300*, *302*, *308*, *310, 311*, *315, 316*, *318*, *357*, *370*
layout_with_lgl, 314
layout_with_lgl(), *19*, *117*, *288*, *290, 291*, *293, 294*, *297–300*, *302*, *308*, *310, 311*, *313*, *316*, *318*, *357*, *370*
layout_with_mds, 315
layout_with_mds(), *19*, *117*, *288*, *290, 291*, *293, 294*, *297–300*, *302*, *308*, *310, 311*, *313*, *315*, *318*, *357*, *370*
layout_with_sugiyama, 317
layout_with_sugiyama(), *19*, *117*, *288–291*, *293, 294*, *297–300*, *302*, *308*, *310, 311*, *313*, *315, 316*, *357*, *370*
length.cohesiveBlocks *(*cohesive_blocks*)*, 107
length.communities *(*membership*)*, 352
LGL *(*read_graph*)*, 405
line_graph *(*make_line_graph*)*, 342
local_efficiency *(*global_efficiency*)*, 194
local_scan, 321
local_scan(), *470*

make_, 323
make_(), *207*, *210*, *217*, *327, 328*, *331*, *333–335*, *339*, *341*, *343–346*, *348*, *419*, *482*, *536*, *538, 539*
make_bipartite_graph, 325
make_bipartite_graph(), *62, 63*, *209*, *258*
make_chordal_ring, 326
make_chordal_ring(), *207*, *210*, *217*, *324*, *328*, *331*, *333–335*, *339*, *341*, *343–346*, *348*
make_circulant, 327
make_circulant(), *207*, *210*, *217*, *324*, *327*, *331*, *333–335*, *339*, *341*, *343–346*, *348*
make_clusters, 328
make_clusters(), *47*, *85*, *87*, *89, 90*, *92*, *95*, *97*, *99*, *101*, *104*, *106*, *113*, *222*, *356*, *365*, *388*, *486*, *530*
make_de_bruijn_graph, 329
make_de_bruijn_graph(), *340*
make_directed_graph *(*make_graph*)*, 336
make_ego_graph *(*connect*)*, 119
make_empty_graph, 330
make_empty_graph(), *207*, *210*, *217*, *324*, *327, 328*, *333–335*, *339*, *341*, *343–346*, *348*
make_from_prufer, 331
make_from_prufer(), *264*, *273*, *464*, *506*
make_full_bipartite_graph, 332
make_full_citation_graph, 333

make_full_citation_graph(), *207*, *210*, *217*, *324*, *327, 328*, *331*, *334, 335*, *339*, *341*, *343–346*, *348*
make_full_graph, 334
make_full_graph(), *207*, *210*, *217*, *260*, *324*, *327, 328*, *331*, *333*, *335*, *339*, *341*, *343–346*, *348*
make_full_multipartite, 335
make_full_multipartite(), *207*, *210*, *217*, *324*, *327, 328*, *331*, *333, 334*, *339*, *341*, *343–346*, *348*
make_graph, 336
make_graph(), *204*, *207*, *210*, *215*, *217*, *324–328*, *331*, *333–335*, *341*, *343–346*, *348*
make_kautz_graph, 339
make_kautz_graph(), *330*
make_lattice, 340
make_lattice(), *207*, *210*, *217*, *324*, *327, 328*, *331*, *333–335*, *339*, *343–346*, *348*, *463*
make_line_graph, 342
make_line_graph(), *330*, *340*
make_neighborhood_graph *(*connect*)*, 119
make_ring, 343
make_ring(), *207*, *210*, *217*, *324*, *327, 328*, *331*, *333–335*, *339*, *341*, *344–346*, *348*
make_star, 344
make_star(), *207*, *210*, *217*, *324*, *327, 328*, *331*, *333–335*, *339*, *341*, *343*, *345, 346*, *348*
make_tree, 345
make_tree(), *207*, *210*, *217*, *324*, *327, 328*, *331*, *333–335*, *339*, *341*, *343, 344*, *346*, *348*
make_turan, 346
make_turan(), *207*, *210*, *217*, *324*, *327, 328*, *331*, *333–335*, *339*, *341*, *343–345*, *348*
make_undirected_graph *(*make_graph*)*, 336
make_wheel, 347
make_wheel(), *207*, *210*, *217*, *324*, *327, 328*, *331*, *333–335*, *339*, *341*, *343–346*
match_vertices, 348
max_bipartite_match *(*is_matching*)*, 267
max_cardinality, 349
max_cardinality(), *259*
max_cliques *(*cliques*)*, 79
max_cohesion *(*cohesive_blocks*)*, 107
max_degree *(*degree*)*, 138
max_flow, 351
max_flow(), *164*, *175*, *269*, *272*, *359, 360*, *363*, *487*, *489*, *529*
max_ivs *(*ivs*)*, 279
mean(), *233*
mean_degree *(*degree*)*, 138
mean_distance *(*distance_table*)*, 155
median(), *233*
median.sir *(*time_bins*)*, 500
membership, 352
membership(), *47*, *85*, *87*, *89, 90*, *92*, *95*, *97*, *99*, *101*, *104*, *106*, *113*, *222*, *329*, *365*, *388*, *486*, *530*
merge_coords, 356
merge_coords(), *19*, *117*, *288*, *290, 291*, *293, 294*, *297–300*, *302*, *308*, *310, 311*, *313*, *315, 316*, *318*, *370*
merges *(*membership*)*, 352
min_cut, 358
min_cut(), *164*, *175*, *269*, *272*, *352*, *360*, *363*, *487*, *489*, *529*
min_separators, 360
min_separators(), *164*, *175*, *269*, *272*, *352*, *359*, *363*, *487*, *489*, *529*
min_st_separators, 361
min_st_separators(), *164*, *175*, *269*, *272*, *352*, *359, 360*, *487*, *489*, *529*
modularity *(*modularity.igraph*)*, 363
modularity(), *95*, *101*, *106*
modularity.communities *(*membership*)*, 352
modularity.igraph, 363
modularity.igraph(), *47*, *85*, *87*, *89, 90*, *92*, *95*, *97*, *99*, *101*, *104*, *106*, *113*, *222*, *329*, *354*, *356*, *388*, *486*, *530*
modularity_matrix *(*modularity.igraph*)*, 363
motifs, 365
motifs(), *133*, *165*, *451*, *512, 513*
mst, 366

neighborhood *(*connect*)*, 119
neighborhood_size *(*connect*)*, 119
neighbors, 368
neighbors(), *13, 14*, *21*, *26*, *183*, *192*, *196*, *223*, *227*, *249, 250*, *263*, *500*
nicely *(*layout_nicely*)*, 296

norm_coords, 369
norm_coords(), *19*, *117*, *288*, *290, 291*, *293, 294*, *297–300*, *302*, *308*, *310, 311*, *313*, *315, 316*, *318*, *357*, *370*
normalize, 370
normalize(), *19*, *117*, *288*, *290, 291*, *293–295*, *297–300*, *302*, *308*, *310, 311*, *313*, *315, 316*, *318*, *357*, *370*

on_grid *(*layout_on_grid*)*, 297
on_sphere *(*layout_on_sphere*)*, 299
options(), *248*

pa *(*sample_pa*)*, 451
pa(), *419*
pa_age *(*sample_pa_age*)*, 454
page_rank, 371
page_rank(), *25*, *28*, *30*, *51*, *55*, *83*, *162*, *178*, *225*, *229*, *392*, *485*, *491*, *495*
Pajek *(*read_graph*)*, 405
parent *(*cohesive_blocks*)*, 107
path, 373
path(), *16–18*, *20*, *114*, *119*, *121*, *126*, *141*, *143*, *149, 150*, *155*, *170*, *241*, *251, 252*, *375*, *413*, *415*, *483*, *508*, *516, 517*, *523*
permute, 374
permute(), *17, 18*, *20*, *52*, *67*, *114*, *119*, *121*, *126*, *130*, *141*, *143*, *149, 150*, *155*, *170*, *241*, *251, 252*, *276*, *294*, *298, 299*, *373*, *413*, *415*, *483*, *508*, *516, 517*, *523*
plot(), *108*, *110, 111*, *355, 356*, *380*, *383*, *385*
plot.cohesiveBlocks *(*cohesive_blocks*)*, 107
plot.common, 375
plot.communities *(*membership*)*, 352
plot.graph *(*plot.igraph*)*, 382
plot.hclust(), *353*
plot.igraph, 382
plot.igraph(), *136*, *291*, *297*, *310*, *313*, *316*, *353*, *355*, *357*, *375–378*, *381*, *417*, *506*, *524*
plot.sir, 384
plot.sir(), *502*
plot_dendrogram, 386
plot_dendrogram(), *47*, *85*, *87*, *89, 90*, *92*, *95*, *97*, *99*, *101*, *104*, *106*, *113*, *222*, *247*, *329*, *356*, *365*, *486*, *530*
plot_dendrogram.igraphHRG, 388
plot_hierarchy *(*cohesive_blocks*)*, 107
power_centrality, 390
power_centrality(), *25*, *51*, *55*, *83*, *162*, *178*, *225*, *229*, *372*, *485*, *491*, *495*
predict_edges, 392
predict_edges(), *123*, *188*, *230, 231*, *400*, *446*
pref *(*sample_pref*)*, 457
print, *250*
print(), *108*, *110, 111*, *287*, *354, 355*, *396, 397*
print.cohesiveBlocks *(*cohesive_blocks*)*, 107
print.communities *(*membership*)*, 352
print.igraph, 394
print.igraph(), *247*
print.igraph.es, 396
print.igraph.es(), *46*, *167*, *236*, *238, 239*, *242*, *244*, *246*, *398*, *522*
print.igraph.vs, 397
print.igraph.vs(), *46*, *167*, *236*, *238, 239*, *242*, *244*, *246*, *397*, *522*
print.igraph_layout_modifier *(*layout_*)*, 287
print.igraph_layout_spec *(*layout_*)*, 287
print.igraphHRG, 399
print.igraphHRG(), *123*, *188*, *230, 231*, *394*, *400*, *446*
print.igraphHRGConsensus, 400
print.igraphHRGConsensus(), *123*, *188*, *230, 231*, *394*, *400*, *446*
print_all *(*print.igraph*)*, 394
printer_callback, 401
printer_callback(), *227*, *271*

quantile.sir *(*time_bins*)*, 500

r_pal, 402
r_pal(), *68*, *160*, *471*
radius, 402
radius(), *23*, *148*, *159*, *169*, *200*
random_edge_walk *(*random_walk*)*, 403
random_walk, 403
randomly *(*layout_randomly*)*, 300
read.csv(), *39*
read.delim(), *39*
read.table(), *39, 40*
read_graph, 405

read_graph(), *37*, *44*, *211, 212*, *542*
realize_bipartite_degseq, 408
realize_degseq, 409
realize_degseq(), *409*, *428*
reciprocity, 411
reciprocity(), *57*, *116*, *122*, *125*, *129*, *139*, *146*, *159*, *176*, *185, 186*, *193*, *256*, *261*, *268*, *282*, *284*, *492*, *494*, *507*, *511*, *515*, *533*, *535*
rep.igraph, 412
rep.igraph(), *17, 18*, *20*, *114*, *119*, *121*, *126*, *141*, *143*, *149, 150*, *155*, *170*, *241*, *251, 252*, *373*, *375*, *415*, *483*, *508*, *516, 517*, *523*
rev.igraph.es, 413
rev.igraph.es(), *64, 65*, *151, 152*, *238, 239*, *244*, *246*, *253, 254*, *414*, *518–520*
rev.igraph.vs, 414
rev.igraph.vs(), *64, 65*, *151, 152*, *238, 239*, *244*, *246*, *253, 254*, *414*, *518–520*
reverse_edges, 415
reverse_edges(), *17, 18*, *20*, *114*, *119*, *121*, *126*, *141*, *143*, *149, 150*, *155*, *170*, *241*, *251, 252*, *373*, *375*, *413*, *483*, *508*, *516, 517*, *523*
rewire, 416
rewire(), *167, 168*, *282, 283*, *463*
rglplot, 416
rglplot(), *136*, *376–381*, *383*
ring *(*make_ring*)*, 343
ring(), *324*
running_mean, 417
running_mean(), *127*, *462*

sample_, 418
sample_(), *60*, *324*, *421*, *424, 425*, *427, 428*, *433*, *435, 436*, *438*, *440–443*, *445*, *447, 448*, *450*, *454*, *457*, *459, 460*, *463*, *468, 469*, *482*, *536*, *538, 539*
sample_asym_pref *(*sample_pref*)*, 457
sample_bipartite, 419
sample_bipartite(), *60*, *419*, *424, 425*, *427, 428*, *433*, *435, 436*, *438*, *440–443*, *445*, *447, 448*, *450*, *454*, *457*, *459, 460*, *463*, *468, 469*
sample_bipartite_gnm *(*bipartite_gnm*)*, 59
sample_bipartite_gnm(), *419*
sample_bipartite_gnp *(*bipartite_gnm*)*, 59
sample_bipartite_gnp(), *419*
sample_chung_lu, 421
sample_chung_lu(), *60*, *419*, *421*, *425*, *427, 428*, *433*, *435, 436*, *438*, *440–443*, *445*, *447, 448*, *450*, *454*, *457*, *459, 460*, *463*, *468, 469*
sample_cit_cit_types *(*sample_last_cit*)*, 448
sample_cit_types *(*sample_last_cit*)*, 448
sample_correlated_gnp, 424
sample_correlated_gnp(), *60*, *349*, *419*, *421*, *424*, *427, 428*, *433*, *435, 436*, *438*, *440–443*, *445*, *447, 448*, *450*, *454*, *457*, *459, 460*, *463*, *468, 469*
sample_correlated_gnp_pair, 426
sample_correlated_gnp_pair(), *60*, *349*, *419*, *421*, *424, 425*, *428*, *433*, *435, 436*, *438*, *440–443*, *445*, *447, 448*, *450*, *454*, *457*, *459, 460*, *463*, *468, 469*
sample_degseq, 427
sample_degseq(), *60*, *283*, *411*, *419*, *421*, *424, 425*, *427*, *433–436*, *438*, *440–443*, *445*, *447, 448*, *450*, *454*, *457*, *459, 460*, *463*, *468, 469*
sample_dirichlet, 431
sample_dirichlet(), *433*, *465, 466*
sample_dot_product, 432
sample_dot_product(), *60*, *180*, *182*, *419*, *421*, *424, 425*, *427, 428*, *435, 436*, *438*, *440–443*, *445*, *447, 448*, *450*, *454*, *457*, *459, 460*, *463*, *468, 469*
sample_fitness, 433
sample_fitness(), *60*, *419*, *421*, *424, 425*, *427, 428*, *433*, *436*, *438*, *440–443*, *445*, *447, 448*, *450*, *454*, *457*, *459, 460*, *463*, *468, 469*
sample_fitness_pl, 435
sample_fitness_pl(), *60*, *419*, *421*, *424, 425*, *427, 428*, *433–435*, *438*, *440–443*, *445*, *447, 448*, *450*, *454*, *457*, *459, 460*, *463*, *468, 469*
sample_forestfire, 437
sample_forestfire(), *60*, *419*, *421*, *424, 425*, *427, 428*, *433*, *435, 436*, *440–443*, *445*, *447, 448*, *450*, *454*, *457*, *459, 460*, *463*, *468, 469*
sample_gnm, 439
sample_gnm(), *60*, *419*, *421*, *424, 425*, *427,*

428, *433*, *435, 436*, *438*, *441–443*, *445*, *447, 448*, *450*, *454*, *457*, *459, 460*, *463*, *468, 469*
sample_gnp, 440
sample_gnp(), *60*, *419*, *421*, *424, 425*, *427, 428*, *433*, *435, 436*, *438*, *440*, *442, 443*, *445*, *447, 448*, *450*, *454*, *457*, *459, 460*, *463*, *468, 469*
sample_grg, 441
sample_grg(), *60*, *419*, *421*, *424, 425*, *427*, *429*, *433*, *435, 436*, *438*, *440, 441*, *443*, *445*, *447, 448*, *450*, *454*, *457*, *459, 460*, *463*, *468, 469*
sample_growing, 443
sample_growing(), *60*, *419*, *421*, *424, 425*, *427*, *429*, *433*, *435, 436*, *438*, *440–442*, *445*, *447, 448*, *450*, *454*, *457*, *459, 460*, *463*, *468, 469*
sample_hierarchical_sbm, 444
sample_hierarchical_sbm(), *60*, *419*, *421*, *424, 425*, *427*, *429*, *433*, *435, 436*, *438*, *440–443*, *447, 448*, *450*, *454*, *457*, *459, 460*, *463*, *468, 469*
sample_hrg, 445
sample_hrg(), *123*, *188*, *230, 231*, *394*, *400*
sample_islands, 446
sample_islands(), *60*, *419*, *421*, *424, 425*, *427*, *429*, *433*, *435, 436*, *438*, *440–443*, *445*, *448*, *450*, *454*, *457*, *459, 460*, *463*, *468, 469*
sample_k_regular, 447
sample_k_regular(), *60*, *419*, *421*, *424, 425*, *427*, *429*, *433*, *435, 436*, *438*, *440–443*, *445*, *447*, *450*, *454*, *457*, *459, 460*, *463*, *468, 469*
sample_last_cit, 448
sample_last_cit(), *60*, *419*, *421*, *424, 425*, *427*, *429*, *433*, *435*, *437, 438*, *440–443*, *445*, *447, 448*, *454*, *457*, *459, 460*, *463*, *468, 469*
sample_motifs, 450
sample_motifs(), *133*, *165*, *366*
sample_pa, 451
sample_pa(), *60*, *419*, *421*, *424, 425*, *427*, *429*, *433*, *435*, *437, 438*, *440–443*, *445*, *447, 448*, *450*, *457*, *459, 460*, *463*, *468, 469*
sample_pa_age, 454
sample_pa_age(), *60*, *419*, *421*, *424, 425*, *427*, *429*, *433*, *435*, *437, 438*, *440–443*, *445*, *447, 448*, *450*, *454*, *459, 460*, *463*, *468, 469*
sample_pref, 457
sample_pref(), *60*, *419*, *421*, *424, 425*, *427*, *429*, *433*, *435*, *437, 438*, *440–443*, *445*, *447, 448*, *450*, *454*, *457*, *460*, *463*, *468, 469*
sample_sbm, 459
sample_sbm(), *60*, *419*, *421*, *424, 425*, *427*, *429*, *433*, *435*, *437, 438*, *440–443*, *445*, *447, 448*, *450*, *454*, *457*, *459*, *463*, *468, 469*
sample_seq, 461
sample_seq(), *127*, *418*
sample_smallworld, 462
sample_smallworld(), *60*, *419*, *421*, *424, 425*, *427*, *429*, *433*, *435*, *437, 438*, *440–443*, *445*, *447, 448*, *450*, *454*, *457*, *459, 460*, *468, 469*
sample_spanning_tree, 463
sample_spanning_tree(), *264*, *273*, *331*, *506*
sample_sphere_surface, 464
sample_sphere_surface(), *431*, *433*, *466*
sample_sphere_volume, 465
sample_sphere_volume(), *431*, *433*, *465*
sample_traits *(*sample_traits_callaway*)*, 466
sample_traits_callaway, 466
sample_traits_callaway(), *60*, *419*, *421*, *424, 425*, *427*, *429*, *433*, *435*, *437, 438*, *440–443*, *445*, *447, 448*, *450*, *454*, *457*, *459, 460*, *463*, *469*
sample_tree, 468
sample_tree(), *60*, *419*, *421*, *424, 425*, *427*, *429*, *433*, *435*, *437, 438*, *440–443*, *445*, *447, 448*, *450*, *454*, *457*, *459, 460*, *463*, *468*
sbm *(*sample_sbm*)*, 459
scan_stat, 469
scan_stat(), *323*
seeded.graph.match *(*match_vertices*)*, 348
sequential_pal, 471
sequential_pal(), *68*, *160*, *402*
set_edge_attr, 472
set_edge_attr(), *140*, *142, 143*, *171–173*,

198, *199*, *233–235*, *242*, *473–475*, *525–527*
set_graph_attr, 473
set_graph_attr(), *140*, *142*, *143*, *171–173*, *198*, *199*, *233*, *234*, *242*, *472*, *474*, *475*, *525–527*
set_vertex_attr, 473
set_vertex_attr(), *140*, *142*, *143*, *171–173*, *198*, *199*, *233*, *234*, *242*, *472*, *473*, *475*, *525–527*
set_vertex_attrs, 474
set_vertex_attrs(), *140*, *142*, *143*, *171–173*, *198*, *199*, *233*, *234*, *242*, *472–474*, *525–527*
shape_noclip *(*shapes*)*, 475
shape_noplot *(*shapes*)*, 475
shapes, 475
shapes(), *377*
shortest_paths *(*distance_table*)*, 155
shortest_paths(), *282*
show_trace *(*membership*)*, 352
similarity, 478
similarity(), *107*
simple_cycles, 480
simple_cycles(), *185*, *186*, *193*, *226*, *256*, *261*
simplified, 481
simplified(), *324*, *419*, *536*, *538*, *539*
simplify, 482
simplify(), *17*, *18*, *20*, *42*, *114*, *118*, *119*, *121*, *126*, *141*, *143*, *149*, *150*, *155*, *170*, *176*, *215*, *232*, *241*, *251*, *252*, *373*, *375*, *413*, *415*, *428*, *462*, *508*, *516*, *517*, *523*, *533*
simplify_and_colorize *(*simplify*)*, 482
sir *(*time_bins*)*, 500
sir(), *384*, *385*
sizes *(*membership*)*, 352
smallworld *(*sample_smallworld*)*, 462
solve(), *24*, *390*
spectrum, 483
spectrum(), *25*, *51*, *55*, *83*, *162*, *178*, *225*, *229*, *372*, *392*, *491*, *495*
split_join_distance, 485
split_join_distance(), *47*, *85*, *87*, *89*, *90*, *92*, *95*, *97*, *99*, *101*, *104*, *106*, *113*, *222*, *329*, *356*, *365*, *388*, *530*
st_cuts, 486
st_cuts(), *164*, *175*, *269*, *272*, *352*, *359*, *360*, *363*, *489*, *529*
st_min_cuts, 488
st_min_cuts(), *164*, *175*, *269*, *272*, *352*, *359*, *360*, *363*, *487*, *529*
star *(*make_star*)*, 344
star(), *291*
stats4::mle(), *189*, *190*
stats::as.hclust(), *355*
stats::dendrogram(), *355*
stochastic_matrix, 489
str.igraph *(*print.igraph*)*, 394
strength, 490
strength(), *25*, *51*, *55*, *83*, *162*, *178*, *181*, *225*, *229*, *284*, *372*, *392*, *485*, *495*, *510*
subcomponent, 492
subcomponent(), *57*, *116*, *122*, *125*, *129*, *134*, *139*, *146*, *159*, *176*, *185*, *186*, *193*, *256*, *261*, *268*, *282*, *284*, *412*, *494*, *507*, *511*, *515*, *533*, *535*
subgraph, 493
subgraph(), *57*, *116*, *122*, *125*, *129*, *139*, *146*, *159*, *176*, *185*, *186*, *193*, *256*, *261*, *268*, *282*, *284*, *412*, *492*, *507*, *511*, *515*, *533*, *535*
subgraph_centrality, 494
subgraph_centrality(), *25*, *51*, *55*, *83*, *162*, *178*, *225*, *229*, *372*, *392*, *485*, *491*
subgraph_from_edges *(*subgraph*)*, 493
subgraph_from_edges(), *464*
subgraph_isomorphic, 496
subgraph_isomorphic(), *67*, *131*, *135*, *214*, *276*, *278*, *279*, *499*
subgraph_isomorphisms, 498
subgraph_isomorphisms(), *67*, *131*, *135*, *214*, *276*, *278*, *279*, *497*
summary(), *108*, *110*, *111*
summary.cohesiveBlocks *(*cohesive_blocks*)*, 107
summary.igraph *(*print.igraph*)*, 394

t.igraph *(*reverse_edges*)*, 415
tail(), *233*
tail_of, 500
tail_of(), *13*, *14*, *21*, *26*, *183*, *192*, *196*, *223*, *227*, *249*, *250*, *263*, *368*
tcltk::tkfont.create(), *378*
time_bins, 500

time_bins(), *385*
tk_canvas *(*tkplot*)*, 503
tk_center *(*tkplot*)*, 503
tk_close *(*tkplot*)*, 503
tk_coords *(*tkplot*)*, 503
tk_coords(), *383*
tk_fit *(*tkplot*)*, 503
tk_off *(*tkplot*)*, 503
tk_postscript *(*tkplot*)*, 503
tk_reshape *(*tkplot*)*, 503
tk_rotate *(*tkplot*)*, 503
tk_set_coords *(*tkplot*)*, 503
tkplot, 503
tkplot(), *136*, *291*, *310*, *313*, *357*, *376–379*, *381*, *383*, *417*
to_prufer, 506
to_prufer(), *264*, *273*, *331*, *464*
topo_sort, 507
topo_sort(), *57*, *116*, *122*, *125*, *129*, *139*, *146*, *159*, *176*, *185, 186*, *193*, *256*, *261*, *268*, *282*, *284*, *412*, *492*, *494*, *511*, *515*, *533*, *535*
traits *(*sample_traits_callaway*)*, 466
traits_callaway *(*sample_traits_callaway*)*, 466
transitive_closure, 508
transitive_closure(), *17, 18*, *20*, *114*, *119*, *121*, *126*, *141*, *143*, *149, 150*, *155*, *170*, *241*, *251, 252*, *373*, *375*, *413*, *415*, *483*, *516, 517*, *523*
transitivity, 509
transitivity(), *57*, *116*, *122*, *125*, *129*, *139*, *146*, *159*, *176*, *185, 186*, *193*, *256*, *261*, *268*, *282*, *284*, *412*, *492*, *494*, *507*, *514, 515*, *533*, *535*
tree *(*make_tree*)*, 345
triad_census, 511
triad_census(), *165*
triangles, 513
turan *(*make_turan*)*, 346

UCINET *(*read_graph*)*, 405
undirected_graph *(*make_graph*)*, 336
unfold_tree, 514
unfold_tree(), *57*, *116*, *122*, *125*, *129*, *139*, *146*, *159*, *176*, *185, 186*, *193*, *256*, *261*, *268*, *282*, *284*, *412*, *492*, *494*, *507*, *511*, *533*, *535*
union, 515
union(), *16–18*, *20*, *114*, *119*, *121*, *126*, *141*, *143*, *149, 150*, *155*, *170*, *241*, *251, 252*, *373*, *375*, *413*, *415*, *483*, *508*, *517*, *523*
union.igraph, 516
union.igraph(), *17, 18*, *20*, *114*, *119*, *121*, *126*, *141*, *143*, *149, 150*, *155*, *170*, *241*, *251, 252*, *373*, *375*, *413*, *415*, *483*, *508*, *515, 516*, *523*
union.igraph.es, 517
union.igraph.es(), *64, 65*, *151, 152*, *238, 239*, *244*, *246*, *253, 254*, *414*, *518–520*
union.igraph.vs, 518
union.igraph.vs(), *64, 65*, *151, 152*, *238, 239*, *244*, *246*, *253, 254*, *414*, *515*, *518–520*
unique.igraph.es, 519
unique.igraph.es(), *64, 65*, *151, 152*, *238, 239*, *244*, *246*, *253, 254*, *414*, *518*, *520*
unique.igraph.vs, 520
unique.igraph.vs(), *64, 65*, *151, 152*, *238, 239*, *244*, *246*, *253, 254*, *414*, *518, 519*
upgrade_graph, 520
upgrade_graph(), *218, 219*

V, 521
V(), *46*, *167*, *236*, *238–240*, *242*, *244*, *246*, *397, 398*
V<- *(*igraph-vs-attributes*)*, 241
vcount *(*gorder*)*, 196
vcount(), *176*
vctrs::vec_c(), *154*
vertex, 523
vertex(), *16–18*, *20*, *114*, *119*, *121*, *126*, *141*, *143*, *149, 150*, *155*, *170*, *240, 241*, *251, 252*, *373*, *375*, *413*, *415*, *483*, *508*, *516, 517*
vertex.attributes *(*vertex_attr*)*, 525
vertex.attributes<- *(*vertex_attr<-*)*, 526
vertex.shape.pie, 524
vertex.shape.pie(), *377*
vertex_attr, 525
vertex_attr(), *140*, *142, 143*, *171–173*, *198, 199*, *233, 234*, *242*, *472–475*, *526, 527*
vertex_attr<-, 526

vertex_attr_names, 526
vertex_attr_names(), *140*, *142, 143*, *171–173*, *198, 199*, *233, 234*, *242*, *472–475*, *525*, *527*
vertex_connectivity, 527
vertex_connectivity(), *32*, *59*, *164*, *175*, *257*, *269*, *272*, *352*, *359, 360*, *363*, *487*, *489*
vertex_disjoint_paths *(*vertex_connectivity*)*, 527
vertices *(*vertex*)*, 523
vertices(), *16*, *240*
voronoi_cells, 529
voronoi_cells(), *47*, *85*, *87*, *89, 90*, *92*, *95*, *97*, *99*, *101*, *104*, *106*, *113*, *222*, *329*, *356*, *365*, *388*, *486*

weighted_clique_num *(*cliques*)*, 79
weighted_cliques, 531
weighted_cliques(), *81*, *260*, *280*
wheel *(*make_wheel*)*, 347
which_loop *(*which_multiple*)*, 532
which_loop(), *483*
which_multiple, 532
which_multiple(), *57*, *116*, *122*, *125*, *129*, *139*, *146*, *159*, *176*, *185, 186*, *193*, *256*, *261*, *268*, *282*, *284*, *412*, *483*, *492*, *494*, *507*, *511*, *515*, *535*
which_mutual, 534
which_mutual(), *57*, *116*, *122*, *125*, *129*, *139*, *146*, *159*, *176*, *185, 186*, *193*, *256*, *261*, *268*, *282*, *284*, *412*, *492*, *494*, *507*, *511*, *515*, *533*
with_dh *(*layout_with_dh*)*, 301
with_drl *(*layout_with_drl*)*, 303
with_edge_, 535
with_edge_(), *324*, *419*, *482*, *536*, *538, 539*
with_fr *(*layout_with_fr*)*, 306
with_gem *(*layout_with_gem*)*, 308
with_graph_, 536
with_graph_(), *324*, *419*, *482*, *536*, *538, 539*
with_graphopt *(*layout_with_graphopt*)*, 310
with_igraph_opt, 537
with_igraph_opt(), *248*
with_kk *(*layout_with_kk*)*, 311
with_lgl *(*layout_with_lgl*)*, 314
with_mds *(*layout_with_mds*)*, 315
with_sugiyama *(*layout_with_sugiyama*)*, 317
with_vertex_, 537
with_vertex_(), *324*, *419*, *482*, *536*, *538, 539*
without_attr, 538
without_attr(), *324*, *419*, *482*, *536*, *538, 539*
without_loops, 539
without_loops(), *324*, *419*, *482*, *536*, *538*, *539*
without_multiples, 539
without_multiples(), *324*, *419*, *482*, *536*, *538, 539*
write_graph, 540
write_graph(), *110*, *212*, *376*, *408*