igraph Reference Manual

Gábor Csárdi, Department of Statistics, Harvard University
Tamás Nepusz, Department of Biological Physics, Eötvös Loránd University
Vincent Traag, Centre for Science and Technology Studies, Leiden University
Szabolcs Horvát, Department of Computer Science, Reykjavik University
Fabio Zanini, Lowy Cancer Research Centre, University of New South Wales
Daniel Noom, jitjit software development

igraph Reference Manual

by Gábor Csárdi, Tamás Nepusz, Vincent Traag, Szabolcs Horvát, Fabio Zanini, and Daniel Noom

1.0.0-rc1-114-g133f73958

This manual is for igraph, version 1.0.0-rc1-114-g133f73958.

Table of Contents

1. Introduction 1
    igraph is free software 1
    Citing igraph 2
2. Installation 3
    Prerequisites 3
    Installation 3
        General build instructions 3
        Specific instructions for Windows 4
        Notable configuration options 5
    Building the documentation 6
    Notes for package maintainers 6
        Auto-detection of dependencies 6
        Shared and static builds 7
        Cross-compiling 7
        Additional notes 8
3. Tutorial 9
    Compiling programs using igraph 9
        Compiling with CMake 10
        Compiling without CMake 10
        Running the program 11
    Creating your first graphs 11
    Calculating various properties of graphs 13
4. Basic data types and interface 15
    The igraph data model 15
    General conventions of igraph functions 16
    Atomic data types 16
    Setup and initialization 16
        `igraph_setup` — Initializes the igraph library. 17
    The basic interface 17
        Graph constructors and destructors 17
        Basic query operations 19
        Adding and deleting vertices and edges 27
    Miscellaneous macros and helper functions 31
        `IGRAPH_VCOUNT_MAX` — The maximum number of vertices supported in igraph graphs. 31
        `IGRAPH_ECOUNT_MAX` — The maximum number of edges supported in igraph graphs. 31
        `IGRAPH_UNLIMITED` — Constant for "do not limit results". 31
        `igraph_expand_path_to_pairs` — Helper function to convert a sequence of vertex IDs describing a path into a "pairs" vector. 31
        `igraph_invalidate_cache` — Invalidates the internal cache of an igraph graph. 32
        `igraph_is_same_graph` — Are two graphs identical as labelled graphs? 32
5. Error handling 34
    Error handling basics 34
    Error handlers 34
        `igraph_error_handler_t` — The type of error handler functions. 34
        `igraph_error_handler_abort` — Abort program in case of error. 35
        `igraph_error_handler_ignore` — Ignore errors. 35
        `igraph_error_handler_printignore` — Print and ignore errors. 35
    Error codes 35
        `igraph_error_t` — Return type for functions returning an error code. 35
        `igraph_error_type_t` — Error code type. 35
        `igraph_strerror` — Textual description of an error. 37
    Warning messages 37

igraph_warning_handler_t — The type of igraph warning handler functions. 37
igraph_set_warning_handler — Installs a warning handler. 38
IGRAPH_WARNING — Triggers a warning. 38
IGRAPH_WARNINGF — Triggers a warning, with printf-like syntax. 38
igraph_warning — Reports a warning. 38
igraph_warningf — Reports a warning, printf-like version. 39
igraph_warning_handler_ignore — Ignores all warnings. 39
igraph_warning_handler_print — Prints all warnings to the standard error. 39
Advanced topics 40
Writing error handlers 40
Error handling internals 40
Deallocating memory 43
Writing **igraph** functions with proper error handling 44
Fatal errors 45
Error handling and threads 47
6. Memory (de)allocation 48
About allocation functions 48
Available allocation functions 48
igraph_malloc — Allocates memory that can be safely deallocated by igraph functions. 48
igraph_calloc — Allocates memory that can be safely deallocated by igraph functions. 48
igraph_realloc — Reallocate memory that can be safely deallocated by igraph functions. 49
igraph_free — Deallocates memory that was allocated by igraph functions. 49
7. Data structure library: vector, matrix, other data types 50
About template types 50
Vectors 51
About igraph_vector_t objects 51
Constructors and destructors 51
Initializing elements 53
Accessing elements 54
Vector views 57
Copying vectors 58
Exchanging elements 59
Vector operations 61
Vector comparisons 64
Finding minimum and maximum 69
Vector properties 71
Searching for elements 74
Resizing operations 76
Complex vector operations 80
Sorting 82
Set operations on sorted vectors 83
Pointer vectors (igraph_vector_ptr_t) 86
Matrices 94
About igraph_matrix_t objects 94
Matrix constructors and destructors 94
Initializing elements 95
Accessing elements of a matrix 96
Matrix views 97
Copying matrices 98
Operations on rows and columns 99
Matrix operations 103
Matrix comparisons 107
Combining matrices 109

Finding minimum and maximum 110
Matrix properties 112
Searching for elements 115
Resizing operations 116
Complex matrix operations 118
Sparse matrices 121
About sparse matrices 121
Creating sparse matrix objects 121
Query properties of a sparse matrix 124
Operations on sparse matrices 130
Operations on sparse matrix iterators 136
Operations that change the internal representation 139
Decompositions and solving linear systems 139
Eigenvalues and eigenvectors 145
Conversion to other data types 147
Writing to a file, or to the screen 148
Stacks 148
igraph_stack_init — Initializes a stack. 148
igraph_stack_destroy — Destroys a stack object. 148
igraph_stack_reserve — Reserve memory. 149
igraph_stack_empty — Decides whether a stack object is empty. 149
igraph_stack_size — Returns the number of elements in a stack. 149
igraph_stack_clear — Removes all elements from a stack. 150
igraph_stack_push — Places an element on the top of a stack. 150
igraph_stack_pop — Removes and returns an element from the top of a stack. 150
igraph_stack_top — Query top element. 151
Double-ended queues 151
igraph_dqueue_init — Initialize a double ended queue (deque). 151
igraph_dqueue_destroy — Destroy a double ended queue. 151
igraph_dqueue_empty — Decide whether the queue is empty. 152
igraph_dqueue_full — Check whether the queue is full. 152
igraph_dqueue_clear — Remove all elements from the queue. 152
igraph_dqueue_size — Number of elements in the queue. 152
igraph_dqueue_head — Head of the queue. 153
igraph_dqueue_back — Tail of the queue. 153
igraph_dqueue_get — Access an element in a queue. 153
igraph_dqueue_pop — Remove the head. 154
igraph_dqueue_pop_back — Removes the tail. 154
igraph_dqueue_push — Appends an element. 154
Maximum and minimum heaps 155
igraph_heap_init — Initializes an empty heap object. 155
igraph_heap_init_array — Build a heap from an array. 155
igraph_heap_destroy — Destroys an initialized heap object. 155
igraph_heap_clear — Removes all elements from a heap. 156
igraph_heap_empty — Decides whether a heap object is empty. 156
igraph_heap_push — Add an element. 156
igraph_heap_top — Top element. 157
igraph_heap_delete_top — Removes and returns the top element. 157
igraph_heap_size — Number of elements in the heap. 157
igraph_heap_reserve — Reserves memory for a heap. 158
String vectors 158
igraph_strvector_init — Initializes a string vector. 158
igraph_strvector_init_copy — Initialization by copying. 158
igraph_strvector_destroy — Frees the memory allocated for the string vector. 159
STR — Indexing string vectors. 159
igraph_strvector_get — Retrieves an element of a string vector. 160

igraph_strvector_set — Sets an element of the string vector from a string. 160
igraph_strvector_set_len — Sets an element of the string vector given a buffer and its size. 160
igraph_strvector_push_back — Adds an element to the back of a string vector. 161
igraph_strvector_push_back_len — Adds a string of the given length to the back of a string vector. 161
igraph_strvector_swap_elements — Swap two elements in a string vector. 161
igraph_strvector_remove — Removes a single element from a string vector. 162
igraph_strvector_remove_section — Removes a section from a string vector. 162
igraph_strvector_append — Concatenates two string vectors. 162
igraph_strvector_merge — Moves the contents of a string vector to the end of another. 163
igraph_strvector_swap — Swaps all elements of two string vectors. 163
igraph_strvector_update — Updates a string vector from another one. 164
igraph_strvector_clear — Removes all elements from a string vector. 164
igraph_strvector_resize — Resizes a string vector. 164
igraph_strvector_reserve — Reserves memory for a string vector. 165
igraph_strvector_resize_min — Deallocates the unused memory of a string vector. 165
igraph_strvector_size — Returns the size of a string vector. 165
igraph_strvector_capacity — Returns the capacity of a string vector. 166
Lists of vectors, matrices and graphs 166
About igraph_vector_list_t objects 166
Constructors and destructors 167
Accessing elements 168
Vector properties 170
Resizing operations 171
Sorting and reordering 177
Adjacency lists 179
Adjacent vertices 180
Incident edges 184
Lazy adjacency list for vertices 186
Lazy incidence list for edges 188
Partial prefix sum trees 190
igraph_psumtree_init — Initializes a partial prefix sum tree. 191
igraph_psumtree_destroy — Destroys a partial prefix sum tree. 191
igraph_psumtree_size — Returns the size of the tree. 191
igraph_psumtree_get — Retrieves the value corresponding to an item in the tree. 192
igraph_psumtree_sum — Returns the sum of the values of the leaves in the tree. 192
igraph_psumtree_search — Finds an item in the tree, given a value. 192
igraph_psumtree_update — Updates the value associated to an item in the tree. 193
igraph_psumtree_reset — Resets all the values in the tree to zero. 193
Bitsets 194
About igraph_bitset_t objects 194
Constructors and destructors 194
Accessing elements 195
Bitset operations 198
Bitset properties 203
Resizing operations 203
Copying bitsets 204

8. Random numbers 206
About random numbers in igraph 206
The default random number generator 206
igraph_rng_default — Query the default random number generator. 206
igraph_rng_set_default — Set the default igraph random number generator. 206
Creating random number generators 206
igraph_rng_init — Initializes a random number generator. 206
igraph_rng_destroy — Deallocates memory associated with a random number generator. 207
igraph_rng_seed — Seeds a random number generator. 207
igraph_rng_bits — The number of random bits that a random number generator can produces in a single round. 207
igraph_rng_max — The maximum possible integer for a random number generator. 208
igraph_rng_name — The type of a random number generator. 208
Generating random numbers 208
igraph_rng_get_bool — Generate a random boolean. 208
igraph_rng_get_integer — Generate an integer random number from an interval. 209
igraph_rng_get_unif01 — Samples uniformly from the unit interval. 209
igraph_rng_get_unif — Samples real numbers from a given interval. 210
igraph_rng_get_normal — Samples from a normal distribution. 210
igraph_rng_get_exp — Samples from an exponential distribution. 210
igraph_rng_get_gamma — Samples from a gamma distribution. 211
igraph_rng_get_binom — Samples from a binomial distribution. 211
igraph_rng_get_geom — Samples from a geometric distribution. 212
igraph_rng_get_pois — Samples from a Poisson distribution. 212
Supported random number generators 212
igraph_rngtype_mt19937 — The MT19937 random number generator. 213
igraph_rngtype_glibc2 — The random number generator introduced in GNU libc 2. 213
igraph_rngtype_pcg32 — The PCG random number generator (32-bit version). 214
igraph_rngtype_pcg64 — The PCG random number generator (64-bit version). 214
Use cases 215
Normal (default) use 215
Reproducible simulations 215
Changing the default generator 215
Using multiple generators 215
Example 215
9. Vertex and edge selectors and sequences, iterators 216
About selectors, iterators 216
Vertex selector constructors 216
igraph_vs_all — Vertex set, all vertices of a graph. 216
igraph_vs_adj — Adjacent vertices of a vertex. 217
igraph_vs_nonadj — Non-adjacent vertices of a vertex. 217
igraph_vs_none — Empty vertex set. 218
igraph_vs_1 — Vertex set with a single vertex. 219
igraph_vs_vector — Vertex set based on a vector. 219
igraph_vs_vector_small — Create a vertex set by giving its elements. 220
igraph_vs_vector_copy — Vertex set based on a vector, with copying. 220
igraph_vs_range — Vertex set, an interval of vertices. 221
Generic vertex selector operations 221
igraph_vs_copy — Creates a copy of a vertex selector. 221
igraph_vs_destroy — Destroy a vertex set. 221
igraph_vs_is_all — Check whether all vertices are included. 222

igraph_vs_size — Returns the size of the vertex selector. 222
igraph_vs_type — Returns the type of the vertex selector. 222
Immediate vertex selectors 223
igraph_vss_all — All vertices of a graph (immediate version). 223
igraph_vss_none — Empty vertex set (immediate version). 223
igraph_vss_1 — Vertex set with a single vertex (immediate version). 223
igraph_vss_vector — Vertex set based on a vector (immediate version). 224
igraph_vss_range — An interval of vertices (immediate version). 224
Vertex iterators 225
igraph_vit_create — Creates a vertex iterator from a vertex selector. 225
igraph_vit_destroy — Destroys a vertex iterator. 225
Stepping over the vertices 226
IGRAPH_VIT_NEXT — Next vertex. 226
IGRAPH_VIT_END — Are we at the end? 226
IGRAPH_VIT_SIZE — Size of a vertex iterator. 226
IGRAPH_VIT_RESET — Reset a vertex iterator. 227
IGRAPH_VIT_GET — Query the current position. 227
Edge selector constructors 227
igraph_es_all — Edge set, all edges. 227
igraph_es_incident — Edges incident on a given vertex. 228
igraph_es_none — Empty edge selector. 229
igraph_es_1 — Edge selector containing a single edge. 229
igraph_es_all_between — Edge selector, all edge IDs between a pair of vertices. 229
igraph_es_vector — Handle a vector as an edge selector. 230
igraph_es_range — Edge selector, a sequence of edge IDs. 230
igraph_es_pairs — Edge selector, multiple edges defined by their endpoints in a vector. 231
igraph_es_pairs_small — Edge selector, multiple edges defined by their endpoints as arguments. 231
igraph_es_path — Edge selector, edge IDs on a path. 232
igraph_es_vector_copy — Edge set, based on a vector, with copying. 233
Immediate edge selectors 233
igraph_ess_all — Edge set, all edges (immediate version). 233
igraph_ess_none — Immediate empty edge selector. 233
igraph_ess_1 — Immediate version of the single edge edge selector. 234
igraph_ess_vector — Immediate vector view edge selector. 234
igraph_ess_range — Immediate version of the sequence edge selector. 235
Generic edge selector operations 235
igraph_es_as_vector — Transform edge selector into vector. 235
igraph_es_copy — Creates a copy of an edge selector. 235
igraph_es_destroy — Destroys an edge selector object. 236
igraph_es_is_all — Check whether an edge selector includes all edges. 236
igraph_es_size — Returns the size of the edge selector. 236
igraph_es_type — Returns the type of the edge selector. 237
Edge iterators 237
igraph_eit_create — Creates an edge iterator from an edge selector. 237
igraph_eit_destroy — Destroys an edge iterator. 238
Stepping over the edges 238
IGRAPH_EIT_NEXT — Next edge. 238
IGRAPH_EIT_END — Are we at the end? 238
IGRAPH_EIT_SIZE — Number of edges in the iterator. 238
IGRAPH_EIT_RESET — Reset an edge iterator. 239
IGRAPH_EIT_GET — Query an edge iterator. 239
10. Graph, vertex and edge attributes 240
The attribute handler interface 240
igraph_attribute_table_t — Table of functions to perform operations on attributes. 240

igraph_set_attribute_table — Attach an attribute table. 244
igraph_attribute_type_t — The possible types of the attributes. 244
igraph_attribute_elemtype_t — Types of objects to which attributes can be attached. 245
Attribute records 245
igraph_attribute_record_t — An attribute record holding the name, type and values of an attribute. 246
igraph_attribute_record_init — Initializes an attribute record with a given name and type. 246
igraph_attribute_record_init_copy — Initializes an attribute record by copying another record. 246
igraph_attribute_record_size — Returns the size of the value vector in an attribute record. 247
igraph_attribute_record_resize — Resizes the value vector in an attribute record. 247
igraph_attribute_record_set_name — Sets the attribute name in an attribute record. 247
igraph_attribute_record_set_type — Sets the type of an attribute record. 248
igraph_attribute_record_set_default_numeric — Sets the default value of the attribute to the given number. 248
igraph_attribute_record_set_default_string — Sets the default value of the attribute to the given string. 249
igraph_attribute_record_set_default_boolean — Sets the default value of the attribute to the given logical value. 249
igraph_attribute_record_destroy — Destroys an attribute record. 249
Handling attribute combination lists 250
igraph_attribute_combination_init — Initialize attribute combination list. 250
igraph_attribute_combination_add — Add combination record to attribute combination list. 250
igraph_attribute_combination_remove — Remove a record from an attribute combination list. 251
igraph_attribute_combination_destroy — Destroy attribute combination list. 251
igraph_attribute_combination_type_t — The possible types of attribute combinations. 251
igraph_attribute_combination — Initialize attribute combination list and add records. 252
Accessing attributes from C 253
Query attributes 254
Set attributes 266
Remove attributes 278
Custom attribute combination functions 281
11. Deterministic graph generators 282
About generators 282
Basic graph creation 282
igraph_create — Creates a graph with the specified edges. 282
igraph_small — Shorthand to create a small graph, giving the edges as arguments. 282
Graphs from adjacency matrices and adjacency lists 283
igraph_adjacency — Creates a graph from an adjacency matrix. 283
igraph_weighted_adjacency — Creates a graph from a weighted adjacency matrix. 284
igraph_sparse_adjacency — Creates a graph from a sparse adjacency matrix. 286
igraph_sparse_weighted_adjacency — Creates a graph from a weighted sparse adjacency matrix. 286

igraph_adjlist — Creates a graph from an adjacency list. 286
Regular structures 287
igraph_star — Creates a *star* graph, every vertex connects only to the center. .. 287
igraph_wheel — Creates a *wheel* graph, a union of a star and a cycle graph. 288
igraph_hypercube — The n-dimensional hypercube graph. 288
igraph_square_lattice — Arbitrary dimensional square lattices. 289
igraph_triangular_lattice — A triangular lattice with the given shape. .. 290
igraph_hexagonal_lattice — A hexagonal lattice with the given shape. ... 291
igraph_ring — Creates a *cycle* graph or a *path* graph. 291
igraph_path_graph — A path graph P_n. 292
igraph_cycle_graph — A cycle graph C_n. 293
igraph_lcf — Creates a graph from LCF notation. 293
igraph_lcf_small — Shorthand to create a graph from LCF notation, giving shifts as the arguments. 294
igraph_circulant — Creates a circulant graph. 294
igraph_extended_chordal_ring — Create an extended chordal ring. 295
Tree generators 295
igraph_kary_tree — Creates a k-ary tree in which almost all vertices have k children. 296
igraph_symmetric_tree — Creates a symmetric tree with the specified number of branches at each level. 296
igraph_regular_tree — Creates a regular tree. 297
igraph_tree_from_parent_vector — Constructs a tree or forest from a vector encoding the parent of each vertex. 298
igraph_from_prufer — Generates a tree from a Prüfer sequence. 299
Graphs with given degrees 299
igraph_realize_degree_sequence — Generates a graph with the given degree sequence. 299
igraph_realize_bipartite_degree_sequence — Generates a bipartite graph with the given bidegree sequence. 302
Complete graphs 303
igraph_full — Creates a full graph (complete graph). 303
igraph_full_citation — Creates a full citation graph (a complete directed acyclic graph). 304
igraph_full_multipartite — Creates a full multipartite graph. 304
igraph_turan — Creates a Turán graph. 305
Pre-defined graphs 306
igraph_famous — Create a famous graph by simply providing its name. 306
igraph_atlas — Create a small graph from the “Graph Atlas”. 308
Other well-known graphs from graph theory 309
igraph_de_bruijn — Generate a de Bruijn graph. 309
igraph_kautz — Generate a Kautz graph. 309
igraph_generalized_petersen — Creates a Generalized Petersen graph. 310
igraph_mycielski_graph — The Mycielski graph of order *k*. 311
12. Stochastic graph generators ("games") 312
The Erd#s-Rényi and related models 312
igraph_erdos_renyi_game_gnm — Generates a random (Erd#s-Rényi) graph with a fixed number of edges. 312
igraph_erdos_renyi_game_gnp — Generates a random (Erd#s-Rényi) graph with fixed edge probabilities. 313
igraph_iea_game — Generates a random multigraph through independent edge assignment. 314
igraph_sbm_game — Sample from a stochastic block model. 315
igraph_hsbm_game — Hierarchical stochastic block model. 316
igraph_hsbm_list_game — Hierarchical stochastic block model, more general version. 317

igraph_preference_game — Generates a graph with vertex types and connection preferences. 318
igraph_asymmetric_preference_game — Generates a graph with asymmetric vertex types and connection preferences. 319
igraph_correlated_game — Generates a random graph correlated to an existing graph. 320
igraph_correlated_pair_game — Generates pairs of correlated random graphs. 320
Preferential attachment and related models 321
igraph_barabasi_game — Generates a graph based on the Barabási-Albert model. 321
igraph_barabasi_aging_game — Preferential attachment with aging of vertices. 323
igraph_recent_degree_game — Stochastic graph generator based on the number of incident edges a node has gained recently. 324
igraph_recent_degree_aging_game — Preferential attachment based on the number of edges gained recently, with aging of vertices. 325
igraph_lastcit_game — Simulates a citation network, based on time passed since the last citation. 326
Growing random graph models 327
igraph_growing_random_game — Generates a growing random graph. 327
igraph_callaway_traits_game — Simulates a growing network with vertex types. 328
igraph_establishment_game — Generates a graph with a simple growing model with vertex types. 328
igraph_cited_type_game — Simulates a citation based on vertex types. 329
igraph_citing_cited_type_game — Simulates a citation network based on vertex types. 330
igraph_forest_fire_game — Generates a network according to the “forest fire game”. 331
Degree-constrained models 332
igraph_degree_sequence_game — Generates a random graph with a given degree sequence. 332
igraph_k_regular_game — Generates a random graph where each vertex has the same degree. 334
igraph_rewire — Randomly rewires a graph while preserving its degree sequence. 335
igraph_chung_lu_game — Samples graphs from the Chung-Lu model. 335
igraph_static_fitness_game — Non-growing random graph with edge probabilities proportional to node fitness scores. 337
igraph_static_power_law_game — Generates a non-growing random graph with expected power-law degree distributions. 339
Edge rewiring models 340
igraph_watts_strogatz_game — The Watts-Strogatz small-world model. 340
igraph_rewire_edges — Rewires the edges of a graph with constant probability. 341
igraph_rewire_directed_edges — Rewires the chosen endpoint of directed edges. 341
Other random graphs 342
igraph_grg_game — Generates a geometric random graph. 342
igraph_dot_product_game — Generates a random dot product graph. 343
igraph_simple_interconnected_islands_game — Generates a random graph made of several interconnected islands, each island being a random graph. 343
igraph_tree_game — Generates a random tree with the given number of nodes. 344
Common types and constants 345

igraph_edge_type_sw_t — What types of non-simple edges to allow? 345
13. Bipartite, i.e. two-mode graphs 346
Bipartite networks in igraph 346
Create two-mode networks 346
igraph_create_bipartite — Create a bipartite graph. 346
igraph_full_bipartite — Creates a complete bipartite graph. 346
igraph_bipartite_game_gnm — Generate a random bipartite graph with a fixed number of edges. 347
igraph_bipartite_game_gnp — Generates a random bipartite graph with a fixed connection probability. 348
igraph_bipartite_iea_game — Generates a random bipartite multigraph through independent edge assignment. 350
Bipartite adjacency matrices 351
igraph_biadjacency — Creates a bipartite graph from a bipartite adjacency matrix. 351
igraph_weighted_biadjacency — Creates a bipartite graph from a weighted bipartite adjacency matrix. 352
igraph_get_biadjacency — Converts a bipartite graph into a bipartite adjacency matrix. 352
Project two-mode graphs 353
igraph_bipartite_projection_size — Calculate the number of vertices and edges in the bipartite projections. 353
igraph_bipartite_projection — Create one or both projections of a bipartite (two-mode) network. 354
Other operations on bipartite graphs 355
igraph_is_bipartite — Check whether a graph is bipartite. 355
14. Spatial graphs 357
Metrics 357
igraph_metric_t — Metric functions for use with spatial computation. 357
Spatial graph generators 357
igraph_delaunay_graph — Computes the Delaunay graph of a spatial point set. 357
igraph_nearest_neighbor_graph — Computes the nearest neighbor graph for a spatial point set. 358
igraph_gabriel_graph — The Gabriel graph of a point set. 358
igraph_relative_neighborhood_graph — The relative neighborhood graph of a point set. 359
igraph_lune_beta_skeleton — The lune based β-skeleton of a spatial point set. 360
igraph_circle_beta_skeleton — The circle based β-skeleton of a 2D spatial point set. 360
igraph_beta_weighted_gabriel_graph — A Gabriel graph, with edges weighted by the β value at which it disappears. 361
Properties of spatial graphs 362
igraph_spatial_edge_lengths — Edge lengths based on spatial vertex coordinates. 362
Non-graph related spatial processing 363
igraph_convex_hull_2d — Determines the convex hull of a given set of points in the 2D plane. 363
15. Graph operators 364
Union and intersection 364
igraph_disjoint_union — Creates the union of two disjoint graphs. 364
igraph_disjoint_union_many — The disjoint union of many graphs. 364
igraph_join — Creates the join of two disjoint graphs. 365
igraph_union — Calculates the union of two graphs. 366
igraph_union_many — Creates the union of many graphs. 366
igraph_intersection — Collect the common edges from two graphs. 367
igraph_intersection_many — The intersection of more than two graphs. .. 368

Other set-like operators 369
igraph_difference — Calculates the difference of two graphs. 369
igraph_complementer — Creates the complementer of a graph. 369
igraph_compose — Calculates the composition of two graphs. 370
Miscellaneous operators 371
igraph_connect_neighborhood — Connects each vertex to its neighborhood. 371
igraph_contract_vertices — Replace multiple vertices with a single one. 371
igraph_graph_power — The k-th power of a graph. 372
igraph_product — The graph product of two graphs, according to the chosen product type. 373
igraph_rooted_product — The rooted graph product of two graphs. 375
igraph_induced_subgraph — Creates a subgraph induced by the specified vertices. 375
igraph_induced_subgraph_map — Creates an induced subraph and returns the mapping from the original. 376
igraph_induced_subgraph_edges — The edges contained within an induced sugraph. 377
igraph_linegraph — Create the line graph of a graph. 378
igraph_mycielskian — Generate the Mycielskian of a graph with *k* iterations. 378
igraph_simplify — Removes loop and/or multiple edges from the graph. 379
igraph_subgraph_from_edges — Creates a subgraph with the specified edges and their endpoints. 380
igraph_reverse_edges — Reverses some edges of a directed graph. 381
16. Graph visitors 382
Breadth-first search 382
igraph_bfs — Breadth-first search. 382
igraph_bfs_simple — Breadth-first search, single-source version 383
igraph_bfshandler_t — Callback type for BFS function. 384
Depth-first search 385
igraph_dfs — Depth-first search. 385
igraph_dfshandler_t — Callback type for the DFS function. 386
Random walks 387
igraph_random_walk — Performs a random walk on a graph. 387
17. Structural properties of graphs 388
Basic properties 388
igraph_are_adjacent — Decides whether two vertices are adjacent. 388
Sparsifiers 388
igraph_spanner — Calculates a spanner of a graph with a given stretch factor. 388
(Shortest)-path related functions 389
igraph_distances — Length of the shortest paths between vertices. 389
igraph_distances_cutoff — Length of the shortest paths between vertices, with cutoff. 390
igraph_distances_dijkstra — Weighted shortest path lengths between vertices. 391
igraph_distances_dijkstra_cutoff — Weighted shortest path lengths between vertices, with cutoff. 392
igraph_distances_bellman_ford — Weighted shortest path lengths between vertices, allowing negative weights. 393
igraph_distances_johnson — Weighted shortest path lengths between vertices, using Johnson's algorithm. 394
igraph_distances_floyd_warshall — Weighted all-pairs shortest path lengths with the Floyd-Warshall algorithm. 395
igraph_get_shortest_paths — Shortest paths from a vertex. 396

igraph_get_shortest_path — Shortest path from one vertex to another one. 398
igraph_get_shortest_paths_dijkstra — Weighted shortest paths from a vertex. 399
igraph_get_shortest_path_dijkstra — Weighted shortest path from one vertex to another one (Dijkstra). 400
igraph_get_shortest_paths_bellman_ford — Weighted shortest paths from a vertex, allowing negative weights. 401
igraph_get_shortest_path_bellman_ford — Weighted shortest path from one vertex to another one (Bellman-Ford). 403
igraph_get_shortest_path_astar — A* gives the shortest path from one vertex to another, with heuristic. 404
igraph_astar_heuristic_func_t — Distance estimator for A* algorithm. 405
igraph_get_all_shortest_paths — All shortest paths (geodesics) from a vertex. 405
igraph_get_all_shortest_paths_dijkstra — All weighted shortest paths (geodesics) from a vertex. 407
igraph_get_k_shortest_paths — k shortest paths between two vertices. 408
igraph_get_all_simple_paths — List all simple paths from one source. .. 409
igraph_average_path_length — The average shortest path length between all vertex pairs. 410
igraph_path_length_hist — Create a histogram of all shortest path lengths. 411
igraph_diameter — Calculates the weighted diameter of a graph using Dijkstra's algorithm. 411
igraph_girth — The girth of a graph is the length of the shortest cycle in it. 412
igraph_eccentricity — Eccentricity of some vertices. 413
igraph_radius — Radius of a graph, using weighted edges. 414
igraph_graph_center — Central vertices of a graph. 414
igraph_pseudo_diameter — Approximation and lower bound of the diameter of a graph. 415
igraph_voronoi — Voronoi partitioning of a graph. 416
igraph_vertex_path_from_edge_path — Converts a walk of edge IDs to the traversed vertex IDs. 417
Widest-path related functions 418
igraph_get_widest_path — Widest path from one vertex to another one. ... 418
igraph_get_widest_paths — Widest paths from a single vertex. 419
igraph_widest_path_widths_dijkstra — Widths of widest paths between vertices. 420
igraph_widest_path_widths_floyd_warshall — Widths of widest paths between vertices. 421
Efficiency measures 422
igraph_global_efficiency — Calculates the global efficiency of a network. 422
igraph_local_efficiency — Calculates the local efficiency around each vertex in a network. 423
igraph_average_local_efficiency — Calculates the average local efficiency in a network. 424
Neighborhood of a vertex 425
igraph_neighborhood_size — Calculates the size of the neighborhood of a given vertex. 425
igraph_neighborhood — Calculate the neighborhood of vertices. 426
igraph_neighborhood_graphs — Create graphs from the neighborhood(s) of some vertex/vertices. 427
Local scan statistics 428
"Us" statistics 428

"Them" statistics 429
Pre-calculated subsets 431
Graph components 432
igraph_subcomponent — The vertices reachable from a given vertex. 432
igraph_connected_components — Calculates the (weakly or strongly) connected components in a graph. 433
igraph_is_connected — Decides whether the graph is (weakly or strongly) connected. 433
igraph_decompose — Decomposes a graph into connected components. 434
igraph_reachability — Calculates which vertices are reachable from each vertex in the graph. 435
igraph_count_reachable — The number of vertices reachable from each vertex in the graph. 436
igraph_transitive_closure — Computes the transitive closure of a graph. 436
igraph_biconnected_components — Calculates biconnected components. 437
igraph_articulation_points — Finds the articulation points in a graph. .. 438
igraph_bridges — Finds all bridges in a graph. 439
igraph_is_biconnected — Checks whether a graph is biconnected. 439
Percolation 440
igraph_site_percolation — The size of the largest component as vertices are added to a graph. 440
igraph_bond_percolation — The size of the largest component as edges are added to a graph. 440
igraph_edgelist_percolation — The size of the largest component as vertex pairs are connected. 441
Degree sequences 442
igraph_is_graphical — Is there a graph with the given degree sequence? ... 442
igraph_is_bigraphical — Is there a bipartite graph with the given bi-degree-sequence? 443
Centrality measures 444
igraph_closeness — Closeness centrality calculations for some vertices. 444
igraph_harmonic_centrality — Harmonic centrality for some vertices. ... 446
igraph_betweenness — Betweenness centrality of some vertices. 447
igraph_edge_betweenness — Betweenness centrality of the edges. 448
igraph_pagerank_algo_t — PageRank algorithm implementation. 449
igraph_pagerank — Calculates the Google PageRank for the specified vertices. 449
igraph_personalized_pagerank — Calculates the personalized Google PageRank for the specified vertices. 451
igraph_personalized_pagerank_vs — Calculates the personalized Google PageRank for the specified vertices. 452
igraph_constraint — Burt's constraint scores. 453
igraph_maxdegree — The maximum degree in a graph (or set of vertices). 454
igraph_strength — Strength of the vertices, also called weighted vertex degree. 455
igraph_eigenvector_centrality — Eigenvector centrality of the vertices. 455
igraph_hub_and_authority_scores — Kleinberg's hub and authority scores (HITS). 457
igraph_convergence_degree — Calculates the convergence degree of each edge in a graph. 458
Range-limited centrality measures 459
igraph_closeness_cutoff — Range limited closeness centrality. 459
igraph_harmonic_centrality_cutoff — Range limited harmonic centrality. 460
igraph_betweenness_cutoff — Range-limited betweenness centrality. 461

igraph_edge_betweenness_cutoff — Range-limited betweenness centrality of the edges. 462
Subset-limited centrality measures 463
igraph_betweenness_subset — Betweenness centrality for a subset of source and target vertices. 463
igraph_edge_betweenness_subset — Edge betweenness centrality for a subset of source and target vertices. 464
Centralization 465
igraph_centralization — Calculate the centralization score from the node level scores. 465
igraph_centralization_degree — Calculate vertex degree and graph centralization. 466
igraph_centralization_betweenness — Calculate vertex betweenness and graph centralization. 467
igraph_centralization_closeness — Calculate vertex closeness and graph centralization. 468
igraph_centralization_eigenvector_centrality — Calculate eigenvector centrality scores and graph centralization. 468
igraph_centralization_degree_tmax — Theoretical maximum for graph centralization based on degree. 469
igraph_centralization_betweenness_tmax — Theoretical maximum for graph centralization based on betweenness. 470
igraph_centralization_closeness_tmax — Theoretical maximum for graph centralization based on closeness. 471
igraph_centralization_eigenvector_centrality_tmax — Theoretical maximum centralization for eigenvector centrality. 472
Similarity measures 473
igraph_bibcoupling — Bibliographic coupling. 473
igraph_cocitation — Cocitation coupling. 473
igraph_similarity_jaccard — Jaccard similarity coefficient for the given vertices. 474
igraph_similarity_jaccard_pairs — Jaccard similarity coefficient for given vertex pairs. 475
igraph_similarity_jaccard_es — Jaccard similarity coefficient for a given edge selector. 476
igraph_similarity_dice — Dice similarity coefficient. 476
igraph_similarity_dice_pairs — Dice similarity coefficient for given vertex pairs. 477
igraph_similarity_dice_es — Dice similarity coefficient for a given edge selector. 478
igraph_similarity_inverse_log_weighted — Vertex similarity based on the inverse logarithm of vertex degrees. 479
Trees and forests 480
igraph_minimum_spanning_tree — Calculates a minimum spanning tree of a graph. 480
igraph_random_spanning_tree — Uniformly samples the spanning trees of a graph. 481
igraph_is_tree — Decides whether the graph is a tree. 482
igraph_is_forest — Decides whether the graph is a forest. 483
igraph_to_prufer — Converts a tree to its Prüfer sequence. 483
Transitivity or clustering coefficient 484
igraph_transitivity_undirected — Calculates the transitivity (clustering coefficient) of a graph. 484
igraph_transitivity_local_undirected — The local transitivity (clustering coefficient) of some vertices. 485
igraph_transitivity_avglocal_undirected — Average local transitivity (clustering coefficient). 486

igraph_transitivity_barrat — Weighted local transitivity of some vertices, as defined by A. Barrat. ... 486
igraph_ecc — Edge clustering coefficient of some edges. ... 487
Directedness conversion ... 488
igraph_to_directed — Convert an undirected graph to a directed one. ... 488
igraph_to_undirected — Convert a directed graph to an undirected one. ... 489
Spectral properties ... 489
igraph_get_laplacian — Returns the Laplacian matrix of a graph. ... 489
igraph_get_laplacian_sparse — Returns the Laplacian of a graph in a sparse matrix format. ... 490
igraph_laplacian_normalization_t — Normalization methods for a Laplacian matrix. ... 491
Non-simple graphs: Multiple and loop edges ... 491
igraph_is_simple — Decides whether the input graph is a simple graph. ... 491
igraph_is_loop — Find the loop edges in a graph. ... 492
igraph_has_loop — Returns whether the graph has at least one loop edge. ... 492
igraph_count_loops — Counts the self-loops in the graph. ... 493
igraph_is_multiple — Find the multiple edges in a graph. ... 493
igraph_has_multiple — Check whether the graph has at least one multiple edge. ... 494
igraph_count_multiple — The multiplicity of some edges in a graph. ... 495
igraph_count_multiple_1 — The multiplicity of a single edge in a graph. ... 495
Mixing patterns and degree correlations ... 496
igraph_assortativity_nominal — Assortativity of a graph based on vertex categories. ... 496
igraph_assortativity — Assortativity based on numeric properties of vertices. ... 497
igraph_assortativity_degree — Assortativity of a graph based on vertex degree. ... 498
igraph_avg_nearest_neighbor_degree — Average neighbor degree. ... 499
igraph_degree_correlation_vector — Degree correlation function. ... 500
igraph_joint_type_distribution — Mixing matrix for vertex categories. ... 502
igraph_joint_degree_distribution — The joint degree distribution of a graph. ... 503
igraph_joint_degree_matrix — The joint degree matrix of a graph. ... 504
igraph_rich_club_sequence — Density sequence of subgraphs formed by sequential vertex removal. ... 505
K-cores and k-trusses ... 507
igraph_coreness — The coreness of the vertices in a graph. ... 507
igraph_trussness — Finding the "trussness" of the edges in a network. ... 507
Maximum cardinality search and chordal graphs ... 508
igraph_maximum_cardinality_search — Maximum cardinality search. ... 508
igraph_is_chordal — Decides whether a graph is chordal. ... 509
Matchings ... 509
igraph_is_matching — Checks whether the given matching is valid for the given graph. ... 509
igraph_is_maximal_matching — Checks whether a matching in a graph is maximal. ... 510
igraph_maximum_bipartite_matching — Calculates a maximum matching in a bipartite graph. ... 511
Unfolding a graph into a tree ... 512
igraph_unfold_tree — Unfolding a graph into a tree, by possibly multiplicating its vertices. ... 512
Other operations ... 512
igraph_density — Calculate the density of a graph. ... 512
igraph_mean_degree — The mean degree of a graph. ... 513

igraph_reciprocity — Calculates the reciprocity of a directed graph. 514
igraph_diversity — Structural diversity index of the vertices. 514
igraph_is_mutual — Check whether some edges of a directed graph are mutual. 515
igraph_has_mutual — Check whether a directed graph has any mutual edges. 516
igraph_get_adjacency — The adjacency matrix of a graph. 516
igraph_get_adjacency_sparse — Returns the adjacency matrix of a graph in a sparse matrix format. 517
igraph_get_stochastic — Stochastic adjacency matrix of a graph. 518
igraph_get_stochastic_sparse — The stochastic adjacency matrix of a graph. 519
igraph_get_edgelist — The list of edges in a graph. 519
Common types and constants 520
igraph_loops_t — How to interpret self-loops in undirected graphs? 520
igraph_neimode_t — How to interpret edge directions in directed graphs? 521
18. Graph cycles 522
Finding cycles 522
igraph_find_cycle — Finds a single cycle in the graph. 522
igraph_simple_cycles — Finds all simple cycles. 522
igraph_simple_cycles_callback — Finds all simple cycles (callback version). 523
igraph_cycle_handler_t — Type of cycle handler functions. 524
Acyclic graphs and feedback sets 525
igraph_is_dag — Checks whether a graph is a directed acyclic graph (DAG). 525
igraph_is_acyclic — Checks whether a graph is acyclic or not. 525
igraph_topological_sorting — Calculate a possible topological sorting of the graph. 526
igraph_feedback_arc_set — Feedback arc set of a graph using exact or heuristic methods. 527
igraph_feedback_vertex_set — Feedback vertex set of a graph. 528
Eulerian cycles and paths 529
igraph_is_eulerian — Checks whether an Eulerian path or cycle exists. 529
igraph_eulerian_cycle — Finds an Eulerian cycle. 529
igraph_eulerian_path — Finds an Eulerian path. 530
Cycle bases 530
igraph_fundamental_cycles — Finds a fundamental cycle basis. 530
igraph_minimum_cycle_basis — Computes a minimum weight cycle basis. 531
19. Cliques and independent vertex sets 533
Cliques 533
igraph_is_complete — Decides whether the graph is complete. 533
igraph_is_clique — Does a set of vertices form a clique? 533
igraph_cliques — Finds all or some cliques in a graph. 534
igraph_clique_size_hist — Counts cliques of each size in the graph. 534
igraph_cliques_callback — Calls a function for each clique in the graph. 535
igraph_clique_handler_t — Type of clique handler functions. 536
igraph_largest_cliques — Finds the largest clique(s) in a graph. 536
igraph_maximal_cliques — Finds all maximal cliques in a graph. 537
igraph_maximal_cliques_count — Count the number of maximal cliques in a graph. 538
igraph_maximal_cliques_file — Find maximal cliques and write them to a file. 539
igraph_maximal_cliques_subset — Maximal cliques for a subset of initial vertices. 539

igraph_maximal_cliques_hist — Counts the number of maximal cliques of each size in a graph. 540
igraph_maximal_cliques_callback — Finds maximal cliques in a graph and calls a function for each one. 541
igraph_clique_number — Finds the clique number of the graph. 541
Weighted cliques 542
igraph_weighted_cliques — Finds all cliques in a given weight range in a vertex weighted graph. 542
igraph_largest_weighted_cliques — Finds the largest weight clique(s) in a graph. 543
igraph_weighted_clique_number — Finds the weight of the largest weight clique in the graph. 543
Independent vertex sets 544
igraph_is_independent_vertex_set — Does a set of vertices form an independent set? 544
igraph_independent_vertex_sets — Finds all independent vertex sets in a graph. 545
igraph_largest_independent_vertex_sets — Finds the largest independent vertex set(s) in a graph. 546
igraph_maximal_independent_vertex_sets — Finds all maximal independent vertex sets of a graph. 546
igraph_independence_number — Finds the independence number of the graph. 547
20. Graph motifs, dyad census and triad census 548
igraph_dyad_census — Dyad census, as defined by Holland and Leinhardt. 548
igraph_triad_census — Triad census, as defined by Davis and Leinhardt. 548
Finding triangles 550
igraph_count_adjacent_triangles — Count the number of triangles a vertex is part of. 550
igraph_count_triangles — Counts triangles in a graph. 550
igraph_list_triangles — Find all triangles in a graph. 551
Graph motifs 551
igraph_motifs_randesu — Count the number of motifs in a graph. 551
igraph_motifs_randesu_no — Count the total number of motifs in a graph. 552
igraph_motifs_randesu_estimate — Estimate the total number of motifs in a graph. 553
igraph_motifs_randesu_callback — Finds motifs in a graph and calls a function for each of them. 554
igraph_motifs_handler_t — Callback type for igraph_motifs_randesu_callback. 554
21. Graph isomorphism 556
The simple interface 556
igraph_isomorphic — Are two graphs isomorphic? 556
igraph_subisomorphic — Decide subgraph isomorphism. 557
igraph_count_automorphisms — Number of automorphisms of a graph. 557
igraph_automorphism_group — Automorphism group generators of a graph. 558
igraph_canonical_permutation — Canonical permutation of a graph. 558
The BLISS algorithm 559
igraph_bliss_sh_t — Splitting heuristics for Bliss. 559
igraph_bliss_info_t — Information about a Bliss run. 560
igraph_isomorphic_bliss — Graph isomorphism via Bliss. 560
igraph_count_automorphisms_bliss — Number of automorphisms using Bliss. 561
igraph_automorphism_group_bliss — Automorphism group generators using Bliss. 562

igraph_canonical_permutation_bliss — Canonical permutation using Bliss. ... 563
The VF2 algorithm ... 563
igraph_isomorphic_vf2 — Isomorphism via VF2. ... 564
igraph_count_isomorphisms_vf2 — Number of isomorphisms via VF2. ... 565
igraph_get_isomorphisms_vf2 — Collect all isomorphic mappings of two graphs. ... 566
igraph_get_isomorphisms_vf2_callback — The generic VF2 interface ... 567
igraph_isohandler_t — Callback type, called when an isomorphism was found ... 568
igraph_isocompat_t — Callback type, called to check whether two vertices or edges are compatible ... 568
igraph_subisomorphic_vf2 — Decide subgraph isomorphism using VF2 .. 569
igraph_count_subisomorphisms_vf2 — Number of subgraph isomorphisms using VF2 ... 570
igraph_get_subisomorphisms_vf2 — Return all subgraph isomorphic mappings. ... 571
igraph_get_subisomorphisms_vf2_callback — Generic VF2 function for subgraph isomorphism problems. ... 572
The LAD algorithm ... 573
igraph_subisomorphic_lad — Check subgraph isomorphism with the LAD algorithm ... 573
Functions for small graphs ... 574
igraph_isoclass — Determine the isomorphism class of small graphs. ... 574
igraph_isoclass_subgraph — The isomorphism class of a subgraph of a graph. ... 575
igraph_isoclass_create — Creates a graph from the given isomorphism class. ... 576
igraph_graph_count — The number of unlabelled graphs on the given number of vertices. ... 576
Utility functions ... 577
igraph_invert_permutation — Inverts a permutation. ... 577
igraph_permute_vertices — Permute the vertices. ... 577
igraph_simplify_and_colorize — Simplify the graph and compute self-loop and edge multiplicities. ... 578
22. Graph coloring ... 579
igraph_vertex_coloring_greedy — Computes a vertex coloring using a greedy algorithm. ... 579
igraph_coloring_greedy_t — Ordering heuristics for greedy graph coloring. ... 579
igraph_is_vertex_coloring — Checks whether a vertex coloring is valid. ... 580
igraph_is_bipartite_coloring — Checks whether a bipartite vertex coloring is valid. ... 580
igraph_is_edge_coloring — Checks whether an edge coloring is valid. ... 581
igraph_is_perfect — Checks if the graph is perfect. ... 581
23. Maximum flows, minimum cuts and related measures ... 583
Maximum flows ... 583
igraph_maxflow — Maximum network flow between a pair of vertices. ... 583
igraph_maxflow_value — Maximum flow in a network with the push/relabel algorithm. ... 584
igraph_dominator_tree — Calculates the dominator tree of a flowgraph. ... 585
igraph_maxflow_stats_t — Data structure holding statistics from the push-relabel maximum flow solver. ... 586
Cuts and minimum cuts ... 586
igraph_st_mincut — Minimum cut between a source and a target vertex. ... 586
igraph_st_mincut_value — The minimum s-t cut in a graph. ... 587

igraph_all_st_cuts — List all edge-cuts between two vertices in a directed graph 588
igraph_all_st_mincuts — All minimum s-t cuts of a directed graph. 589
igraph_mincut — Calculates the minimum cut in a graph. 589
igraph_mincut_value — The minimum edge cut in a graph. 590
igraph_gomory_hu_tree — Gomory-Hu tree of a graph. 591
Connectivity 592
igraph_st_edge_connectivity — Edge connectivity of a pair of vertices. 592
igraph_edge_connectivity — The minimum edge connectivity in a graph. 593
igraph_st_vertex_connectivity — The vertex connectivity of a pair of vertices. 593
igraph_vertex_connectivity — The vertex connectivity of a graph. 594
Edge- and vertex-disjoint paths 595
igraph_edge_disjoint_paths — The maximum number of edge-disjoint paths between two vertices. 595
igraph_vertex_disjoint_paths — Maximum number of vertex-disjoint paths between two vertices. 596
Graph adhesion and cohesion 596
igraph_adhesion — Graph adhesion, this is (almost) the same as edge connectivity. 596
igraph_cohesion — Graph cohesion, this is the same as vertex connectivity. .. 597
Cohesive blocks 598
igraph_cohesive_blocks — Identifies the hierarchical cohesive block structure of a graph. 598
24. Vertex separators 599
igraph_is_separator — Would removing this set of vertices disconnect the graph? 599
igraph_is_minimal_separator — Decides whether a set of vertices is a minimal separator. 599
igraph_all_minimal_st_separators — List all vertex sets that are minimal (s,t) separators for some s and t. 600
igraph_minimum_size_separators — Find all minimum size separating vertex sets. 600
igraph_even_tarjan_reduction — Even-Tarjan reduction of a graph. 601
25. Detecting community structure 603
Common functions related to community structure 603
igraph_modularity — Calculates the modularity of a graph with respect to some clusters or vertex types. 603
igraph_modularity_matrix — Calculates the modularity matrix. 604
igraph_community_optimal_modularity — Calculate the community structure with the highest modularity value. 605
igraph_community_to_membership — Cut a dendrogram after a given number of merges. 606
igraph_reindex_membership — Makes the IDs in a membership vector contiguous. 607
igraph_compare_communities — Compares community structures using various metrics. 608
igraph_split_join_distance — Calculates the split-join distance of two community structures. 609
Community structure based on statistical mechanics 610
igraph_community_spinglass — Community detection based on statistical mechanics. 610
igraph_community_spinglass_single — Community of a single node based on statistical mechanics. 612
Community structure based on eigenvectors of matrices 613

igraph_community_leading_eigenvector — Leading eigenvector community finding (proper version). 614
igraph_community_leading_eigenvector_callback_t — Callback for the leading eigenvector community finding method. 616
igraph_le_community_to_membership — Cut an incomplete dendrogram after a given number of merges, starting with an initial cluster assignment. 617
Walktrap: Community structure based on random walks 618
igraph_community_walktrap — Community finding using a random walk based similarity measure. 618
Edge betweenness based community detection 619
igraph_community_edge_betweenness — Community finding based on edge betweenness. 619
igraph_community_eb_get_merges — Calculating the merges, i.e. the dendrogram for an edge betweenness community structure. 621
Community structure based on the optimization of modularity 622
igraph_community_fastgreedy — Finding community structure by greedy optimization of modularity. 622
igraph_community_multilevel — Finding community structure by multi-level optimization of modularity (Louvain). 623
igraph_community_leiden — Finding community structure using the Leiden algorithm. 624
igraph_community_leiden_simple — Finding community structure using the Leiden algorithm, simple interface. 626
Fluid communities 628
igraph_community_fluid_communities — Community detection based on fluids interacting on the graph. 628
Label propagation 628
igraph_community_label_propagation — Community detection based on label propagation. 628
The InfoMAP algorithm 630
igraph_community_infomap — Community structure that minimizes the expected description length of a random walker trajectory. 630
Voronoi communities 632
igraph_community_voronoi — Finds communities using Voronoi partitioning. 632
26. Graphlets 634
Introduction 634
Performing graphlet decomposition 634
igraph_graphlets — Calculate graphlets basis and project the graph on it. 634
igraph_graphlets_candidate_basis — Calculate a candidate graphlets basis 635
igraph_graphlets_project — Project a graph on a graphlets basis. 635
27. Hierarchical random graphs 637
Introduction 637
Representing HRGs 637
igraph_hrg_t — Data structure to store a hierarchical random graph. 637
igraph_hrg_init — Allocate memory for a HRG. 638
igraph_hrg_destroy — Deallocate memory for an HRG. 638
igraph_hrg_size — Returns the size of the HRG, the number of leaf nodes. ... 638
igraph_hrg_resize — Resize a HRG. 639
Fitting HRGs 639
igraph_hrg_fit — Fit a hierarchical random graph model to a network. 639
igraph_hrg_consensus — Calculate a consensus tree for a HRG. 639
HRG sampling 640
igraph_hrg_sample — Sample from a hierarchical random graph model. 640
igraph_hrg_game — Generate a hierarchical random graph. 641
Conversion to and from igraph graphs 641

igraph_from_hrg_dendrogram — Create a graph representation of the dendrogram of a hierarchical random graph model. 641
igraph_hrg_create — Create a HRG from an igraph graph. 641
Predicting missing edges 642
igraph_hrg_predict — Predict missing edges in a graph, based on HRG models. 642
Deprecated functions 643
igraph_hrg_dendrogram — Create a dendrogram from a hierarchical random graph. 643
28. Embedding of graphs 644
Spectral embedding 644
igraph_adjacency_spectral_embedding — Adjacency spectral embedding 644
igraph_laplacian_spectral_embedding — Spectral embedding of the Laplacian of a graph 645
igraph_dim_select — Dimensionality selection. 646
29. Generating layouts for graph drawing 648
2D layout generators 648
igraph_layout_random — Places the vertices uniformly randomly within a square. 648
igraph_layout_circle — Places the vertices uniformly on a circle in arbitrary order. 648
igraph_layout_star — Generates a star-like layout. 649
igraph_layout_grid — Places the vertices on a regular grid on the plane. 649
igraph_layout_graphopt — Optimizes vertex layout via the graphopt algorithm. 650
igraph_layout_bipartite — Simple layout for bipartite graphs. 651
The DrL layout generator 651
igraph_layout_fruchterman_reingold — Places the vertices on a plane according to the Fruchterman-Reingold algorithm. 655
igraph_layout_kamada_kawai — Places the vertices on a plane according to the Kamada-Kawai algorithm. 656
igraph_layout_gem — Layout graph according to GEM algorithm. 658
igraph_layout_davidson_harel — Davidson-Harel layout algorithm. 658
igraph_layout_mds — Place the vertices on a plane using multidimensional scaling. 660
igraph_layout_lgl — Force based layout algorithm for large graphs. 660
Layouts for trees and acyclic graphs 661
igraph_layout_reingold_tilford — Reingold-Tilford layout for tree graphs. 661
igraph_layout_reingold_tilford_circular — Circular Reingold-Tilford layout for trees. 662
igraph_roots_for_tree_layout — Roots suitable for a nice tree layout. .. 663
igraph_layout_sugiyama — Sugiyama layout algorithm for layered directed acyclic graphs. 664
igraph_layout_umap — Layout using Uniform Manifold Approximation and Projection (UMAP). 665
igraph_layout_umap_compute_weights — Compute weights for a UMAP layout starting from distances. 666
3D layout generators 668
igraph_layout_random_3d — Places the vertices uniformly randomly in a cube. 668
igraph_layout_sphere — Places vertices (more or less) uniformly on a sphere. 668
igraph_layout_grid_3d — Places the vertices on a regular grid in the 3D space. 669
igraph_layout_fruchterman_reingold_3d — 3D Fruchterman-Reingold algorithm. 669

igraph_layout_kamada_kawai_3d — 3D version of the Kamada-Kawai layout generator. ... 670
igraph_layout_umap_3d — 3D layout using UMAP. ... 671
Post-processing layouts ... 672
igraph_layout_merge_dla — Merges multiple layouts by using a DLA algorithm. ... 672
igraph_layout_align — Aligns a graph layout with the coordinate axes. ... 673
30. Processes on graphs ... 674
Epidemic models ... 674
igraph_sir — Performs a number of SIR epidemics model runs on a graph. ... 674
igraph_sir_t — The result of one SIR model simulation. ... 674
igraph_sir_destroy — Deallocates memory associated with a SIR simulation run. ... 675
31. Reading and writing graphs from and to files ... 676
Simple edge list and similar formats ... 676
igraph_read_graph_edgelist — Reads an edge list from a file and creates a graph. ... 676
igraph_write_graph_edgelist — Writes the edge list of a graph to a file. ... 676
igraph_read_graph_ncol — Reads an .ncol file used by LGL. ... 677
igraph_write_graph_ncol — Writes the graph to a file in .ncol format. ... 678
igraph_read_graph_lgl — Reads a graph from an .lgl file. ... 679
igraph_write_graph_lgl — Writes the graph to a file in .lgl format. ... 680
igraph_read_graph_dimacs_flow — Read a graph in DIMACS format. ... 680
igraph_write_graph_dimacs_flow — Write a graph in DIMACS format. ... 681
Binary formats ... 682
igraph_read_graph_graphdb — Read a graph in the binary graph database format. ... 682
GraphML format ... 683
igraph_read_graph_graphml — Reads a graph from a GraphML file. ... 683
igraph_write_graph_graphml — Writes the graph to a file in GraphML format. ... 684
GML format ... 684
igraph_read_graph_gml — Read a graph in GML format. ... 684
igraph_write_graph_gml — Write the graph to a stream in GML format. ... 685
Pajek format ... 686
igraph_read_graph_pajek — Reads a file in Pajek format. ... 686
igraph_write_graph_pajek — Writes a graph to a file in Pajek format. ... 688
UCINET's DL file format ... 689
igraph_read_graph_dl — Reads a file in the DL format of UCINET. ... 689
Graphviz format ... 689
igraph_write_graph_dot — Write the graph to a stream in DOT format. ... 689
LEDA format ... 690
igraph_write_graph_leda — Write a graph in LEDA native graph format. ... 690
Convenience functions for locale change ... 691
igraph_enter_safelocale — Temporarily set the C locale. ... 691
igraph_exit_safelocale — Temporarily set the C locale. ... 691
32. Using BLAS, LAPACK and ARPACK for igraph matrices and graphs ... 692
BLAS interface in igraph ... 692
igraph_blas_ddot — Dot product of two vectors. ... 692
igraph_blas_dnrm2 — Euclidean norm of a vector. ... 692
igraph_blas_dgemv — Matrix-vector multiplication using BLAS, vector version. ... 693
igraph_blas_dgemm — Matrix-matrix multiplication using BLAS. ... 693
igraph_blas_dgemv_array — Matrix-vector multiplication using BLAS, array version. ... 694

LAPACK interface in igraph 694
Matrix factorization, solving linear systems 695
Eigenvalues and eigenvectors of matrices 696
ARPACK interface in igraph 701
Data structures 701
ARPACK solvers 707
33. Non-graph related functions 710
igraph version number 710
igraph_version — The version of the igraph C library. 710
Running mean of a time series 710
igraph_running_mean — Calculates the running mean of a vector. 710
Random sampling from very long sequences 711
igraph_random_sample — Generates an increasing random sequence of integers. 711
Random sampling of spatial points 711
igraph_rng_sample_sphere_surface — Sample points uniformly from the surface of a sphere. 711
igraph_rng_sample_sphere_volume — Sample points uniformly from the volume of a sphere. 712
igraph_rng_sample_dirichlet — Sample points from a Dirichlet distribution. 713
Fitting power-law distributions to empirical data 713
igraph_plfit_result_t — Result of fitting a power-law distribution to a vector. 713
igraph_power_law_fit — Fits a power-law distribution to a vector of numbers. 714
igraph_plfit_result_calculate_p_value — Calculates the p-value of a fitted power-law model. 715
Comparing floats with a tolerance 716
igraph_cmp_epsilon — Compare two double-precision floats with a tolerance. 716
igraph_almost_equals — Compare two double-precision floats with a tolerance. 717
igraph_complex_almost_equals — Compare two complex numbers with a tolerance. 717
34. Advanced igraph programming 718
Using igraph in multi-threaded programs 718
IGRAPH_THREAD_SAFE — Specifies whether igraph was built in thread-safe mode. 718
Thread-safe ARPACK library 718
Thread-safety of random number generators 718
Progress handlers 718
About progress handlers 718
Setting up progress handlers 719
Invoking the progress handler 720
Writing progress handlers 722
Writing igraph functions with progress reporting 722
Multi-threaded programs 722
Status handlers 722
Status reporting 722
Setting up status handlers 723
Invoking the status handler 724
35. Glossary 726
36. Licenses for igraph and this manual 727
THE GNU GENERAL PUBLIC LICENSE 727
Preamble 727
GNU GENERAL PUBLIC LICENSE 727
How to Apply These Terms to Your New Programs 730

The GNU Free Documentation License 731
0. PREAMBLE 731
1. APPLICABILITY AND DEFINITIONS 732
2. VERBATIM COPYING 733
3. COPYING IN QUANTITY 733
4. MODIFICATIONS 733
5. COMBINING DOCUMENTS 735
6. COLLECTIONS OF DOCUMENTS 735
7. AGGREGATION WITH INDEPENDENT WORKS 735
8. TRANSLATION 735
9. TERMINATION 736
10. FUTURE REVISIONS OF THIS LICENSE 736
G.1.1 ADDENDUM: How to use this License for your documents 736
Index 737

List of Examples

4.1. File examples/simple/creation.c 18
4.2. File examples/simple/igraph_copy.c 19
4.3. File examples/simple/igraph_is_directed.c 20
4.4. File examples/simple/igraph_get_eid.c 23
4.5. File examples/simple/igraph_get_eids.c 24
4.6. File examples/simple/igraph_neighbors.c 25
4.7. File examples/simple/igraph_degree.c 27
4.8. File examples/simple/creation.c 28
4.9. File examples/simple/creation.c 29
4.10. File examples/simple/igraph_delete_edges.c 29
4.11. File examples/simple/igraph_delete_vertices.c 30
5.1. File examples/simple/igraph_contract_vertices.c 44
7.1. File examples/simple/igraph_fisher_yates_shuffle.c 61
7.2. File examples/simple/igraph_vector_int_list_sort.c 67
7.3. File examples/simple/igraph_vector_int_list_sort.c 68
7.4. File examples/simple/igraph_sparsemat.c 121
7.5. File examples/simple/igraph_sparsemat3.c 121
7.6. File examples/simple/igraph_sparsemat4.c 121
7.7. File examples/simple/igraph_sparsemat6.c 121
7.8. File examples/simple/igraph_sparsemat7.c 121
7.9. File examples/simple/igraph_sparsemat8.c 121
7.10. File examples/simple/dqueue.c 151
7.11. File examples/simple/igraph_strvector.c 158
7.12. File examples/simple/adjlist.c 180
8.1. File examples/simple/random_seed.c 215
9.1. File examples/simple/igraph_vs_nonadj.c 218
9.2. File examples/simple/igraph_vs_vector.c 219
9.3. File examples/simple/igraph_vs_range.c 221
9.4. File examples/simple/igraph_es_pairs.c 231
10.1. File examples/simple/igraph_attribute_combination.c 253
10.2. File examples/simple/cattributes.c 253
10.3. File examples/simple/cattributes2.c 254
10.4. File examples/simple/cattributes3.c 254
10.5. File examples/simple/cattributes4.c 254
11.1. File examples/simple/igraph_create.c 282
11.2. File examples/simple/igraph_small.c 283
11.3. File examples/simple/igraph_weighted_adjacency.c 285
11.4. File examples/simple/igraph_star.c 287
11.5. File examples/simple/igraph_ring.c 292
11.6. File examples/simple/igraph_lcf.c 294
11.7. File examples/simple/igraph_kary_tree.c 296
11.8. File examples/simple/igraph_symmetric_tree.c 297
11.9. File examples/simple/igraph_regular_tree.c 298
11.10. File examples/simple/igraph_realize_degree_sequence.c 302
11.11. File examples/simple/igraph_full.c 304
11.12. File examples/simple/igraph_atlas.c 309
12.1. File examples/simple/igraph_erdos_renyi_game_gnm.c 313
12.2. File examples/simple/igraph_erdos_renyi_game_gnp.c 314
12.3. File examples/simple/igraph_barabasi_game.c 323
12.4. File examples/simple/igraph_barabasi_game2.c 323
12.5. File examples/simple/igraph_degree_sequence_game.c 334
12.6. File examples/simple/igraph_grg_game.c 343
13.1. File examples/simple/igraph_bipartite_create.c 346
15.1. File examples/simple/igraph_disjoint_union.c 364
15.2. File examples/simple/igraph_union.c 366

15.3. File examples/simple/igraph_intersection.c 368
15.4. File examples/simple/igraph_difference.c 369
15.5. File examples/simple/igraph_complementer.c 370
15.6. File examples/simple/igraph_compose.c 371
15.7. File examples/simple/igraph_contract_vertices.c 372
15.8. File examples/simple/igraph_simplify.c 380
16.1. File examples/simple/igraph_bfs.c 383
16.2. File examples/simple/igraph_bfs_callback.c 383
16.3. File examples/simple/igraph_bfs_simple.c 384
17.1. File examples/simple/distances.c 390
17.2. File examples/simple/distances.c 391
17.3. File examples/simple/distances.c 392
17.4. File examples/simple/distances.c 393
17.5. File examples/simple/bellman_ford.c 394
17.6. File examples/simple/igraph_get_shortest_paths.c 398
17.7. File examples/simple/igraph_get_shortest_paths_dijkstra.c 400
17.8. File examples/simple/igraph_get_all_shortest_paths_dijkstra.c . 408
17.9. File examples/simple/igraph_grg_game.c 411
17.10. File examples/simple/igraph_diameter.c 412
17.11. File examples/simple/igraph_girth.c 413
17.12. File examples/simple/igraph_eccentricity.c 414
17.13. File examples/simple/igraph_contract_vertices.c 433
17.14. File examples/simple/igraph_decompose.c 435
17.15. File examples/simple/igraph_biconnected_components.c 438
17.16. File examples/simple/igraph_is_biconnected.c 439
17.17. File examples/simple/igraph_pagerank.c 451
17.18. File examples/simple/eigenvector_centrality.c 457
17.19. File examples/simple/centralization.c 466
17.20. File examples/simple/igraph_cocitation.c 473
17.21. File examples/simple/igraph_cocitation.c 474
17.22. File examples/simple/igraph_similarity.c 475
17.23. File examples/simple/igraph_similarity.c 476
17.24. File examples/simple/igraph_similarity.c 476
17.25. File examples/simple/igraph_similarity.c 477
17.26. File examples/simple/igraph_similarity.c 478
17.27. File examples/simple/igraph_similarity.c 479
17.28. File examples/simple/igraph_similarity.c 480
17.29. File examples/simple/igraph_minimum_spanning_tree.c 481
17.30. File examples/simple/igraph_kary_tree.c 483
17.31. File examples/simple/igraph_transitivity.c 485
17.32. File examples/simple/igraph_to_undirected.c 489
17.33. File examples/simple/igraph_get_laplacian.c 490
17.34. File examples/simple/igraph_get_laplacian_sparse.c 491
17.35. File examples/simple/igraph_is_loop.c 492
17.36. File examples/simple/igraph_is_loop.c 493
17.37. File examples/simple/igraph_is_loop.c 493
17.38. File examples/simple/igraph_is_multiple.c 494
17.39. File examples/simple/igraph_has_multiple.c 495
17.40. File examples/simple/igraph_assortativity_nominal.c 497
17.41. File examples/simple/igraph_assortativity_degree.c 499
17.42. File examples/simple/igraph_avg_nearest_neighbor_degree.c 500
17.43. File examples/simple/igraph_maximum_bipartite_matching.c 510
17.44. File examples/simple/igraph_maximum_bipartite_matching.c 511
17.45. File examples/simple/igraph_maximum_bipartite_matching.c 512
17.46. File examples/simple/igraph_reciprocity.c 514
18.1. File examples/simple/igraph_topological_sorting.c 527
18.2. File examples/simple/igraph_feedback_arc_set.c 528
18.3. File examples/simple/igraph_feedback_arc_set_ip.c 528

19.1. File examples/simple/igraph_cliques.c 534
19.2. File examples/simple/igraph_maximal_cliques.c 538
19.3. File examples/simple/igraph_maximal_cliques.c 538
19.4. File examples/simple/igraph_independent_sets.c 545
20.1. File examples/simple/igraph_list_triangles.c 551
20.2. File examples/simple/igraph_motifs_randesu.c 552
20.3. File examples/simple/igraph_motifs_randesu.c 554
21.1. File examples/simple/igraph_isomorphic_vf2.c 565
21.2. File examples/simple/igraph_subisomorphic_lad.c 574
22.1. File examples/simple/coloring.c 579
22.2. File examples/simple/coloring.c 580
23.1. File examples/simple/flow.c 584
23.2. File examples/simple/flow2.c 584
23.3. File examples/simple/dominator_tree.c 586
23.4. File examples/simple/igraph_all_st_mincuts.c 589
23.5. File examples/simple/igraph_mincut.c 590
23.6. File examples/simple/cohesive_blocks.c 598
24.1. File examples/simple/igraph_is_separator.c 599
24.2. File examples/simple/igraph_is_minimal_separator.c 600
24.3. File examples/simple/igraph_minimal_separators.c 600
24.4. File examples/simple/igraph_minimum_size_separators.c 601
24.5. File examples/simple/even_tarjan.c 602
25.1. File examples/simple/igraph_community_optimal_modularity.c 606
25.2. File examples/simple/igraph_community_leading_eigenvector.c 614
25.3. File examples/simple/walktrap.c 619
25.4. File examples/simple/igraph_community_edge_betweenness.c 620
25.5. File examples/simple/igraph_community_fastgreedy.c 623
25.6. File examples/simple/igraph_community_multilevel.c 624
25.7. File examples/simple/igraph_community_leiden.c 626
25.8. File examples/simple/igraph_community_label_propagation.c 630
29.1. File examples/simple/igraph_layout_reingold_tilford.c 662
31.1. File examples/simple/igraph_read_graph_lgl.c 680
31.2. File examples/simple/igraph_write_graph_lgl.c 680
31.3. File examples/simple/igraph_read_graph_graphdb.c 683
31.4. File examples/simple/graphml.c 684
31.5. File examples/simple/graphml.c 684
31.6. File examples/simple/gml.c 685
31.7. File examples/simple/gml.c 686
31.8. File examples/simple/foreign.c 688
31.9. File examples/simple/igraph_write_graph_pajek.c 689
31.10. File examples/simple/igraph_read_graph_dl.c 689
31.11. File examples/simple/dot.c 690
31.12. File examples/simple/safelocale.c 691
32.1. File examples/simple/blas.c 692
32.2. File examples/simple/blas.c 693
32.3. File examples/simple/blas_dgemm.c 694
32.4. File examples/simple/igraph_lapack_dgesv.c 696
32.5. File examples/simple/igraph_lapack_dsyevr.c 698
32.6. File examples/simple/igraph_lapack_dgeev.c 699
32.7. File examples/simple/igraph_lapack_dgeevx.c 701
33.1. File examples/simple/igraph_version.c 710
33.2. File examples/simple/igraph_random_sample.c 711
33.3. File examples/simple/igraph_power_law_fit.c 715

Chapter 1. Introduction

igraph is a library for creating and manipulating graphs. You can look at it in two ways: first, igraph contains the implementation of quite a lot of graph algorithms. These include classic graph algorithms like graph isomorphism, graph girth and connectivity and also the new wave graph algorithms like transitivity, graph motifs and community structure detection. Skim through the table of contents or the index of this book to get an impression of what is available.

Second, igraph provides a platform for developing and/or implementing graph algorithms. It has an efficient data structure for representing graphs, and a number of other data structures like flexible vectors, stacks, heaps, queues, adjacency lists that are useful for implementing graph algorithms. In fact these data structures evolved along with the implementation of the classic and non-classic graph algorithms which make up the major part of the igraph library. This way, they were fine-tuned and checked for correctness several times.

Our main goal with developing igraph was to create a graph library which is efficient on large, but not extremely large graphs. More precisely, it is assumed that the graph(s) fit into the physical memory of the computer. Nowadays this means graphs with several million vertices and/or edges. Our definition of efficient is that it runs fast, both in theory and (more importantly) in practice.

We believe that one of the big strengths of igraph is that it can be embedded into a higher-level language or environment. Three such embeddings (or interfaces if you look at them another way) are currently being developed by us: an R package, a Python extension module, and a Mathematica (Wolfram Language) package. Others are likely to come. High level languages such as R or Python make it possible to use graph routines with much greater comfort, without actually writing a single line of C code. They have some, usually very small, speed penalty compared to the C version, but add ease of use and much flexibility. This manual, however, covers only the C library. If you want to use Python, R or the Wolfram Language, please see the documentation written specifically for these interfaces and come back here only if you are interested in some detail which is not covered in those documents.

We still consider igraph as a child project. It has much room for development and we are sure that it will improve a lot in the near future. Any feedback we can get from the users is very important for us, as most of the time these questions and comments guide us in what to add and what to improve.

igraph is open source and distributed under the terms of the GNU GPL version 2 or (at your option) any later version. We strongly believe that all the algorithms used in science, let that be graph theory or not, should have an efficient open-source implementation allowing use and modification for anyone.

igraph is free software

Citing igraph

To cite igraph in publications, please use the following reference:

Gábor Csárdi, Tamás Nepusz: The igraph software package for complex network research. InterJournal Complex Systems, 1695, 2006.

The igraph C library is assigned the DOI 10.5281/zenodo.3630268 [https://doi.org/10.5281/zenodo.3630268] on Zenodo.

Chapter 2. Installation

This chapter describes building igraph from source code and installing it. The source archive of the latest stable release is always available from the igraph website [https://igraph.org/c/#downloads]. igraph is also included in many Linux distributions, as well as several package managers such as vcpkg [https://vcpkg.io/] (convenient on Windows), MacPorts [https://www.macports.org/] (macOS) and Homebrew [https://brew.sh/] (macOS), which provide an easier means of installation. If you decide to use them, please consult their documentation on how to install packages.

Prerequisites

To build igraph from sources, you will need at least:

- CMake [https://cmake.org] 3.18 or later
- C and C++ compilers

Visual Studio 2015 and later are supported. Earlier Visual Studio versions may or may not work.

Certain features also require the following libraries:

- libxml2 [http://www.xmlsoft.org/], required for GraphML support

igraph bundles a number of libraries for convenience. However, it is preferable to use external versions of these libraries, which may improve performance. These are:

- GMP [https://gmplib.org/] (the bundled alternative is Mini-GMP)
- GLPK [https://www.gnu.org/software/glpk/] (version 4.57 or later)
- ARPACK [https://github.com/opencollab/arpack-ng]
- plfit [https://github.com/ntamas/plfit]
- A library providing a BLAS [https://www.netlib.org/blas/] API (available by default on macOS; OpenBLAS [http://www.openmathlib.org/OpenBLAS/] is one option on other systems)
- A library providing a LAPACK [https://www.netlib.org/lapack/] API (available by default on macOS; OpenBLAS [http://www.openmathlib.org/OpenBLAS/] is one option on other systems)

When building the development version of igraph, `bison`, `flex` and `git` are also required. Released versions do not require these tools.

To run the tests, `diff` is also required.

Installation

General build instructions

igraph uses a CMake-based build system [https://cmake.org/cmake/help/latest/guide/user-interaction/index.html]. To compile it,

- Enter the directory where the igraph sources are:

```
$ cd igraph
```

- Create a new directory. This is where igraph will be built:

```
$ mkdir build
$ cd build
```

- Run CMake, which will automatically configure igraph, and report the configuration:

```
$ cmake ..
```

To set a non-default installation location, such as `/opt/local`, use:

```
cmake .. -DCMAKE_INSTALL_PREFIX=/opt/local
```

- Check the output carefully, and ensure that all features you need are enabled. If CMake could not find certain libraries, some features such as GraphML support may have been automatically disabled.
- There are several ways to adjust the configuration:
  - Run `ccmake .` on Unix-like systems or `cmake-gui` on Windows for a convenient interface.
  - Simply edit the `CMakeCache.txt` file. Some of the relevant options are listed below.
- Once the configuration has been adjusted, run `cmake ..` again.
- Once igraph has been successfully configured, it can be built, tested and installed using:

```
$ cmake --build .
$ cmake --build . --target check
$ cmake --install .
```

Specific instructions for Windows

Microsoft Visual Studio

With Visual Studio, the steps to build igraph are generally the same as above. However, since the Visual Studio CMake generator is a multi-configuration one, we must specify the configuration (typically Release or Debug) with each build command using the `--config` option:

```
mkdir build
cd build
cmake ..
cmake --build . --config Release
cmake --build . --target check --config Release
```

When building the development version, `bison` and `flex` must be available on the system. `winflexbison` [https://github.com/lexxmark/winflexbison] for Bison version 3.x can be useful for this purpose—make sure that the executables are in the system `PATH`. The easiest installation option is probably by installing `winflexbison3` from the Chocolatey package manager [https://chocolatey.org/packages/winflexbison3].

vcpkg

Most external dependencies can be conveniently installed using `vcpkg` [https://github.com/microsoft/vcpkg#quick-start-windows]. Note that `igraph` bundles all dependencies except `libxml2`, which is needed for GraphML support.

In order to use vcpkg integrate it in the build environment by executing `vcpkg.exe integrate install` on the command line. When configuring igraph, point CMake to the correct `vcpkg.cmake` file using `-DCMAKE_TOOLCHAIN_FILE=...`, as instructed.

Additionally, it might be that you need to set the appropriate so-called triplet using `-DVCPKG_TARGET_TRIPLET` when running `cmake`, for exampling, setting it to `x64-windows` when using shared builds of packages or `x64-windows-static` when using static builds. Similarly, you also need to specify this target triplet when installing packages. For example, to install `libxml2` as a shared library, use `vcpkg.exe install libxml2:x64-windows` and to install `libxml2` as a static library, use `vcpkg.exe install libxml2:x64-windows-static`. In addition, there is the possibility to use a static library with dynamic runtime linking using the `x64-windows-static-md` triplet.

MSYS2

MSYS2 can be installed from msys2.org [https://www.msys2.org/]. After installing MSYS2, ensure that it is up to date by opening a terminal and running `pacman -Syuu`.

The instructions below assume that you want to compile for a 64-bit target.

Install the following packages using `pacman -S`.

- Minimal requirements: `mingw-w64-x86_64-toolchain`, `mingw-w64-x86_64-cmake`.
- Optional dependencies that enable certain features: `mingw-w64-x86_64-gmp`, `mingw-w64-x86_64-libxml2`
- Optional external libraries for better performance: `mingw-w64-x86_64-openblas`, `mingw-w64-x86_64-arpack`, `mingw-w64-x86_64-glpk`
- Only needed for running the tests: `diffutils`
- Required only when building the development version: `git`, `bison`, `flex`

The following command will install of these at once:

```
pacman -S \
  mingw-w64-x86_64-toolchain mingw-w64-x86_64-cmake \
  mingw-w64-x86_64-gmp mingw-w64-x86_64-libxml2 \
  mingw-w64-x86_64-openblas mingw-w64-x86_64-arpack \
  mingw-w64-x86_64-glpk diffutils git bison flex
```

In order to build igraph, follow the **General build instructions** above, paying attention to the following:

- When using MSYS2, start the “MSYS2 MinGW 64-bit” terminal, and *not* the “MSYS2 MSYS” one.
- Be sure to install the `mingw-w64-x86_64-cmake` package and not the `cmake` one. The latter will not work.
- When running `cmake`, pass the option `-G"MSYS Makefiles"`.
- Note that `ccmake` is not currently available. `cmake-gui` can be used only if the `mingw-w64-x86_64-qt5` package is installed.

Notable configuration options

The following options may be set to `ON` or `OFF`. Some of them have an `AUTO` setting, which chooses a reasonable default based on what libraries are available on the current system.

- igraph bundles some of its dependencies for convenience. The `IGRAPH_USE_INTERNAL_XXX` flags control whether these should be used instead of external versions. Set them to `ON` to use the bundled (“vendored”) versions. Generally, external versions are preferable as they may be newer and usually provide better performance.

- `IGRAPH_GLPK_SUPPORT`: whether to make use of the GLPK [https://www.gnu.org/software/glpk/] library. Some features, such as finding a minimum feedback arc set or finding communities through exact modularity optimization, require this.
- `IGRAPH_GRAPHML_SUPPORT`: whether to enable support for reading and writing GraphML [http://graphml.graphdrawing.org/] files. Requires the libxml2 [http://xmlsoft.org/] library.
- `IGRAPH_OPENMP_SUPPORT`: whether to use OpenMP parallelization to accelerate certain functions such as PageRank calculation. Compiler support is required.
- `IGRAPH_ENABLE_LTO`: whether to build igraph with link-time optimization, which improves performance. Not supported with all compilers.
- `IGRAPH_ENABLE_TLS`: whether to enable thread-local storage. Required when using igraph from multiple threads.
- `IGRAPH_WARNINGS_AS_ERRORS`: whether to treat compiler warnings as errors. We strive to eliminate all compiler warnings during development so this switch is turned on by default. If your compiler prints warnings for some parts of the code that we did not anticipate, you can turn off this option to prevent the warnings from stopping the compilation.
- `BUILD_SHARED_LIBS` [https://cmake.org/cmake/help/latest/variable/BUILD_SHARED_LIBS.html]: whether to build a shared library instead of a static one.
- `BLA_VENDOR`: controls which library to use for BLAS [https://cmake.org/cmake/help/latest/module/FindBLAS.html] and LAPACK [https://cmake.org/cmake/help/latest/module/FindLAPACK.html] functionality.
- `CMAKE_INSTALL_PREFIX` [https://cmake.org/cmake/help/latest/variable/CMAKE_INSTALL_PREFIX.html]: the location where igraph will be installed.

Building the documentation

Most users will not need to build the documentation, as the release tarball contains pre-built HTML documentation in the `doc` directory.

To build the documentation for the development version, simply build the `html`, `pdf` or `info` targets for the HTML, PDF and Info versions of the documentation, respectively.

```
$ cmake --build . --target html
```

Building the HTML documentation requires Python 3, `xmlto` and `source-highlight`. On some platforms, it is necessary to explicitly install the docbook-xsl package as well. Building the PDF documentation also requires `xsltproc`, `xmllint` and `fop`. Building the Texinfo documentation also requires the docbook2X package, `xmllint` and `makeinfo`.

Notes for package maintainers

This section is for people who package igraph for Linux distros or other package managers. Please read it carefully before packaging igraph.

Auto-detection of dependencies

igraph bundles several of its dependencies (or simplified versions of its dependencies). During configuration time, it checks whether each dependency is present on the system. If yes, it uses it. Otherwise, it falls back to the bundled ("vendored") version. In order to make configuration as deterministic as possible, you may want to disable this auto-detection. To do so, set each of the `IGRAPH_USE_IN-`

TERNAL_XXX options described above. Additionally, set BLA_VENDOR to use the BLAS and LAPACK implementations of your choice. This should be the same BLAS and LAPACK library that igraph's other dependencies (e.g., ARPACK) are linked against.

For example, to force igraph to use external versions of all dependencies except plfit, and to use OpenBLAS for BLAS/LAPACK, use

```
$ cmake .. \
    -DIGRAPH_USE_INTERNAL_BLAS=OFF \
    -DIGRAPH_USE_INTERNAL_LAPACK=OFF \
    -DIGRAPH_USE_INTERNAL_ARPACK=OFF \
    -DIGRAPH_USE_INTERNAL_GLPK=OFF \
    -DIGRAPH_USE_INTERNAL_GMP=OFF \
    -DIGRAPH_USE_INTERNAL_PLFIT=ON \
    -DBLA_VENDOR=OpenBLAS \
    -DIGRAPH_GRAPHML_SUPPORT=ON
```

Shared and static builds

On Windows, shared and static builds should not be installed in the same location. If you decide to do so anyway, keep in mind the following: Both builds contain an `igraph.lib` file. The static one should be renamed to avoid conflict. The headers from the static build are incompatible with the shared library. The headers from the shared build may be used with the static library, but `IGRAPH_STATIC` must be defined when compiling programs that will link to igraph statically.

These issues do not affect Unix-like systems.

Cross-compiling

When building igraph with an internal ARPACK, LAPACK or BLAS, it makes use of f2c, which compiles and runs the `arithchk` program at build time to detect the floating point characteristics of the current system. It writes the results into the `arith.h` header. However, running this program is not possible when cross-compiling without providing a userspace emulator that can run executables of the target platform on the host system. Therefore, when cross-compiling, you either need to provide such an emulator with the `CMAKE_CROSSCOMPILING_EMULATOR` option, or you need to specify a pre-generated version of the `arith.h` header file through the `F2C_EXTERNAL_ARITH_HEADER` CMake option. An example version of this header follows for the x86_64 and arm64 target architectures on macOS. Warning: Do not use this version of `arith.h` on other systems or architectures.

```
#define IEEE_8087
#define Arith_Kind_ASL 1
#define Long int
#define Intcast (int)(long)
#define Double_Align
#define X64_bit_pointers
#define NANCHECK
#define QNaN0 0x0
#define QNaN1 0x7ff80000
```

igraph also checks whether the endianness of `uint64_t` matches the endianness of `double` on the platform being compiled. This is needed to ensure that certain functions in igraph's random number generator work properly. However, it is not possible to execute this check when cross-compiling without an emulator, so in this case igraph simply assumes that the endianness matches (which is the case for the vast majority of platforms anyway). The only case where you might run into problems is when you cross-compile for Apple Silicon (`arm64`) from an Intel-based Mac, in which case CMake might not realize that you are cross-compiling and will try to execute the check anyway. You can

work around this by setting `IEEE754_DOUBLE_ENDIANNESS_MATCHES` to `ON` explicitly before invoking CMake.

Providing an emulator in `CMAKE_CROSSCOMPILING_EMULATOR` has the added benefit that you can run the compiled unit tests on the host platform. We have experimented with cross-compiling to 64-bit ARM CPUs (`aarch64`) on 64-bit Intel CPUs (`amd64`), and we can confirm that using `qemu-aarch64` works as a cross-compiling emulator in this setup.

Additional notes

- As of igraph 0.10, there is no tangible benefit to using an external GMP, as igraph does not yet use GMP in any performance-critical way. The bundled Mini-GMP is sufficient.
- Link-time optimization noticeably improves the performance of some igraph functions. To enable it, use `-DIGRAPH_ENABLE_LTO=ON`. The `AUTO` setting is also supported, and will enable link-time optimization only if the current compiler supports it. Note that this is detected by CMake, and the detection is not always accurate.
- We saw occasional hangs on Windows when igraph was built for a 32-bit target with MinGW and linked to OpenBLAS. We believe this to be an issue with OpenBLAS, not igraph. On this platform, you may want to opt for a different BLAS/LAPACK or the bundled BLAS/LAPACK.

Chapter 3. Tutorial

Compiling programs using igraph

The following short example program demonstrates the basic usage of the **igraph** library. Save it into a file named `igraph_test.c`.

```
#include <igraph.h>

int main(void) {
    igraph_int_t num_vertices = 1000;
    igraph_int_t num_edges = 1000;
    igraph_real_t diameter, mean_degree;
    igraph_t graph;

    /* Initialize the library. */
    igraph_setup();

    /* Ensure identical results across runs. */
    igraph_rng_seed(igraph_rng_default(), 42);

    igraph_erdos_renyi_game_gnm(
            &graph, num_vertices, num_edges,
            IGRAPH_UNDIRECTED, IGRAPH_SIMPLE_SW, IGRAPH_EDGE_UNLABELED);

    igraph_diameter(
        &graph, /* weights = */ NULL,
        &diameter,
        /* from = */ NULL, /* to = */ NULL,
        /* vertex_path = */ NULL, /* edge_path = */ NULL,
        IGRAPH_UNDIRECTED, /* unconn= */ true);

    igraph_mean_degree(&graph, &mean_degree, IGRAPH_LOOPS);
    printf("Diameter of a random graph with average degree %g: %g\n",
           mean_degree, diameter);

    igraph_destroy(&graph);

    return 0;
}
```

This example illustrates a couple of points:

- First, programs using the **igraph** library should include the `igraph.h` header file. Note that while igraph installs several sub-headers, the organization of these may change without notice. Only use `igraph.h` in your projects, not any of the sub-headers.
- Second, the library must be initialized using `igraph_setup()` before use.
- Third, **igraph** uses the igraph_int_t type for integers instead of int or long int, and it also uses the igraph_real_t type for real numbers instead of double. Depending on how **igraph** was compiled, and whether you are using a 32-bit or 64-bit system, igraph_int_t may be a 32-bit or 64-bit integer.
- Fourth, **igraph** graph objects are represented by the igraph_t data type.
- Fifth, the `igraph_erdos_renyi_game_gnm()` creates a graph and `igraph_destroy()` destroys it, i.e. deallocates the memory associated to it.

For compiling this program you need a C compiler. Optionally, CMake [https://cmake.org] can be used to automate the compilation.

Compiling with CMake

It is convenient to use CMake because it can automatically discover the necessary compilation flags on all operating systems. Many IDEs support CMake, and can work with CMake projects directly. To create a CMake project for this example program, create a file name `CMakeLists.txt` with the following contents:

```
cmake_minimum_required(VERSION 3.18)
project(igraph_test)

find_package(igraph REQUIRED)

add_executable(igraph_test igraph_test.c)
target_link_libraries(igraph_test PUBLIC igraph::igraph)
```

To compile the project, create a new directory called `build` in the root of the **igraph** source tree, and switch to it:

```
mkdir build
cd build
```

Run CMake to configure the project:

```
cmake ..
```

If **igraph** was installed at a non-standard location, specify its prefix using the `-DCMAKE_PREFIX_PATH=...` option. The prefix must be the same directory that was specified as the `CMAKE_INSTALL_PREFIX` when compiling igraph.

If configuration has succeeded, build the program using

```
cmake --build .
```

C++ must be enabled in igraph projects

Parts of **igraph** are implemented in C++; therefore, any CMake target that depends on **igraph** should use the C++ linker. Furthermore, OpenMP support in igraph works correctly only if C++ is enabled in the CMake project. The script that finds **igraph** on the host machine will throw an error if C++ support is not enabled in the CMake project.

C++ support is enabled by default when no languages are explicitly specified in CMake's `project` [https://cmake.org/cmake/help/latest/command/project.html] command, e.g. `project(igraph_test)`. If you do specify some languages explicitly, make sure to also include `CXX`, e.g. `project(igraph_test C CXX)`.

Compiling without CMake

On most Unix-like systems, the default C compiler is called **cc**. To compile the test program, you will need a command similar to the following:

```
cc igraph_test.c -I/usr/local/include/igraph -L/usr/local/lib -ligraph -o igraph
```

The exact form depends on where **igraph** was installed on your system, whether it was compiled as a shared or static library, and the external libraries it was linked to. The directory after the `-I` switch is the one containing the `igraph.h` file, while the one following `-L` should contain the library file itself, usually a file called `libigraph.a` (static library on macOS and Linux), `libigraph.so` (shared library on Linux), `libigraph.dylib` (shared library on macOS), `igraph.lib` (static library on Windows) or `igraph.dll` (shared library on Windows). If **igraph** was compiled as a static library, it is also necessary to manually link to all of its dependencies.

If your system has the **pkg-config** utility you are likely to get the necessary compile options by issuing the command

```
pkg-config --libs --cflags igraph
```

(if **igraph** was built as a shared library) or

```
pkg-config --static --libs --cflags igraph
```

(if **igraph** was built as a static library).

Running the program

On most systems, the executable can be run by simply typing its name like this:

```
./igraph_test
```

If you use dynamic linking and the **igraph** library is not installed in a standard place, you may need to add its location to the `LD_LIBRARY_PATH` (Linux), `DYLD_LIBRARY_PATH` (macOS) or `PATH` (Windows) environment variables. This is typically necessary on Windows systems.

Creating your first graphs

The functions generating graph objects are called graph generators. Stochastic (i.e. randomized) graph generators are called “games”.

igraph can handle directed and undirected graphs. Most graph generators are able to create both types of graphs and most other functions are usually also capable of handling both. E.g., `igraph_get_shortest_paths()`, which calculates shortest paths from a vertex to other vertices, can calculate directed or undirected paths.

igraph has sophisticated ways for creating graphs. The simplest graphs are deterministic regular structures like star graphs (`igraph_star()`), cycle graphs (`igraph_cycle_graph()`), lattices (`igraph_square_lattice()`) or trees (`igraph_kary_tree()`), and many more.

The following example creates an undirected regular circular lattice, adds some random edges to it and calculates the average length of shortest paths between all pairs of vertices in the graph before and after adding the random edges. (The message is that some random edges can reduce path lengths a lot.)

```
#include <igraph.h>

int main(void) {
    igraph_t graph;
    igraph_vector_int_t dimvector;
    igraph_vector_int_t edges;
    igraph_vector_bool_t periodic;
    igraph_real_t avg_path_len;
```

```
    /* Initialize the library. */
    igraph_setup();

    igraph_vector_int_init(&dimvector, 2);
    VECTOR(dimvector)[0] = 30;
    VECTOR(dimvector)[1] = 30;

    igraph_vector_bool_init(&periodic, 2);
    igraph_vector_bool_fill(&periodic, true);
    igraph_square_lattice(&graph, &dimvector, 0, IGRAPH_UNDIRECTED,
                          /* mutual= */ false, &periodic);

    igraph_average_path_length(&graph, NULL, &avg_path_len, NULL,
                               IGRAPH_UNDIRECTED, /* unconn= */ true);
    printf("Average path length (lattice):           %g\n", (double) avg_path_

    /* Seed the RNG to ensure identical results across runs. */
    igraph_rng_seed(igraph_rng_default(), 42);

    igraph_vector_int_init(&edges, 20);
    for (igraph_int_t i = 0; i < igraph_vector_int_size(&edges); i++) {
        VECTOR(edges)[i] = RNG_INTEGER(0, igraph_vcount(&graph) - 1);
    }

    igraph_add_edges(&graph, &edges, NULL);
    igraph_average_path_length(&graph, NULL, &avg_path_len, NULL,
                               IGRAPH_UNDIRECTED, /* unconn= */ true);
    printf("Average path length (randomized lattice): %g\n", (double) avg_path_

    igraph_vector_bool_destroy(&periodic);
    igraph_vector_int_destroy(&dimvector);
    igraph_vector_int_destroy(&edges);
    igraph_destroy(&graph);

    return 0;
}
```

This example illustrates some new points. **igraph** uses igraph_vector_t and its related types (igraph_vector_int_t, igraph_vector_bool_t and so on) instead of plain C arrays. igraph_vector_t is superior to regular arrays in almost every sense. Vectors are created by the `igraph_vector_init()` function and, like graphs, they should be destroyed if not needed any more by calling `igraph_vector_destroy()` on them. A vector can be indexed by the `VECTOR()` function (right now it is a macro). The elements of a vector are of type igraph_real_t for igraph_vector_t, and of type igraph_int_t for igraph_vector_int_t. As you might expect, igraph_vector_bool_t holds igraph_bool_t values. Vectors can be resized and most **igraph** functions returning the result in a vector automatically resize it to the size they need.

`igraph_square_lattice()` takes an integer vector argument specifying the dimensions of the lattice. In this example we generate a 30x30 two dimensional periodic lattice. See the documentation of `igraph_square_lattice()` in the reference manual for the other arguments.

The vertices in a graph are identified by a *vertex ID*, an integer between `0` and `n - 1`, where `n` is the number of vertices in the graph. The vertex count can be retrieved using `igraph_vcount()`, as in the example.

The `igraph_add_edges()` function simply takes a graph and a vector of vertex IDs defining the new edges. The first edge is between the first two vertex IDs in the vector, the second edge is between the second two, etc. This way we add ten random edges to the lattice.

Note that this example program may add *loop edges*, edges pointing a vertex to itself, or *multiple edges*, more than one edge between the same pair of vertices. igraph_t can of course represent loops and multiple edges, although some routines expect simple graphs, i.e. graphs which contain neither of these. This is because some structural properties are ill-defined for non-simple graphs. Loop and multi-edges can be removed by calling `igraph_simplify()`.

Calculating various properties of graphs

In our next example we will calculate various centrality measures in a friendship graph. The friendship graph is from the famous Zachary karate club study. (Do a web search on "Zachary karate" if you want to know more about this.) Centrality measures quantify how central is the position of individual vertices in the graph.

```
#include <igraph.h>

int main(void) {
    igraph_t graph;
    igraph_vector_int_t result;
    igraph_vector_t result_real;
    igraph_int_t edges_array[] = {
        0,1, 0,2, 0,3, 0,4, 0,5, 0,6, 0,7, 0,8,
        0,10, 0,11, 0,12, 0,13, 0,17, 0,19, 0,21, 0,31,
        1, 2, 1, 3, 1, 7, 1,13, 1,17, 1,19, 1,21, 1,30,
        2, 3, 2, 7, 2,27, 2,28, 2,32, 2, 9, 2, 8, 2,13,
        3, 7, 3,12, 3,13, 4, 6, 4,10, 5, 6, 5,10, 5,16,
        6,16, 8,30, 8,32, 8,33, 9,33, 13,33, 14,32, 14,33,
        15,32, 15,33, 18,32, 18,33, 19,33, 20,32, 20,33,
        22,32, 22,33, 23,25, 23,27, 23,32, 23,33, 23,29,
        24,25, 24,27, 24,31, 25,31, 26,29, 26,33, 27,33,
        28,31, 28,33, 29,32, 29,33, 30,32, 30,33, 31,32,
        31,33, 32,33
    };
    igraph_vector_int_t edges =
        igraph_vector_int_view(edges_array, sizeof(edges_array) / sizeof(edges_a

    /* Initialize the library. */
    igraph_setup();

    igraph_create(&graph, &edges, 0, IGRAPH_UNDIRECTED);

    igraph_vector_int_init(&result, 0);
    igraph_vector_init(&result_real, 0);

    igraph_degree(&graph, &result, igraph_vss_all(), IGRAPH_ALL, IGRAPH_LOOPS);
    printf("Maximum degree is      %10" IGRAPH_PRId ", vertex %2" IGRAPH_PRId "
           igraph_vector_int_max(&result),
           igraph_vector_int_which_max(&result));

    igraph_closeness(&graph, &result_real, NULL, NULL, igraph_vss_all(),
                     IGRAPH_ALL, /* weights= */ NULL, /* normalized= */ false);
    printf("Maximum closeness is   %10g, vertex %2" IGRAPH_PRId ".\n",
           (double) igraph_vector_max(&result_real),
           igraph_vector_which_max(&result_real));

    igraph_betweenness(&graph, /* weights= */ NULL, &result_real, igraph_vss_al
                       IGRAPH_UNDIRECTED, /* normalized= */ false);
    printf("Maximum betweenness is %10g, vertex %2" IGRAPH_PRId ".\n",
           (double) igraph_vector_max(&result_real),
```

```
            igraph_vector_which_max(&result_real));

    igraph_vector_int_destroy(&result);
    igraph_vector_destroy(&result_real);
    igraph_destroy(&graph);

    return 0;
}
```

This example demonstrates some new operations. First of all, it shows a way to create a graph a list of edges stored in a plain C array. Function `igraph_vector_view()` creates a *view* of a C array. It does not copy any data, which means that you must not call `igraph_vector_destroy()` on a vector created this way. This vector is then used to create the undirected graph.

Then the degree, closeness and betweenness centrality of the vertices is calculated and the highest values are printed. Note that the vector `result`, into which these functions will write their result, must be initialized first, and also that the functions resize it to be able to hold the result.

Notice that in order to print values of type igraph_int_t, we used the `IGRAPH_PRId` format macro constant. This macro is similar to the standard `PRI` constants defined in `stdint.h`, and expands to the correct `printf` format specifier on each platform that **igraph** supports.

The `igraph_vss_all()` argument tells the functions to calculate the property for every vertex in the graph. It is shorthand for a *vertex selector*, represented by type igraph_vs_t. Vertex selectors help perform operations on a subset of vertices. You can read more about them in one of the following chapters.

Chapter 4. Basic data types and interface

The igraph data model

The igraph library can handle directed and undirected graphs. The igraph graphs are multisets of ordered (if directed) or unordered (if undirected) labeled pairs. The labels of the pairs plus the number of vertices always starts with zero and ends with the number of edges minus one. In addition to that, a table of metadata is also attached to every graph, its most important entries being the number of vertices in the graph and whether the graph is directed or undirected.

Like the edges, the igraph vertices are also labeled by numbers between zero and the number of vertices minus one. So, to summarize, a directed graph can be imagined like this:

```
( vertices: 6,
  directed: yes,
  {
   (0,2),
   (2,2),
   (3,2),
   (3,3),
   (3,4),
   (3,4),
   (4,3),
   (4,1)
  }
)
```

Here the edges are ordered pairs or vertex ids, and the graph is a multiset of edges plus some metadata.

An undirected graph is like this:

```
( vertices: 6,
  directed: no,
  {
   (0,2),
   (2,2),
   (2,3),
   (3,3),
   (3,4),
   (3,4),
   (3,4),
   (1,4)
  }
)
```

Here, an edge is an unordered pair of two vertex IDs. A graph is a multiset of edges plus metadata, just like in the directed case.

It is possible to convert between directed and undirected graphs, see the `igraph_to_directed()` and `igraph_to_undirected()` functions.

igraph aims to robustly support multigraphs, i.e. graphs which have more than one edge between some pairs of vertices, as well as graphs with self-loops. Most functions which do not support such graphs

will check their input and issue an error if it is not valid. Those rare functions which do not perform this check clearly indicate this in their documentation. To eliminate multiple edges from a graph, you can use `igraph_simplify()`.

General conventions of igraph functions

igraph has a simple and consistent interface. Most functions check their input for validity and display an informative error message when something goes wrong. In order to support this, the majority of functions return an error code. In basic usage, this code can be ignored, as the default behaviour is to abort the program immediately upon error. See the section on error handling for more information on this topic.

Results are typically returned through *output arguments*, i.e. pointers to a data structure into which the result will be written. In almost all cases, this data structure is expected to be pre-initialized. A few simple functions communicate their result directly through their return value—these functions can never encounter an error.

Atomic data types

igraph introduces a few aliases to standard C data types that are then used throughout the library. The most important of these types is igraph_int_t, which is an alias to either a 32-bit or a 64-bit *signed* integer, depending on whether igraph was compiled in 32-bit or 64-bit mode. The size of igraph_int_t also influences the maximum number of vertices that an igraph graph can represent as the number of vertices is stored in a variable of type igraph_int_t.

Before igraph 1.0, igraph_int_t was called igraph_integer_t. This is still available as an alias to igraph_int_t and will remain accessible until at least version 2.0 of the library.

Since the size of a variable of type igraph_int_t may change depending on how igraph is compiled, you cannot simply use `%d` or `%ld` as a placeholder for igraph integers in `printf` format strings. igraph provides the `IGRAPH_PRId` macro, which maps to `d`, `ld` or `lld` depending on the size of igraph_int_t, and you must use this macro in `printf` format strings to avoid compiler warnings.

Similarly to how igraph_int_t maps to the standard size signed integer in the library, igraph_uint_t maps to a 32-bit or a 64-bit *unsigned* integer. It is guaranteed that the size of igraph_int_t is the same as the size of igraph_uint_t. igraph provides `IGRAPH_PRIu` as a format string placeholder for variables of type igraph_uint_t.

Real numbers (i.e. quantities that can potentially be fractional or infinite) are represented with a type named igraph_real_t. Currently igraph_real_t is always aliased to double, but it is still good practice to use igraph_real_t in your own code for sake of consistency.

Boolean values are represented with a type named igraph_bool_t. It tries to be as small as possible since it only needs to represent a truth value. For printing purposes, you can treat it as an integer and use `%d` in format strings as a placeholder for an igraph_bool_t.

Upper and lower limits of igraph_int_t and igraph_uint_t are provided by the constants named `IGRAPH_INTEGER_MIN`, `IGRAPH_INTEGER_MAX`, `IGRAPH_UINT_MIN` and `IGRAPH_UINT_MAX`.

Setup and initialization

Certain parts of igraph must be initialized before first use, which can be accomplished using the setup functions below. As of igraph 1.0, most functions will work correctly even if setup is not performed, as currently the only setup action is seeding the random number generator. That said, it is strongly recommended to call `igraph_setup()` before using any other function, as future igraph versions may add critical initialization steps.

`igraph_setup` — Initializes the igraph library.

```
igraph_error_t igraph_setup(void);
```

This function is a convenience function to call all setup functions that are provided by the igraph library.

Most of the library functions will work even if this function is not called, but it is recommended to call it before using any igraph functions that may use random numbers, such as graph generators or random sampling functions. This function initializes the random number generator with a seed based on the current time, ensuring that the random numbers generated by igraph are different each time the program is run.

Returns:

Error code; currently always `IGRAPH_SUCCESS`.

The basic interface

This is the very minimal API in **igraph**. All the other functions use this minimal set for creating and manipulating graphs.

This is a very important principle since it makes possible to implement other data representations by implementing only this minimal set.

This section lists all the functions and macros that are considered as part of the core API from the point of view of the *users* of igraph. Some of these functions and macros have sensible default implementations that simply call some other core function (e.g., `igraph_empty()` calls `igraph_empty_attrs()` with a null attribute table pointer). If you wish to experiment with implementing an alternative data type, the actual number of functions that you need to replace is lower as you can rely on the same default implementations in most cases.

Graph constructors and destructors

`igraph_empty` — Creates an empty graph with some vertices and no edges.

```
igraph_error_t igraph_empty(igraph_t *graph, igraph_int_t n, igraph_bool_t dire
```

The most basic constructor, all the other constructors should call this to create a minimal graph object. Our use of the term "empty graph" in the above description should be distinguished from the mathematical definition of the empty or null graph. Strictly speaking, the empty or null graph in graph theory is the graph with no vertices and no edges. However by "empty graph" as used in **igraph** we mean a graph having zero or more vertices, but no edges.

Arguments:

`graph`: Pointer to a not-yet initialized graph object.

`n`: The number of vertices in the graph, a non-negative integer number is expected.

`directed`: Boolean; whether the graph is directed or not. Supported values are:

`IGRAPH_DIRECTED` The graph will be *directed.*

`IGRAPH_UNDIRECTED` The graph will be *undirected.*

Returns:

Error code: `IGRAPH_EINVAL`: invalid number of vertices.

Time complexity: O(|V|) for a graph with |V| vertices (and no edges).

Example 4.1. File `examples/simple/creation.c`

`igraph_empty_attrs` — Creates an empty graph with some vertices, no edges and some graph attributes.

```
igraph_error_t igraph_empty_attrs(
    igraph_t *graph, igraph_int_t n, igraph_bool_t directed,
    const igraph_attribute_record_list_t *attr
);
```

Use this instead of `igraph_empty()` if you wish to add some graph attributes right after initialization. This function is currently not very interesting for the ordinary user. Just supply 0 here or use `igraph_empty()`.

This function does not set any vertex attributes. To create a graph which has vertex attributes, call this function specifying 0 vertices, then use `igraph_add_vertices()` to add vertices and their attributes.

Arguments:

`graph`: Pointer to a not-yet initialized graph object.

`n`: The number of vertices in the graph; a non-negative integer number is expected.

`directed`: Boolean; whether the graph is directed or not. Supported values are:

`IGRAPH_DIRECTED` Create a *directed* graph.

`IGRAPH_UNDIRECTED` Create an *undirected* graph.

`attr`: The graph attributes. Supply `NULL` if not graph attributes are to be set.

Returns:

Error code: `IGRAPH_EINVAL`: invalid number of vertices.

See also:

`igraph_empty()` to create an empty graph without attributes; `igraph_add_vertices()` and `igraph_add_edges()` to add vertices and edges, possibly with associated attributes.

Time complexity: O(|V|) for a graph with |V| vertices (and no edges).

`igraph_copy` — Creates an exact (deep) copy of a graph.

```
igraph_error_t igraph_copy(igraph_t *to, const igraph_t *from);
```

This function deeply copies a graph object to create an exact replica of it. The new replica should be destroyed by calling `igraph_destroy()` on it when not needed any more.

You can also create a shallow copy of a graph by simply using the standard assignment operator, but be careful and do *not* destroy a shallow replica. To avoid this mistake, creating shallow copies is not recommended.

Arguments:

to: Pointer to an uninitialized graph object.

from: Pointer to the graph object to copy.

Returns:

Error code.

Time complexity: O(|V|+|E|) for a graph with |V| vertices and |E| edges.

Example 4.2. File `examples/simple/igraph_copy.c`

`igraph_destroy` — Frees the memory allocated for a graph object.

```
void igraph_destroy(igraph_t *graph);
```

This function should be called for every graph object exactly once.

This function invalidates all iterators (of course), but the iterators of a graph should be destroyed before the graph itself anyway.

Arguments:

graph: Pointer to the graph to free.

Time complexity: operating system specific.

Basic query operations

`igraph_vcount` — The number of vertices in a graph.

```
igraph_int_t igraph_vcount(const igraph_t *graph);
```

Arguments:

graph: The graph.

Returns:

Number of vertices.

Time complexity: O(1)

`igraph_ecount` — The number of edges in a graph.

```
igraph_int_t igraph_ecount(const igraph_t *graph);
```

Arguments:

`graph`: The graph.

Returns:

Number of edges.

Time complexity: O(1)

`igraph_is_directed` — Is this a directed graph?

```
igraph_bool_t igraph_is_directed(const igraph_t *graph);
```

Arguments:

`graph`: The graph.

Returns:

Boolean value, `true` if the graph is directed, `false` otherwise.

Time complexity: O(1)

Example 4.3. File `examples/simple/igraph_is_directed.c`

`igraph_edge` — Returns the head and tail vertices of an edge.

```
igraph_error_t igraph_edge(
    const igraph_t *graph, igraph_int_t eid,
    igraph_int_t *from, igraph_int_t *to
);
```

Arguments:

`graph`: The graph object.

`eid`: The edge ID.

`from`: Pointer to an igraph_int_t. The tail (source) of the edge will be placed here.

`to`: Pointer to an igraph_int_t. The head (target) of the edge will be placed here.

Returns:

Error code.

See also:

`igraph_get_eid()` for the opposite operation; `igraph_edges()` to get the endpoints of several edges; `IGRAPH_TO()`, `IGRAPH_FROM()` and `IGRAPH_OTHER()` for a faster but non-error-checked version.

Added in version 0.2.

Time complexity: O(1).

`igraph_edges` — Gives the head and tail vertices of a series of edges.

```
igraph_error_t igraph_edges(
    const igraph_t *graph, igraph_es_t eids, igraph_vector_int_t *edges,
    igraph_bool_t bycol
);
```

Arguments:

`graph`: The graph object.

`eids`: Edge selector, the series of edges.

`edges`: Pointer to an initialized vector. The start and endpoints of each edge will be placed here.

`bycol`: Boolean constant. If true, the edges will be returned columnwise, e.g. the first edge is `res[0]->res[|E|]`, the second is `res[1]->res[|E|+1]`, etc. Supply false to get the edge list in a format compatible with `igraph_add_edges()`.

Returns:

Error code.

See also:

`igraph_get_eids()` for the opposite operation; `igraph_edge()` for getting the endpoints of a single edge; `IGRAPH_TO()`, `IGRAPH_FROM()` and `IGRAPH_OTHER()` for a faster but non-error-checked method.

Time complexity: O(k) where k is the number of edges in the selector.

`IGRAPH_FROM` — The source vertex of an edge.

```
#define IGRAPH_FROM(graph,eid)
```

Faster than `igraph_edge()`, but no error checking is done: *`eid`* is assumed to be valid.

Arguments:

`graph`: The graph.

`eid`: The edge ID.

Returns:

The source vertex of the edge.

See also:

`igraph_edge()` if error checking is desired.

`IGRAPH_TO` — The target vertex of an edge.

```
#define IGRAPH_TO(graph,eid)
```

Faster than `igraph_edge()`, but no error checking is done: *`eid`* is assumed to be valid.

Arguments:

`graph`: The graph object.

`eid`: The edge ID.

Returns:

The target vertex of the edge.

See also:

`igraph_edge()` if error checking is desired.

`IGRAPH_OTHER` — The other endpoint of an edge.

```
#define IGRAPH_OTHER(graph,eid,vid)
```

Typically used with undirected edges when one endpoint of the edge is known, and the other endpoint is needed. No error checking is done: *`eid`* and *`vid`* are assumed to be valid.

Arguments:

`graph`: The graph object.

`eid`: The edge ID.

`vid`: The vertex ID of one endpoint of an edge.

Returns:

The other endpoint of the edge.

See also:

`IGRAPH_TO()` and `IGRAPH_FROM()` to get the source and target of directed edges.

`igraph_get_eid` — Get the edge ID from the endpoints of an edge.

```
igraph_error_t igraph_get_eid(const igraph_t *graph, igraph_int_t *eid,
                  igraph_int_t from, igraph_int_t to,
                  igraph_bool_t directed, igraph_bool_t error);
```

For undirected graphs `from` and `to` are exchangeable.

Arguments:

graph: The graph object.

eid: Pointer to an integer, the edge ID will be stored here. If *error* is false and no edge was found, `-1` will be returned.

from: The starting point of the edge.

to: The end point of the edge.

directed: Boolean, whether to search for directed edges in a directed graph. Ignored for undirected graphs.

error: Boolean, whether to report an error if the edge was not found. If it is false, then `-1` will be assigned to *eid*. Note that invalid vertex IDs in input arguments (*from* or *to*) always trigger an error, regardless of this setting.

Returns:

Error code.

See also:

`igraph_edge()` for the opposite operation, `igraph_get_all_eids_between()` to retrieve all edge IDs between a pair of vertices.

Time complexity: O(log (d)), where d is smaller of the out-degree of `from` and in-degree of `to` if *directed* is true. If *directed* is false, then it is O(log(d)+log(d2)), where d is the same as before and d2 is the minimum of the out-degree of `to` and the in-degree of `from`.

Example 4.4. File `examples/simple/igraph_get_eid.c`

`igraph_get_eids` — Return edge IDs based on the adjacent vertices.

```
igraph_error_t igraph_get_eids(const igraph_t *graph, igraph_vector_int_t *eids
                   const igraph_vector_int_t *pairs,
                   igraph_bool_t directed, igraph_bool_t error);
```

The pairs of vertex IDs for which the edges are looked up are taken consecutively from the `pairs` vector, i.e. `VECTOR(pairs)[0]` and `VECTOR(pairs)[1]` specify the first pair, `VECTOR(pairs)[2]` and `VECTOR(pairs)[3]` the second pair, etc.

If you have a sequence of vertex IDs that describe a *path* on the graph, use `igraph_expand_path_to_pairs()` to convert them to a list of vertex pairs along the path.

If the `error` argument is true, then it is an error to specify pairs of vertices that are not connected. Otherwise -1 is reported for vertex pairs without at least one edge between them.

If there are multiple edges in the graph, then these are ignored; i.e. for a given pair of vertex IDs, igraph always returns the same edge ID, even if the pair appears multiple times in `pairs`.

Arguments:

graph: The input graph.

eids: Pointer to an initialized vector, the result is stored here. It will be resized as needed.

pairs: Vector giving pairs of vertices to fetch the edges for.

directed: Boolean, whether to consider edge directions in directed graphs. This is ignored for undirected graphs.

error: Boolean, whether it is an error to supply non-connected vertices. If false, then -1 is returned for non-connected pairs.

Returns:

Error code.

Time complexity: O(n log(d)), where n is the number of queried edges and d is the average degree of the vertices.

See also:

`igraph_get_eid()` for a single edge.

Example 4.5. File `examples/simple/igraph_get_eids.c`

`igraph_get_all_eids_between` — Returns all edge IDs between a pair of vertices.

```
igraph_error_t igraph_get_all_eids_between(
    const igraph_t *graph, igraph_vector_int_t *eids,
    igraph_int_t source, igraph_int_t target, igraph_bool_t directed
);
```

For undirected graphs `source` and `target` are exchangeable.

Arguments:

graph: The input graph.

eids: Pointer to an initialized vector, the result is stored here. It will be resized as needed.

source: The ID of the source vertex

target: The ID of the target vertex

directed: Boolean, whether to consider edge directions in directed graphs. This is ignored for undirected graphs.

Returns:

Error code.

Time complexity: TODO

See also:

`igraph_get_eid()` for a single edge.

`igraph_neighbors` — Adjacent vertices to a vertex.

```
igraph_error_t igraph_neighbors(
    const igraph_t *graph, igraph_vector_int_t *neis, igraph_int_t pnode,
    igraph_neimode_t mode, igraph_loops_t loops, igraph_bool_t multiple
);
```

Arguments:

graph: The graph to work on.

neis: This vector will contain the result. The vector should be initialized beforehand and will be resized. Starting from igraph version 0.4 this vector is always sorted, the vertex IDs are in increasing order. If one neighbor is connected with multiple edges, the neighbor will be returned multiple times.

pnode: The id of the node for which the adjacent vertices are to be searched.

mode: Defines the way adjacent vertices are searched in directed graphs. It can have the following values: `IGRAPH_OUT`, vertices reachable by an edge from the specified vertex are searched; `IGRAPH_IN`, vertices from which the specified vertex is reachable are searched; `IGRAPH_ALL`, both kinds of vertices are searched. This parameter is ignored for undirected graphs.

loops: Specifies how to treat loop edges. `IGRAPH_NO_LOOPS` removes loop edges from the result. `IGRAPH_LOOPS_ONCE` makes each loop edge appear only once in the result. `IGRAPH_LOOPS_TWICE` makes loop edges appear *twice* in the result if the graph is undirected or *mode* is set to `IGRAPH_ALL` (and once otherwise as returning them twice does not make sense for directed graphs).

multiple: Specifies how to treat multiple (parallel) edges. `IGRAPH_NO_MULTIPLE` collapses parallel edges into a single one; `IGRAPH_MULTIPLE` keeps the multiplicities of parallel edges so the same neighbor will appear as many times in the result as the number of parallel edges going between the two vertices.

Returns:

Error code: `IGRAPH_EINVVID`: invalid vertex ID. `IGRAPH_EINVMODE`: invalid mode argument. `IGRAPH_ENOMEM`: not enough memory.

Time complexity: O(d), d is the number of adjacent vertices to the queried vertex.

Example 4.6. File `examples/simple/igraph_neighbors.c`

`igraph_incident` — Gives the incident edges of a vertex.

```
igraph_error_t igraph_incident(
    const igraph_t *graph, igraph_vector_int_t *eids, igraph_int_t pnode,
```

```
    igraph_neimode_t mode, igraph_loops_t loops
);
```

Arguments:

graph: The graph object.

eids: An initialized vector. It will be resized to hold the result.

pnode: A vertex ID.

mode: Specifies what kind of edges to include for directed graphs. IGRAPH_OUT means only outgoing edges, IGRAPH_IN means only incoming edges, IGRAPH_ALL means both. This parameter is ignored for undirected graphs.

loops: Specifies how to treat loop edges. IGRAPH_NO_LOOPS removes loop edges from the result. IGRAPH_LOOPS_ONCE makes each loop edge appear only once in the result. IGRAPH_LOOPS_TWICE makes loop edges appear *twice* in the result if the graph is undirected or *mode* is set to IGRAPH_ALL (and once otherwise as returning them twice does not make sense for directed graphs).

Returns:

Error code: IGRAPH_EINVVID: invalid vertex ID. IGRAPH_EINVMODE: invalid mode argument. IGRAPH_ENOMEM: not enough memory.

Time complexity: O(d), the number of incident edges to *pnode*.

igraph_degree — The degree of some vertices in a graph.

```
igraph_error_t igraph_degree(
    const igraph_t *graph, igraph_vector_int_t *res, const igraph_vs_t vids,
    igraph_neimode_t mode, igraph_loops_t loops
);
```

This function calculates the in-, out- or total degree of the specified vertices.

This function returns the result as a vector of igraph_int_t values. In applications where igraph_real_t is desired, use igraph_strength() with NULL weights.

Arguments:

graph: The graph.

res: Integer vector, this will contain the result. It should be initialized and will be resized to be the appropriate size.

vids: Vertex selector, giving the vertex IDs of which the degree will be calculated.

mode: Defines the type of the degree for directed graphs. Valid modes are: IGRAPH_OUT, out-degree; IGRAPH_IN, in-degree; IGRAPH_ALL, total degree (sum of the in- and out-degree). This parameter is ignored for undirected graphs.

loops: Constant of type igraph_loops_t, specifies how to treat loop edges when calculating the degree. IGRAPH_NO_LOOPS ignores loop edges; IGRAPH_LOOPS_ONCE counts each loop edge only once; IGRAPH_LOOPS_TWICE counts each loop edge twice in undirected graphs and once in directed graphs.

Returns:

Error code: `IGRAPH_EINVVID`: invalid vertex ID. `IGRAPH_EINVMODE`: invalid mode argument.

Time complexity: O(v) if *`loops`* is `true`, and O(v*d) otherwise. v is the number of vertices for which the degree will be calculated, and d is their (average) degree.

See also:

`igraph_strength()` for the version that takes into account edge weights; `igraph_degree_1()` to efficiently compute the degree of a single vertex; `igraph_maxdegree()` if you only need the largest degree.

Example 4.7. File `examples/simple/igraph_degree.c`

`igraph_degree_1` — The degree of of a single vertex in the graph.

```
igraph_error_t igraph_degree_1(
    const igraph_t *graph, igraph_int_t *deg, igraph_int_t vid,
    igraph_neimode_t mode, igraph_loops_t loops
);
```

This function calculates the in-, out- or total degree of a single vertex. For a single vertex, it is more efficient than calling `igraph_degree()`.

Arguments:

`graph`: The graph.

`deg`: Pointer to the integer where the computed degree will be stored.

`vid`: The vertex for which the degree will be calculated.

`mode`: Defines the type of the degree for directed graphs. Valid modes are: `IGRAPH_OUT`, out-degree; `IGRAPH_IN`, in-degree; `IGRAPH_ALL`, total degree (sum of the in- and out-degree). This parameter is ignored for undirected graphs.

`loops`: Boolean, gives whether the self-loops should be counted.

Returns:

Error code.

See also:

`igraph_degree()` to compute the degree of several vertices at once.

Time complexity: O(1) if *`loops`* is `true`, and O(d) otherwise, where d is the degree.

Adding and deleting vertices and edges

`igraph_add_edge` — Adds a single edge to a graph.

```
igraph_error_t igraph_add_edge(igraph_t *graph, igraph_int_t from, igraph_int_t
```

For directed graphs the edge points from *from* to *to*.

Note that if you want to add many edges to a big graph, then it is inefficient to add them one by one, it is better to collect them into a vector and add all of them via a single `igraph_add_edges()` call.

Arguments:

graph: The graph.

from: The id of the first vertex of the edge.

to: The id of the second vertex of the edge.

Returns:

Error code.

See also:

`igraph_add_edges()` to add many edges, `igraph_delete_edges()` to remove edges and `igraph_add_vertices()` to add vertices.

Time complexity: O(|V|+|E|), the number of edges plus the number of vertices.

`igraph_add_edges` — Adds edges to a graph object.

```
igraph_error_t igraph_add_edges(
    igraph_t *graph, const igraph_vector_int_t *edges,
    const igraph_attribute_record_list_t *attr
);
```

The edges are given in a vector, the first two elements define the first edge (the order is `from`, `to` for directed graphs). The vector should contain even number of integer numbers between zero and the number of vertices in the graph minus one (inclusive). If you also want to add new vertices, call `igraph_add_vertices()` first.

Arguments:

graph: The graph to which the edges will be added.

edges: The edges themselves.

attr: The attributes of the new edges. You can supply a null pointer here if you do not need edge attributes.

Returns:

Error code: `IGRAPH_EINVAL`: invalid (odd) edges vector length, `IGRAPH_EINVVID`: invalid vertex ID in edges vector.

This function invalidates all iterators.

Time complexity: O(|V|+|E|) where |V| is the number of vertices and |E| is the number of edges in the *new*, extended graph.

Example 4.8. File `examples/simple/creation.c`

`igraph_add_vertices` — Adds vertices to a graph.

```
igraph_error_t igraph_add_vertices(
    igraph_t *graph, igraph_int_t nv, const igraph_attribute_record_list_t *att
);
```

This function invalidates all iterators.

Arguments:

`graph`: The graph object to extend.

`nv`: Non-negative integer specifying the number of vertices to add.

`attr`: The attributes of the new vertices. You can supply a null pointer here if you do not need vertex attributes.

Returns:

Error code: `IGRAPH_EINVAL`: invalid number of new vertices.

Time complexity: O(|V|) where |V| is the number of vertices in the *new*, extended graph.

Example 4.9. File `examples/simple/creation.c`

`igraph_delete_edges` — Removes edges from a graph.

```
igraph_error_t igraph_delete_edges(igraph_t *graph, igraph_es_t edges);
```

The edges to remove are specified as an edge selector.

This function cannot remove vertices; vertices will be kept even if they lose all their edges.

This function invalidates all iterators.

Arguments:

`graph`: The graph to work on.

`edges`: The edges to remove.

Returns:

Error code.

Time complexity: O(|V|+|E|) where |V| and |E| are the number of vertices and edges in the *original* graph, respectively.

Example 4.10. File `examples/simple/igraph_delete_edges.c`

`igraph_delete_vertices` — Removes some vertices (with all their edges) from the graph.

```
igraph_error_t igraph_delete_vertices(igraph_t *graph, const igraph_vs_t vertic
```

This function changes the IDs of the vertices (except in some very special cases, but these should not be relied on anyway).

This function invalidates all iterators.

Arguments:

graph: The graph to work on.

vertices: The IDs of the vertices to remove, in a vector. The vector may contain the same ID more than once.

Returns:

Error code: `IGRAPH_EINVVID`: invalid vertex ID.

Time complexity: O(|V|+|E|), |V| and |E| are the number of vertices and edges in the original graph.

Example 4.11. File `examples/simple/igraph_delete_vertices.c`

`igraph_delete_vertices_map` — Removes some vertices (with all their edges) from the graph.

```
igraph_error_t igraph_delete_vertices_map(
    igraph_t *graph, const igraph_vs_t vertices, igraph_vector_int_t *map,
    igraph_vector_int_t *invmap
);
```

This function changes the IDs of the vertices (except in some very special cases, but these should not be relied on anyway). You can use the *map* argument to obtain the mapping from old vertex IDs to the new ones, and the *newmap* argument to obtain the reverse mapping.

This function invalidates all iterators.

Arguments:

graph: The graph to work on.

vertices: The IDs of the vertices to remove, in a vector. The vector may contain the same ID more than once.

map: An optional pointer to a vector that provides the mapping from the vertex IDs *before* the removal to the vertex IDs *after* the removal. You can supply `NULL` here if you are not interested.

invmap: An optional pointer to a vector that provides the mapping from the vertex IDs *after* the removal to the vertex IDs *before* the removal. You can supply `NULL` here if you are not interested.

Returns:

Error code: `IGRAPH_EINVVID`: invalid vertex ID.

Time complexity: O(|V|+|E|), |V| and |E| are the number of vertices and edges in the original graph.

Miscellaneous macros and helper functions

IGRAPH_VCOUNT_MAX — The maximum number of vertices supported in igraph graphs.

```
#define IGRAPH_VCOUNT_MAX
```

The value of this constant is one less than IGRAPH_INTEGER_MAX . When igraph is compiled in 32-bit mode, this means that you are limited to $2^{31} - 2$ (about 2.1 billion) vertices. In 64-bit mode, the limit is $2^{63} - 2$ so you are much more likely to hit out-of-memory issues due to other reasons before reaching this limit.

IGRAPH_ECOUNT_MAX — The maximum number of edges supported in igraph graphs.

```
#define IGRAPH_ECOUNT_MAX
```

The value of this constant is half of IGRAPH_INTEGER_MAX . When igraph is compiled in 32-bit mode, this means that you are limited to approximately 2^{30} (about 1.07 billion) vertices. In 64-bit mode, the limit is approximately 2^{62} so you are much more likely to hit out-of-memory issues due to other reasons before reaching this limit.

IGRAPH_UNLIMITED — Constant for "do not limit results".

```
#define IGRAPH_UNLIMITED
```

A constant signifying that no limitation should be used with various cutoff, size limit or result set size parameters, such as minimum or maximum clique size, number of results returned, cutoff for path lengths, etc. Currently defined to -1.

igraph_expand_path_to_pairs — Helper function to convert a sequence of vertex IDs describing a path into a "pairs" vector.

```
igraph_error_t igraph_expand_path_to_pairs(igraph_vector_int_t* path);
```

This function is useful when you have a sequence of vertex IDs in a graph and you would like to retrieve the IDs of the edges between them. The function duplicates all but the first and the last elements in the vector, effectively converting the path into a vector of vertex IDs that can be passed to igraph_get_eids().

Arguments:

path: the input vector. It will be modified in-place and it will be resized as needed. When the vector contains less than two vertex IDs, it will be cleared.

Returns:

Error code: `IGRAPH_ENOMEM` if there is not enough memory to expand the vector.

`igraph_invalidate_cache` — Invalidates the internal cache of an igraph graph.

```
void igraph_invalidate_cache(const igraph_t* graph);
```

igraph graphs cache some basic properties about themselves in an internal data structure. This function invalidates the contents of the cache and forces a recalculation of the cached properties the next time they are needed.

You should not need to call this function during normal usage; however, we might ask you to call this function explicitly if we suspect that you are running into a bug in igraph's cache handling. A tell-tale sign of an invalid cache entry is that the result of a cached igraph function (such as `igraph_is_dag()` or `igraph_is_simple()`) is different before and after a cache invalidation.

Arguments:

`graph`: The graph whose cache is to be invalidated.

Time complexity: O(1).

`igraph_is_same_graph` — Are two graphs identical as labelled graphs?

```
igraph_error_t igraph_is_same_graph(const igraph_t *graph1, const igraph_t *gra
```

Two graphs are considered to be the same if they have the same vertex and edge sets. Graphs which are the same may have multiple different representations in igraph, hence the need for this function.

This function verifies that the two graphs have the same directedness, the same number of vertices, and that they contain precisely the same edges (regardless of their ordering) when written in terms of vertex indices. Graph attributes are not taken into account.

This concept is different from isomorphism. For example, the graphs `0-1, 2-1` and `1-2, 0-1` are considered the same because they only differ in the ordering of their edge lists and the ordering of vertices in an undirected edge. However, they are not the same as `0-2, 1-2`, even though they are isomorphic to it. Note that this latter graph contains the edge `0-2` while the former two do not — thus their edge sets differ.

Arguments:

`graph1`: The first graph object.

`graph2`: The second graph object.

`res`: The result will be stored here.

Returns:

Error code.

Time complexity: O(E), the number of edges in the graphs.

See also:

`igraph_isomorphic()` to test if two graphs are isomorphic.

Chapter 5. Error handling

Error handling basics

igraph functions can run into various problems preventing them from normal operation. The user might have supplied invalid arguments, e.g. a non-square matrix when a square-matrix was expected, or the program has run out of memory while some more memory allocation is required, etc.

By default **igraph** aborts the program when it runs into an error. While this behavior might be good enough for smaller programs, it is without doubt avoidable in larger projects. Please read further if your project requires more sophisticated error handling. You can safely skip the rest of this chapter otherwise.

Error handlers

If **igraph** runs into an error - an invalid argument was supplied to a function, or we've ran out of memory - the control is transferred to the *error handler* function.

The default error handler is `igraph_error_handler_abort` which prints an error message and aborts the program.

The `igraph_set_error_handler()` function can be used to set a new error handler function of type `igraph_error_handler_t`; see the documentation of this type for details.

There are two other predefined error handler functions, `igraph_error_handler_ignore` and `igraph_error_handler_printignore`. These deallocate the temporarily allocated memory (more about this later) and return with the error code. The latter also prints an error message. If you use these error handlers you need to take care about possible errors yourself by checking the return value of (almost) every non-void **igraph** function.

Independently of the error handler installed, all functions in the library do their best to leave their arguments *semantically* unchanged if an error happens. By semantically we mean that the implementation of an object supplied as an argument might change, but its “meaning” in most cases does not. The rare occasions when this rule is violated are documented in this manual.

`igraph_error_handler_t` — The type of error handler functions.

```
typedef void igraph_error_handler_t(const char *reason, const char *file,
                                    int line, igraph_error_t igraph_errno);
```

This is the type of the error handler functions.

Arguments:

`reason`: Textual description of the error.

`file`: The source file in which the error is noticed.

`line`: The number of the line in the source file which triggered the error

`igraph_errno`: The **igraph** error code.

`igraph_error_handler_abort` — Abort program in case of error.

```
IGRAPH_FUNCATTR_NORETURN igraph_error_handler_t igraph_error_handler_abort;
```

The default error handler, prints an error message and aborts the program.

`igraph_error_handler_ignore` — Ignore errors.

```
igraph_error_handler_t igraph_error_handler_ignore;
```

This error handler frees the temporarily allocated memory and returns with the error code.

`igraph_error_handler_printignore` — Print and ignore errors.

```
igraph_error_handler_t igraph_error_handler_printignore;
```

Frees temporarily allocated memory, prints an error message to the standard error and returns with the error code.

Error codes

Every **igraph** function which can fail return a single integer error code. Some functions are very simple and cannot run into any error, these may return other types, or void as well. The error codes are defined by the `igraph_error_type_t` enumeration.

`igraph_error_t` — Return type for functions returning an error code.

```
typedef igraph_error_type_t igraph_error_t;
```

This type is used as the return type of igraph functions that return an error code. It is a type alias because igraph_error_t used to be an `int`, and was used slightly differenly than igraph_error_type_t.

`igraph_error_type_t` — Error code type.

```
typedef enum {
    IGRAPH_SUCCESS           = 0,
    IGRAPH_FAILURE           = 1,
    IGRAPH_ENOMEM            = 2,
    IGRAPH_PARSEERROR        = 3,
    IGRAPH_EINVAL            = 4,
    IGRAPH_EXISTS            = 5,
    /* IGRAPH_EINVEVECTOR       = 6, */   /* removed in 1.0 */
    IGRAPH_EINVVID           = 7,
    IGRAPH_EINVEID           = 8,         /* used to be IGRAPH_NONSQUARE before
    IGRAPH_EINVMODE          = 9,
    IGRAPH_EFILE             = 10,
```

```
    IGRAPH_UNIMPLEMENTED     = 12,
    IGRAPH_INTERRUPTED       = 13,
    IGRAPH_DIVERGED          = 14,
    IGRAPH_EARPACK           = 15,
    /* ARPACK error codes from 15 to 36 were moved to igraph_arpack_error_t in
    IGRAPH_ENEGCYCLE         = 37,
    IGRAPH_EINTERNAL         = 38,
    /* ARPACK error codes from 39 to 41 were moved to igraph_arpack_error_t in
    /* IGRAPH_EDIVZERO          = 42, */   /* removed in 1.0 */
    /* IGRAPH_GLP_EBOUND        = 43, */   /* removed in 1.0 */
    /* IGRAPH_GLP_EROOT         = 44, */   /* removed in 1.0 */
    /* IGRAPH_GLP_ENOPFS        = 45, */   /* removed in 1.0 */
    /* IGRAPH_GLP_ENODFS        = 46, */   /* removed in 1.0 */
    /* IGRAPH_GLP_EFAIL         = 47, */   /* removed in 1.0 */
    /* IGRAPH_GLP_EMIPGAP       = 48, */   /* removed in 1.0 */
    /* IGRAPH_GLP_ETMLIM        = 49, */   /* removed in 1.0 */
    /* IGRAPH_GLP_ESTOP         = 50, */   /* removed in 1.0 */
    /* IGRAPH_EATTRIBUTES       = 51, */   /* removed in 1.0 */
    IGRAPH_EATTRCOMBINE      = 52,
    /* IGRAPH_ELAPACK           = 53, */   /* removed in 1.0 */
    /* IGRAPH_EDRL              = 54, */   /* deprecated in 0.10.2, removed in
    IGRAPH_EOVERFLOW         = 55,
    /* IGRAPH_EGLP              = 56, */   /* removed in 1.0 */
    /* IGRAPH_CPUTIME           = 57, */   /* removed in 1.0 */
    IGRAPH_EUNDERFLOW        = 58,
    IGRAPH_ERWSTUCK          = 59,
    IGRAPH_STOP              = 60,
    IGRAPH_ERANGE            = 61,
    IGRAPH_ENOSOL            = 62
} igraph_error_type_t;
```

These are the possible values returned by **igraph** functions. Note that these are interesting only if you defined an error handler with `igraph_set_error_handler()`. Otherwise the program is aborted and the function causing the error never returns.

Values:

`IGRAPH_SUCCESS`: The function successfully completed its task.

`IGRAPH_FAILURE`: Something went wrong. You'll almost never meet this error as normally more specific error codes are used.

`IGRAPH_ENOMEM`: There wasn't enough memory to allocate on the heap.

`IGRAPH_PARSEERROR`: A parse error was found in a file.

`IGRAPH_EINVAL`: A parameter's value is invalid. E.g. negative number was specified as the number of vertices.

`IGRAPH_EXISTS`: A graph/vertex/edge attribute is already installed with the given name.

`IGRAPH_EINVVID`: Invalid vertex ID, negative or too big.

`IGRAPH_EINVEID`: Invalid edge ID, negative or too big.

`IGRAPH_EINVMODE`: Invalid mode parameter.

`IGRAPH_EFILE`: A file operation failed. E.g. a file doesn't exist, or the user has no rights to open it.

IGRAPH_UNIMPLEMENTED:	Attempted to call an unimplemented or disabled (at compile-time) function.
IGRAPH_DIVERGED:	A numeric algorithm failed to converge.
IGRAPH_ARPACK:	An error happened inside a calculation implemented in ARPACK. The calculation involved is most likely an eigenvector-related calculation.
IGRAPH_ENEGCYCLE:	Negative cycle detected while calculating shortest paths.
IGRAPH_EINTERNAL:	Internal error, likely a bug in igraph.
IGRAPH_EATTRCOMBINE:	Unimplemented attribute combination method for the given attribute type.
IGRAPH_EOVERFLOW:	Integer or double overflow.
IGRAPH_EUNDERFLOW:	Integer or double underflow.
IGRAPH_ERWSTUCK:	Random walk got stuck.
IGRAPH_ERANGE:	Maximum vertex or edge count exceeded.
IGRAPH_ENOSOL:	Input problem has no solution.

igraph_strerror — Textual description of an error.

```
const char *igraph_strerror(const igraph_error_t igraph_errno);
```

This is a simple utility function, it gives a short general textual description for an **igraph** error code.

Arguments:

igraph_errno: The **igraph** error code.

Returns:

pointer to the textual description of the error code.

Warning messages

igraph also supports warning messages in addition to error messages. Warning messages typically do not terminate the program, but they are usually crucial to the user.

igraph warnings are handled similarly to errors. There is a separate warning handler function that is called whenever an **igraph** function triggers a warning. This handler can be set by the `igraph_set_warning_handler()` function. There are two predefined simple warning handlers, `igraph_warning_handler_ignore()` and `igraph_warning_handler_print()`, the latter being the default.

To trigger a warning, **igraph** functions typically use the `IGRAPH_WARNING()` macro, the `igraph_warning()` function, or if more flexibility is needed, `igraph_warningf()`.

igraph_warning_handler_t — The type of igraph warning handler functions.

```
typedef void igraph_warning_handler_t(const char *reason,
                                      const char *file, int line);
```

Currently it is defined to have the same type as `igraph_error_handler_t`, although the last (error code) argument is not used.

`igraph_set_warning_handler` — Installs a warning handler.

```
igraph_warning_handler_t *igraph_set_warning_handler(igraph_warning_handler_t *
```

Install the supplied warning handler function.

Arguments:

`new_handler`: The new warning handler function to install. Supply a null pointer here to uninstall the current warning handler, without installing a new one.

Returns:

The current warning handler function.

`IGRAPH_WARNING` — Triggers a warning.

```
#define IGRAPH_WARNING(reason)
```

This is the usual way of triggering a warning from an igraph function. It calls `igraph_warning()`.

Arguments:

`reason`: The warning message.

`IGRAPH_WARNINGF` — Triggers a warning, with printf-like syntax.

```
#define IGRAPH_WARNINGF(reason, ...)
```

igraph functions can use this macro when they notice a warning and want to pass on extra information to the user about what went wrong. It calls `igraph_warningf()` with the proper parameters and no error code.

Arguments:

`reason`: Textual description of the warning, a template string with the same syntax as the standard printf C library function.

`...`: The additional arguments to be substituted into the template string.

`igraph_warning` — Reports a warning.

```
void igraph_warning(const char *reason, const char *file, int line);
```

Call this function if you want to trigger a warning from within a function that uses **igraph**.

Arguments:

reason: Textual description of the warning.

file: The source file in which the warning was noticed.

line: The number of line in the source file which triggered the warning.

igraph_warningf — Reports a warning, printf-like version.

```
void igraph_warningf(const char *reason, const char *file, int line, ...);
```

This function is similar to `igraph_warning()`, but uses a printf-like syntax. It substitutes the additional arguments into the *reason* template string and calls `igraph_warning()`.

Arguments:

reason: Textual description of the warning, a template string with the same syntax as the standard printf C library function.

file: The source file in which the warning was noticed.

line: The number of line in the source file which triggered the warning.

...: The additional arguments to be substituted into the template string.

igraph_warning_handler_ignore — Ignores all warnings.

```
void igraph_warning_handler_ignore(const char *reason, const char *file, int li
```

This warning handler function simply ignores all warnings.

Arguments:

reason: Textual description of the warning.

file: The source file in which the warning was noticed.

line: The number of line in the source file which triggered the warning..

igraph_warning_handler_print — Prints all warnings to the standard error.

```
void igraph_warning_handler_print(const char *reason, const char *file, int lin
```

This warning handler function simply prints all warnings to the standard error.

Arguments:

reason: Textual description of the warning.

file: The source file in which the warning was noticed.

line: The number of line in the source file which triggered the warning..

Advanced topics

Writing error handlers

The contents of the rest of this chapter might be useful only for those who want to create an interface to **igraph** from another language, or use igraph from a GUI application. Most readers can safely skip to the next chapter.

You can write and install error handlers simply by defining a function of type `igraph_error_handler_t` and calling `igraph_set_error_handler()`. This feature is useful for interface writers, as **igraph** will have the chance to signal errors the appropriate way. For example, the R interface uses R's native printing facilities to communicate errors, while the Python interface converts them into Python exceptions.

The two main tasks of the error handler are to report the error (i.e. print the error message) and ensure proper resource cleanup. This is ensured by calling `IGRAPH_FINALLY_FREE()`, which deallocates some of the temporary memory to avoid memory leaks. Note that this may invalidate the error message buffer *reason* passed to the error handler. Do not access it after having called `IGRAPH_FINALLY_FREE()`.

As of **igraph** 0.10, temporary memory is dellocated in stages, through multiple calls to the error handler (and indirectly to `IGRAPH_FINALLY_FREE()`). Therefore, error handlers that do not abort the program immediately are expected to return. The error handler should not perform a `longjmp`, as this may lead to some of the memory not getting freed.

`igraph_set_error_handler` — Sets a new error handler.

```
igraph_error_handler_t *igraph_set_error_handler(igraph_error_handler_t *new_ha
```

Installs a new error handler. If called with `NULL`, it installs the default error handler (which is currently `igraph_error_handler_abort`).

Arguments:

new_handler: The error handler function to install.

Returns:

The old error handler function. This should be saved and restored if *new_handler* is not needed any more.

Error handling internals

If an error happens, the functions in the library call the `IGRAPH_ERROR()` macro with a textual description of the error and an **igraph** error code. This macro calls (through the `igraph_error()` function) the installed error handler. Another useful macro is `IGRAPH_CHECK()`. This checks the return value of its argument, which is normally a function call, and calls `IGRAPH_ERROR()` if it is not `IGRAPH_SUCCESS`.

IGRAPH_ERROR — Triggers an error.

```
#define IGRAPH_ERROR(reason, igraph_errno)
```

igraph functions usually use this macro when they notice an error. It calls `igraph_error()` with the proper parameters and if that returns the macro returns the "calling" function as well, with the error code. If for some (suspicious) reason you want to call the error handler without returning from the current function, call `igraph_error()` directly.

Arguments:

reason: Textual description of the error. This should be something more descriptive than the text associated with the error code. E.g. if the error code is `IGRAPH_EINVAL`, its associated text (see `igraph_strerror()`) is "Invalid value" and this string should explain which parameter was invalid and maybe why.

igraph_errno: The **igraph** error code.

IGRAPH_ERRORF — Triggers an error, with printf-like syntax.

```
#define IGRAPH_ERRORF(reason, igraph_errno, ...)
```

igraph functions can use this macro when they notice an error and want to pass on extra information to the user about what went wrong. It calls `igraph_errorf()` with the proper parameters and if that returns the macro returns the "calling" function as well, with the error code. If for some (suspicious) reason you want to call the error handler without returning from the current function, call `igraph_errorf()` directly.

Arguments:

reason: Textual description of the error, a template string with the same syntax as the standard printf C library function. This should be something more descriptive than the text associated with the error code. E.g. if the error code is `IGRAPH_EINVAL`, its associated text (see `igraph_strerror()`) is "Invalid value" and this string should explain which parameter was invalid and maybe what was expected and what was recieved.

igraph_errno: The **igraph** error code.

...: The additional arguments to be substituted into the template string.

igraph_error — Reports an error.

```
igraph_error_t igraph_error(const char *reason, const char *file, int line,
                            igraph_error_t igraph_errno);
```

igraph functions usually call this function (most often via the `IGRAPH_ERROR` macro) if they notice an error. It calls the currently installed error handler function with the supplied arguments.

Arguments:

reason: Textual description of the error.

file: The source file in which the error was noticed.

line: The number of line in the source file which triggered the error.

igraph_errno: The **igraph** error code.

Returns:

The error code (if it returns).

See also:

`igraph_errorf()`

`igraph_errorf` — Reports an error, printf-like version.

```
igraph_error_t igraph_errorf(const char *reason, const char *file, int line,
                             igraph_error_t igraph_errno, ...);
```

Arguments:

reason: Textual description of the error, interpreted as a `printf` format string.

file: The source file in which the error was noticed.

line: The line in the source file which triggered the error.

igraph_errno: The **igraph** error code.

...: Additional parameters, the values to substitute into the format string.

Returns:

The error code (if it returns).

See also:

`igraph_error()`

`IGRAPH_CHECK` — Checks the return value of a function call.

```
#define IGRAPH_CHECK(expr)
```

Arguments:

expr: An expression, usually a function call. It is guaranteed to be evaluated only once.

Executes the expression and checks its value. If this is not `IGRAPH_SUCCESS`, it calls `IGRAPH_ERROR` with the value as the error code. Here is an example usage:

```
IGRAPH_CHECK(vector_push_back(&v, 100));
```

There is only one reason to use this macro when writing **igraph** functions. If the user installs an error handler which returns to the auxiliary calling code (like `igraph_error_handler_ignore` and `igraph_error_handler_printignore`), and the **igraph** function signalling the error is called from another **igraph** function then we need to make sure that the error is propagated back to

the auxiliary (i.e. non-igraph) calling function. This is achieved by using IGRAPH_CHECK on every **igraph** call which can return an error code.

IGRAPH_CHECK_CALLBACK — Checks the return value of a callback.

```
#define IGRAPH_CHECK_CALLBACK(expr, code)
```

Identical to IGRAPH_CHECK, but treats IGRAPH_STOP as a normal (non-erroneous) return code. This macro is used in some igraph functions that allow the user to hook into a long-running calculation with a callback function. When the user-defined callback function returns IGRAPH_SUCCESS, the calculation will proceed normally. Returning IGRAPH_STOP from the callback will terminate the calculation without reporting an error. Returning any other value from the callback is treated as an error code, and igraph will trigger the necessary cleanup functions before exiting the function.

Note that IGRAPH_CHECK_CALLBACK does not handle IGRAPH_STOP by any means except returning it in the variable pointed to by code. It is the responsibility of the caller to handle IGRAPH_STOP accordingly.

Arguments:

expr: An expression, usually a call to a user-defined callback function. It is guaranteed to be evaluated only once.

code: Pointer to an optional variable of type igraph_error_t; the value of this variable will be set to the error code if it is not a null pointer.

Deallocating memory

If a function runs into an error (and the program is not aborted) the error handler should deallocate all temporary memory. This is done by storing the address and the destroy function of all temporary objects in a stack. The IGRAPH_FINALLY function declares an object as temporary by placing its address in the stack. If an **igraph** function returns with success it calls IGRAPH_FINALLY_CLEAN() with the number of objects to remove from the stack. If an error happens however, the error handler should call IGRAPH_FINALLY_FREE() to deallocate each object added to the stack. This means that the temporary objects allocated in the calling function (and etc.) will be freed as well.

IGRAPH_FINALLY — Registers an object for deallocation.

```
#define IGRAPH_FINALLY(func, ptr)
```

This macro places the address of an object, together with the address of its destructor on a stack. This stack is used if an error happens to deallocate temporarily allocated objects to prevent memory leaks. After manual deallocation, objects are removed from the stack using IGRAPH_FINALLY_CLEAN().

The typical usage is just after an initialization:

```
IGRAPH_CHECK(igraph_vector_init(&vector, 0));
IGRAPH_FINALLY(igraph_vector_destroy, &vector);
```

The most commonly used data structures, such as igraph_vector_t, have associated convenience macros that initialize the object and register it on this stack in one step. Thus the pattern above can be replaced with a single line:

```
IGRAPH_VECTOR_INIT_FINALLY(&vector, 0);
```

Arguments:

func: The function which is normally called to destroy the object.

ptr: Pointer to the object itself.

IGRAPH_FINALLY_CLEAN — Signals clean deallocation of objects.

```
void IGRAPH_FINALLY_CLEAN(int num);
```

Removes the specified number of objects from the stack of temporarily allocated objects. It is typically called immediately after manually destroying the objects:

```
igraph_vector_t vector;
igraph_vector_init(&vector, 10);
IGRAPH_FINALLY(igraph_vector_destroy, &vector);
// use vector
igraph_vector_destroy(&vector);
IGRAPH_FINALLY_CLEAN(1);
```

Arguments:

num: The number of objects to remove from the bookkeeping stack.

IGRAPH_FINALLY_FREE — Deallocates objects registered at the current level.

```
void IGRAPH_FINALLY_FREE(void);
```

Calls the destroy function for all objects in the current level of the stack of temporarily allocated objects, i.e. up to the nearest mark set by `IGRAPH_FINALLY_ENTER()`. This function must only be called from an error handler. It is *not* appropriate to use it instead of destroying each unneeded object of a function, as it destroys the temporary objects of the caller function (and so on) as well.

Writing igraph functions with proper error handling

There are some simple rules to keep in order to have functions behaving well in erroneous situations. First, check the arguments of the functions and call `IGRAPH_ERROR()` if they are invalid. Second, call `IGRAPH_FINALLY` on each dynamically allocated object and call `IGRAPH_FINALLY_CLEAN()` with the proper argument before returning. Third, use `IGRAPH_CHECK` on all **igraph** function calls which can generate errors.

The size of the stack used for this bookkeeping is fixed, and small. If you want to allocate several objects, write a destroy function which can deallocate all of these. See the `adjlist.c` file in the **igraph** source for an example.

For some functions these mechanisms are simply not flexible enough. These functions should define their own error handlers and restore the error handler before they return.

Example 5.1. File `examples/simple/igraph_contract_vertices.c`

Fatal errors

In some rare situations, **igraph** may encounter an internal error that cannot be fully handled. In this case, it will call the current fatal error handler. The default fatal error handler simply prints the error and aborts the program.

Fatal error handlers do not return. Typically, they might abort the the program immediately, or in the case of the high-level **igraph** interfaces, they might return to the top level using a `longjmp()`. The fatal error handler is only called when a serious error has occurred, and as a result igraph may be in an inconsistent state. The purpose of returning to the top level is to give the user a chance to save their work instead of aborting immediately. However, the program session should be restarted as soon as possible.

Most projects that use **igraph** will use the default fatal error handler.

`igraph_fatal_handler_t` — The type of igraph fatal error handler functions.

```
typedef void igraph_fatal_handler_t(const char *reason, const char *file, int li
```

Functions of this type *must* not return. Typically they call `abort()` or do a `longjmp()`.

Arguments:

`reason`: Textual description of the error.

`file`: The source file in which the error is noticed.

`line`: The number of the line in the source file which triggered the error.

`igraph_set_fatal_handler` — Installs a fatal error handler.

```
igraph_fatal_handler_t *igraph_set_fatal_handler(igraph_fatal_handler_t *new_ha
```

Installs the supplied fatal error handler function.

Fatal error handler functions *must* not return. Typically, the fatal error handler would either call `abort()` or `longjmp()`.

Arguments:

`new_handler`: The new fatal error handler function to install. Supply a null pointer here to uninstall the current fatal error handler, without installing a new one.

Returns:

The current fatal error handler function.

`igraph_fatal_handler_abort` — Abort program in case of fatal error.

```
IGRAPH_FUNCATTR_NORETURN igraph_fatal_handler_t igraph_fatal_handler_abort;
```

The default fatal error handler, prints an error message and aborts the program.

IGRAPH_FATAL — Triggers a fatal error.

```
#define IGRAPH_FATAL(reason)
```

This is the usual way of triggering a fatal error from an igraph function. It calls `igraph_fatal()`.

Use this macro only in situations where the error cannot be handled. The normal way to handle errors is `IGRAPH_ERROR()`.

Arguments:

reason: The error message.

IGRAPH_FATALF — Triggers a fatal error, with printf-like syntax.

```
#define IGRAPH_FATALF(reason, ...)
```

igraph functions can use this macro when a fatal error occurs and want to pass on extra information to the user about what went wrong. It calls `igraph_fatalf()` with the proper parameters.

Arguments:

reason: Textual description of the error, a template string with the same syntax as the standard printf C library function.

...: The additional arguments to be substituted into the template string.

IGRAPH_ASSERT — igraph-specific replacement for `assert()`.

```
#define IGRAPH_ASSERT(condition)
```

This macro is like the standard `assert()`, but instead of calling `abort()`, it calls `igraph_fatal()`. This allows for returning the control to the calling program, e.g. returning to the top level in a high-level **igraph** interface.

Unlike `assert()`, `IGRAPH_ASSERT()` is not disabled when the `NDEBUG` macro is defined.

This macro is meant for internal use by **igraph**.

Since a typical fatal error handler does a `longjmp()`, avoid using this macro in C++ code. With most compilers, destructor will not be called when `longjmp()` leaves the current scope.

Arguments:

condition: The condition to be checked.

igraph_fatal — Triggers a fatal error.

```
void igraph_fatal(const char *reason, const char *file, int line);
```

This function triggers a fatal error. Typically it is called indirectly through `IGRAPH_FATAL()` or `IGRAPH_ASSERT()`.

Arguments:

reason: Textual description of the error.

file: The source file in which the error was noticed.

line: The number of line in the source file which triggered the error.

`igraph_fatalf` — Triggers a fatal error, printf-like syntax.

```
void igraph_fatalf(const char *reason, const char *file, int line, ...);
```

This function is similar to `igraph_fatal()`, but uses a printf-like syntax. It substitutes the additional arguments into the *reason* template string and calls `igraph_fatal()`.

Arguments:

reason: Textual description of the error.

file: The source file in which the error was noticed.

line: The number of line in the source file which triggered the error.

...: The additional arguments to be substituted into the template string.

Error handling and threads

It is likely that the **igraph** error handling method is *not* thread-safe, mainly because of the static global stack which is used to store the address of the temporarily allocated objects. This issue might be addressed in a later version of **igraph**.

Chapter 6. Memory (de)allocation

About allocation functions

Some igraph functions return a pointer vector (igraph_vector_ptr_t) containing pointers to other igraph or other data types. These data types are dynamically allocated and have to be deallocated manually when the user does not need them any more. igraph_vector_ptr_t has functions to deallocate the contained pointers on its own, but in this case it has to be ensured that these pointers are allocated by a function that corresponds to the deallocator function that igraph uses.

To this end, igraph exports the memory allocation functions that are used internally so the user of the library can ensure that the proper functions are used when pointers are moved between the code written by the user and the code of the igraph library.

Additionally, the memory allocator functions used by igraph work around the quirks of classical `malloc`(), `realloc`() and `calloc`() implementations where the behaviour of allocating zero bytes is undefined. igraph allocator functions will always allocate at least one byte.

Available allocation functions

`igraph_malloc` — Allocates memory that can be safely deallocated by igraph functions.

```
void *igraph_malloc(size_t size);
```

This function behaves like `malloc`(), but it ensures that at least one byte is allocated even when the caller asks for zero bytes.

Arguments:

`size`: Number of bytes to be allocated. Zero is treated as one byte.

Returns:

Pointer to the piece of allocated memory; `NULL` if the allocation failed.

See also:

`igraph_calloc()`, `igraph_realloc()`, `igraph_free()`

`igraph_calloc` — Allocates memory that can be safely deallocated by igraph functions.

```
void *igraph_calloc(size_t count, size_t size);
```

This function behaves like `calloc`(), but it ensures that at least one byte is allocated even when the caller asks for zero bytes.

Arguments:

`count`: Number of items to be allocated.

size: Size of a single item to be allocated.

Returns:

Pointer to the piece of allocated memory; `NULL` if the allocation failed.

See also:

`igraph_malloc()`, `igraph_realloc()`, `igraph_free()`

`igraph_realloc` — Reallocate memory that can be safely deallocated by igraph functions.

```
void *igraph_realloc(void *ptr, size_t size);
```

This function behaves like `realloc`(), but it ensures that at least one byte is allocated even when the caller asks for zero bytes.

Arguments:

ptr: The pointer to reallocate.

size: Number of bytes to be allocated.

Returns:

Pointer to the piece of allocated memory; `NULL` if the allocation failed.

See also:

`igraph_free()`, `igraph_malloc()`

`igraph_free` — Deallocates memory that was allocated by igraph functions.

```
void igraph_free(void *ptr);
```

This function exposes the `free`() function used internally by igraph.

Arguments:

ptr: Pointer to the piece of memory to be deallocated.

Time complexity: platform dependent, ideally it should be O(1).

See also:

`igraph_calloc()`, `igraph_malloc()`, `igraph_realloc()`

Chapter 7. Data structure library: vector, matrix, other data types

About template types

Some of the container types listed in this section are defined for many base types. This is similar to templates in C++ and generics in Ada, but it is implemented via preprocessor macros since the C language cannot handle it. Here is the list of template types and the all base types they currently support:

vector	Vector is currently defined for igraph_real_t, igraph_int_t (int), char (char), igraph_bool_t (bool), igraph_complex_t (complex) and and void * (ptr). The default is igraph_real_t.
matrix	Matrix is currently defined for igraph_real_t, igraph_int_t (int), char (char), igraph_bool_t (bool) and igraph_complex_t (complex). The default is igraph_real_t.
array3	Array3 is currently defined for igraph_real_t, igraph_int_t (int), char (char) and igraph_bool_t (bool). The default is igraph_real_t.
stack	Stack is currently defined for igraph_real_t, igraph_int_t (int), char (char) and igraph_bool_t (bool). The default is igraph_real_t.
double-ended queue	Dqueue is currently defined for igraph_real_t, igraph_int_t (int), char (char) and igraph_bool_t (bool). The default is igraph_real_t.
heap	Heap is currently defined for igraph_real_t, igraph_int_t (int), char (char). In addition both maximum and minimum heaps are available. The default is the igraph_real_t maximum heap.
list of vectors	Lists of vectors are currently defined for vectors holding igraph_real_t and igraph_int_t (int). The default is igraph_real_t.
list of matrices	Lists of matrices are currently defined for matrices holding igraph_real_t only.

The name of the base element (in parentheses) is added to the function names, except for the default type.

Some examples:

- igraph_vector_t is a vector of igraph_real_t elements. Its functions are `igraph_vector_init`, `igraph_vector_destroy`, `igraph_vector_sort`, etc.
- igraph_vector_bool_t is a vector of igraph_bool_t elements; initialize it with `igraph_vector_bool_init`, destroy it with `igraph_vector_bool_destroy`, etc.
- igraph_heap_t is a maximum heap with igraph_real_t elements. The corresponding functions are `igraph_heap_init`, `igraph_heap_pop`, etc.
- igraph_heap_min_t is a minimum heap with igraph_real_t elements. The corresponding functions are called `igraph_heap_min_init`, `igraph_heap_min_pop`, etc.
- igraph_heap_int_t is a maximum heap with igraph_int_t elements. Its functions have the `igraph_heap_int_` prefix.

- igraph_heap_min_int_t is a minimum heap containing igraph_int_t elements. Its functions have the `igraph_heap_min_int_` prefix.
- igraph_vector_list_t is a list of (floating-point) vectors; each element in this data structure is an igraph_vector_t. Similarly, igraph_matrix_list_t is a list of (floating-point) matrices; each element in this data structure is an igraph_matrix_t.
- igraph_vector_int_list_t is a list of integer vectors; each element in this data structure is an igraph_vector_int_t.

Note that the VECTOR and the MATRIX macros can be used on *all* vector and matrix types. VECTOR cannot be used on *lists* of vectors, though, only on the individial vectors in the list.

Vectors

About igraph_vector_t objects

The igraph_vector_t data type is a simple and efficient interface to arrays containing numbers. It is something similar to (but much simpler than) the vector template in the C++ standard library.

There are multiple variants of igraph_vector_t; the basic variant stores doubles, but there is also igraph_vector_int_t for integers (of type igraph_int_t), `igraph_vector_bool_t` for booleans (of type igraph_bool_t) and so on. Vectors are used extensively in **igraph**; all functions that expect or return a list of numbers use igraph_vector_t or igraph_vector_int_t to achieve this. Integer vectors are typically used when the vector is supposed to hold vertex or edge identifiers, while igraph_vector_t is used when the vector is expected to hold fractional numbers or infinities.

The igraph_vector_t type and its variants usually use O(n) space to store n elements. Sometimes they use more, this is because vectors can shrink, but even if they shrink, the current implementation does not free a single bit of memory.

The elements in an igraph_vector_t object and its variants are indexed from zero, we follow the usual C convention here.

The elements of a vector always occupy a single block of memory, the starting address of this memory block can be queried with the `VECTOR` macro. This way, vector objects can be used with standard mathematical libraries, like the GNU Scientific Library.

Almost all of the functions described below for igraph_vector_t also exist for all the other vector type variants. These variants are not documented separately; you can simply replace `vector` with `vector_int`, `vector_bool` or something similar if you need a function for another variant. For instance, to initialize a vector of type igraph_vector_int_t, you need to use `igraph_vector_int_init()` and not `igraph_vector_init()`.

Constructors and destructors

igraph_vector_t objects have to be initialized before using them, this is analogous to calling a constructor on them. There are a number of igraph_vector_t constructors, for your convenience. `igraph_vector_init()` is the basic constructor, it creates a vector of the given length, filled with zeros. `igraph_vector_init_copy()` creates a new identical copy of an already existing and initialized vector. `igraph_vector_init_array()` creates a vector by copying a regular C array. `igraph_vector_init_range()` creates a vector containing a regular sequence with increment one.

`igraph_vector_view()` is a special constructor, it allows you to handle a regular C array as a vector without copying its elements.

If a igraph_vector_t object is not needed any more, it should be destroyed to free its allocated memory by calling the igraph_vector_t destructor, `igraph_vector_destroy()`.

Note that vectors created by `igraph_vector_view()` are special, you must not call `igraph_vector_destroy()` on these.

`igraph_vector_init` — Initializes a vector object (constructor).

```
igraph_error_t igraph_vector_init(igraph_vector_t *v, igraph_int_t size);
```

Every vector needs to be initialized before it can be used, and there are a number of initialization functions or otherwise called constructors. This function constructs a vector of the given size and initializes each entry to 0. Note that `igraph_vector_null()` can be used to set each element of a vector to zero. However, if you want a vector of zeros, it is much faster to use this function than to create a vector and then invoke `igraph_vector_null()`.

Every vector object initialized by this function should be destroyed (ie. the memory allocated for it should be freed) when it is not needed anymore, the `igraph_vector_destroy()` function is responsible for this.

Arguments:

v: Pointer to a not yet initialized vector object.

size: The size of the vector.

Returns:

error code: `IGRAPH_ENOMEM` if there is not enough memory.

Time complexity: operating system dependent, the amount of “time” required to allocate O(n) elements, n is the number of elements.

`igraph_vector_init_array` — Initializes a vector from an ordinary C array (constructor).

```
igraph_error_t igraph_vector_init_array(
        igraph_vector_t *v, const igraph_real_t *data, igraph_int_t length);
```

Arguments:

v: Pointer to an uninitialized vector object.

data: A regular C array.

length: The length of the C array.

Returns:

Error code: `IGRAPH_ENOMEM` if there is not enough memory.

Time complexity: operating system specific, usually O(*length*).

`igraph_vector_init_copy` — Initializes a vector from another vector object (constructor).

```
igraph_error_t igraph_vector_init_copy(
```

```
    igraph_vector_t *to, const igraph_vector_t *from
);
```

The contents of the existing vector object will be copied to the new one.

Arguments:

to: Pointer to a not yet initialized vector object.

from: The original vector object to copy.

Returns:

Error code: `IGRAPH_ENOMEM` if there is not enough memory.

Time complexity: operating system dependent, usually O(n), n is the size of the vector.

`igraph_vector_init_range` — Initializes a vector with a range.

```
igraph_error_t igraph_vector_init_range(igraph_vector_t *v, igraph_real_t start,
```

The vector will contain the numbers *start*, *start*+1, ..., *end*-1. Note that the range is closed from the left and open from the right, according to C conventions.

Arguments:

v: Pointer to an uninitialized vector object.

start: The lower limit in the range (inclusive).

end: The upper limit in the range (exclusive).

Returns:

Error code: `IGRAPH_ENOMEM`: out of memory.

Time complexity: O(n), the number of elements in the vector.

`igraph_vector_destroy` — Destroys a vector object.

```
void igraph_vector_destroy(igraph_vector_t *v);
```

All vectors initialized by `igraph_vector_init()` should be properly destroyed by this function. A destroyed vector needs to be reinitialized by `igraph_vector_init()`, `igraph_vector_init_array()` or another constructor.

Arguments:

v: Pointer to the (previously initialized) vector object to destroy.

Time complexity: operating system dependent.

Initializing elements

`igraph_vector_null` — Sets each element in the vector to zero.

```
void igraph_vector_null(igraph_vector_t *v);
```

Note that `igraph_vector_init()` sets the elements to zero as well, so it makes no sense to call this function on a just initialized vector. Thus if you want to construct a vector of zeros, then you should use `igraph_vector_init()`.

Arguments:

v: The vector object.

Time complexity: O(n), the size of the vector.

`igraph_vector_fill` — Fill a vector with a constant element.

```
void igraph_vector_fill(igraph_vector_t *v, igraph_real_t e);
```

Sets each element of the vector to the supplied constant.

Arguments:

vector: The vector to work on.

e: The element to fill with.

Time complexity: O(n), the size of the vector.

`igraph_vector_range` — Updates a vector to store a range.

```
igraph_error_t igraph_vector_range(igraph_vector_t *v, igraph_real_t start, igra
```

Sets the elements of the vector to contain the numbers *start*, *start*+1, ..., *end*-1. Note that the range is closed from the left and open from the right, according to C conventions. The vector will be resized to fit the range.

Arguments:

v: The vector to update.

start: The lower limit in the range (inclusive).

end: The upper limit in the range (exclusive).

Returns:

Error code: `IGRAPH_ENOMEM`: out of memory.

Time complexity: O(n), the number of elements in the vector.

Accessing elements

The simplest and most performant way to access an element of a vector is to use the `VECTOR` macro. This macro can be used both for querying and setting igraph_vector_t elements. If you need a function, `igraph_vector_get()` queries and `igraph_vector_set()` sets an element of a vector. `igraph_vector_get_ptr()` returns the address of an element.

`igraph_vector_tail()` returns the last element of a non-empty vector. There is no `igraph_vector_head()` function however, as it is easy to write `VECTOR(v)[0]` instead.

VECTOR — Accessing an element of a vector.

```
#define VECTOR(v)
```

Usage:

```
VECTOR(v)[0]
```

to access the first element of the vector, you can also use this in assignments, like:

```
VECTOR(v)[10] = 5;
```

Note that there are no range checks right now.

Arguments:

v: The vector object.

Time complexity: O(1).

igraph_vector_get — Access an element of a vector.

```
igraph_real_t igraph_vector_get(const igraph_vector_t *v, igraph_int_t pos);
```

Unless you need a function, consider using the VECTOR macro instead for better performance.

Arguments:

v: The igraph_vector_t object.

pos: The position of the element, the index of the first element is zero.

Returns:

The desired element.

See also:

igraph_vector_get_ptr() and the VECTOR macro.

Time complexity: O(1).

igraph_vector_get_ptr — Get the address of an element of a vector.

```
igraph_real_t* igraph_vector_get_ptr(const igraph_vector_t *v, igraph_int_t pos
```

Unless you need a function, consider using the VECTOR macro instead for better performance.

Arguments:

v: The igraph_vector_t object.

pos: The position of the element, the position of the first element is zero.

Returns:

Pointer to the desired element.

See also:

`igraph_vector_get()` and the `VECTOR` macro.

Time complexity: O(1).

`igraph_vector_set` — Assignment to an element of a vector.

```
void igraph_vector_set(igraph_vector_t *v, igraph_int_t pos, igraph_real_t valu
```

Unless you need a function, consider using the `VECTOR` macro instead for better performance.

Arguments:

v: The igraph_vector_t element.

pos: Position of the element to set.

value: New value of the element.

See also:

`igraph_vector_get()`.

`igraph_vector_tail` — Returns the last element in a vector.

```
igraph_real_t igraph_vector_tail(const igraph_vector_t *v);
```

It is an error to call this function on an empty vector, the result is undefined.

Arguments:

v: The vector object.

Returns:

The last element.

Time complexity: O(1).

`igraph_vector_index` — Extract elements from a vector at specific indices.

```
igraph_error_t igraph_vector_index(const igraph_vector_t *v,
                                   igraph_vector_t *newv,
                                   const igraph_vector_int_t *idx);
```

Arguments:

v: the vector to extract elements from

newv: the result vector

idx: vector containing the indices of the elements to extract

See also:

`igraph_vector_index_in_place` for the in-place variant

`igraph_vector_index_in_place` — Extract elements from a vector at specific indices in-place.

```
igraph_error_t igraph_vector_index_in_place(igraph_vector_t *v,
                                            const igraph_vector_int_t *idx);
```

Arguments:

v: the vector to extract elements from. This will be modified in-place.

idx: vector containing the indices of the elements to extract

See also:

`igraph_vector_index` for a function that does not modify the original vector

Vector views

`igraph_vector_view` — Handle a regular C array as a igraph_vector_t.

```
igraph_vector_t igraph_vector_view(
        const igraph_real_t *data, igraph_int_t length);
```

This function lets you treat an existing C array as an `igraph_vector_t`.

Since this function creates a view into an existing array, you must *not* destroy the `igraph_vector_t` object when you are done with it. Similarly, you must *not* call any function on it that may attempt to modify the size of the vector. Modifying an element in the vector will modify the underlying array as the two share the same memory block.

Typical usage pattern:

```
igraph_real_t array[] = { 1.0, 1.5, 2.0 };
const igraph_vector_t v = igraph_vector_view(array, sizeof(array) / sizeof(arra
printf("The sum of vector elements is %g.\n", igraph_vector_sum(&v));
```

Arguments:

data: The raw array that the vector provides a view into.

length: The length of the C array.

Returns:

The vector object providing the view into the array.

Time complexity: O(1)

Copying vectors

`igraph_vector_copy_to` — Copies the contents of a vector to a C array.

```
void igraph_vector_copy_to(const igraph_vector_t *v, igraph_real_t *to);
```

The C array should have sufficient length.

Arguments:

v: The vector object.

to: The C array.

Time complexity: O(n), n is the size of the vector.

`igraph_vector_update` — Update a vector from another one.

```
igraph_error_t igraph_vector_update(igraph_vector_t *to,
                                    const igraph_vector_t *from);
```

After this operation the contents of *to* will be exactly the same as that of *from*. The vector *to* will be resized if it was originally shorter or longer than *from*.

Arguments:

to: The vector to update.

from: The vector to update from.

Returns:

Error code.

Time complexity: O(n), the number of elements in *from*.

`igraph_vector_append` — Append a vector to another one.

```
igraph_error_t igraph_vector_append(igraph_vector_t *to,
                                    const igraph_vector_t *from);
```

The target vector will be resized (except when *from* is empty).

Arguments:

to: The vector to append to.

from: The vector to append, it is kept unchanged.

Returns:

Error code.

Time complexity: O(n), the number of elements in the new vector.

`igraph_vector_swap` — Swap all elements of two vectors.

```
void igraph_vector_swap(igraph_vector_t *v1, igraph_vector_t *v2);
```

Arguments:

v1: The first vector.

v2: The second vector.

Time complexity: O(1).

Exchanging elements

`igraph_vector_swap_elements` — Swap two elements in a vector.

```
void igraph_vector_swap_elements(igraph_vector_t *v,
                                 igraph_int_t i, igraph_int_t j);
```

Note that currently no range checking is performed.

Arguments:

v: The input vector.

i: Index of the first element.

j: Index of the second element (may be the same as the first one).

Time complexity: O(1).

`igraph_vector_reverse` — Reverse the elements of a vector.

```
void igraph_vector_reverse(igraph_vector_t *v);
```

The first element will be last, the last element will be first, etc.

Arguments:

v: The input vector.

See also:

igraph_vector_reverse_section() to reverse only a section of a vector.

Time complexity: O(n), the number of elements.

`igraph_vector_reverse_section` — Reverse the elements in a section of a vector.

```
void igraph_vector_reverse_section(igraph_vector_t *v, igraph_int_t from, igraph
```

Arguments:

v: The input vector.

from: Index of the first element to include in the reversal.

to: Index of the first element *not* to include in the reversal.

See also:

igraph_vector_reverse() to reverse the entire vector.

Time complexity: O(to - from), the number of elements to reverse.

`igraph_vector_rotate_left` — Rotates the elements of a vector to the left.

```
void igraph_vector_rotate_left(igraph_vector_t *v, igraph_int_t n);
```

Rotates the elements of a vector to the left by the given number of steps. Element index *n* will have index 0 after the rotation. For example, rotating `(0, 1, 2, 3, 4, 5)` by 2 yields `(2, 3, 4, 5, 0, 1)`.

Arguments:

v: The input vector.

n: The number of steps to rotate by. Passing a negative value rotates to the right.

Time complexity: O(n), the number of elements.

`igraph_vector_shuffle` — Shuffles a vector in-place using the Fisher-Yates method.

```
void igraph_vector_shuffle(igraph_vector_t *v);
```

The Fisher-Yates shuffle ensures that every permutation is equally probable when using a proper randomness source. Of course this does not apply to pseudo-random generators as the cycle of these generators is less than the number of possible permutations of the vector if the vector is long enough.

Arguments:

v: The vector object.

Time complexity: O(n), n is the number of elements in the vector.

References:

`(Fisher & Yates 1963)` R. A. Fisher and F. Yates. *Statistical Tables for Biological, Agricultural and Medical Research.* Oliver and Boyd, 6th edition, 1963, page 37.

`(Knuth 1998)` D. E. Knuth. *Seminumerical Algorithms,* volume 2 of *The Art of Computer Programming.* Addison-Wesley, 3rd edition, 1998, page 145.

Example 7.1. File examples/simple/igraph_fisher_yates_shuffle.c

igraph_vector_permute — Permutes the elements of a vector in place according to an index vector.

```
igraph_error_t igraph_vector_permute(igraph_vector_t *v, const igraph_vector_in
```

This function takes a vector v and a corresponding index vector ind, and permutes the elements of v such that v[ind[i]] is moved to become v[i] after the function is executed.

It is an error to call this function with an index vector that does not represent a valid permutation. Each element in the index vector must be between 0 and the length of the vector minus one (inclusive), and each such element must appear only once. The function does not attempt to validate the index vector.

The index vector that this function takes is compatible with the index vector returned from igraph_vector_sort_ind(); passing in the index vector from igraph_vector_sort_ind() will sort the original vector.

As a special case, this function allows the index vector to be *shorter* than the vector being permuted, in which case the elements whose indices do not occur in the index vector will be removed from the vector.

Arguments:

v: the vector to permute

ind: the index vector

Returns:

Error code: IGRAPH_ENOMEM if there is not enough memory.

Time complexity: O(n), the size of the vector.

Vector operations

igraph_vector_add_constant — Add a constant to the vector.

```
void igraph_vector_add_constant(igraph_vector_t *v, igraph_real_t plus);
```

plus is added to every element of *v*. Note that overflow might happen.

Arguments:

v: The input vector.

plus: The constant to add.

Time complexity: O(n), the number of elements.

igraph_vector_scale — Multiplies all elements of a vector by a constant.

```
void igraph_vector_scale(igraph_vector_t *v, igraph_real_t by);
```

Arguments:

v: The vector.

by: The constant.

Returns:

Error code. The current implementation always returns with success.

Added in version 0.2.

Time complexity: O(n), the number of elements in a vector.

`igraph_vector_add` — Add two vectors.

```
igraph_error_t igraph_vector_add(igraph_vector_t *v1,
                                 const igraph_vector_t *v2);
```

Add the elements of *v2* to *v1*, the result is stored in *v1*. The two vectors must have the same length.

Arguments:

v1: The first vector, the result will be stored here.

v2: The second vector, its contents will be unchanged.

Returns:

Error code.

Time complexity: O(n), the number of elements.

`igraph_vector_sub` — Subtract a vector from another one.

```
igraph_error_t igraph_vector_sub(igraph_vector_t *v1,
                                 const igraph_vector_t *v2);
```

Subtract the elements of *v2* from *v1*, the result is stored in *v1*. The two vectors must have the same length.

Arguments:

v1: The first vector, to subtract from. The result is stored here.

v2: The vector to subtract, it will be unchanged.

Returns:

Error code.

Time complexity: O(n), the length of the vectors.

`igraph_vector_mul` — Multiply two vectors.

```
igraph_error_t igraph_vector_mul(igraph_vector_t *v1,
                                 const igraph_vector_t *v2);
```

v1 will be multiplied by *v2*, elementwise. The two vectors must have the same length.

Arguments:

v1: The first vector, the result will be stored here.

v2: The second vector, it is left unchanged.

Returns:

Error code.

Time complexity: O(n), the number of elements.

igraph_vector_div — Divide a vector by another one.

```
igraph_error_t igraph_vector_div(igraph_vector_t *v1,
                                 const igraph_vector_t *v2);
```

v1 is divided by *v2*, elementwise. They must have the same length. If the base type of the vector can generate divide by zero errors then please make sure that *v2* contains no zero if you want to avoid trouble.

Arguments:

v1: The dividend. The result is also stored here.

v2: The divisor, it is left unchanged.

Returns:

Error code.

Time complexity: O(n), the length of the vectors.

igraph_vector_floor — Transform a real vector to an integer vector by flooring each element.

```
igraph_error_t igraph_vector_floor(const igraph_vector_t *from, igraph_vector_i
```

Flooring means rounding down to the nearest integer.

Arguments:

from: The original real vector object.

to: Pointer to an initialized integer vector. The result will be stored here.

Returns:

Error code: IGRAPH_ENOMEM: out of memory

Time complexity: O(n), where n is the number of elements in the vector.

Vector comparisons

igraph_vector_all_e — Are all elements equal?

```
igraph_bool_t igraph_vector_all_e(const igraph_vector_t *lhs,
                                  const igraph_vector_t *rhs);
```

Checks element-wise equality of two vectors. For vectors containing floating point values, consider using `igraph_matrix_all_almost_e()`.

Arguments:

lhs: The first vector.

rhs: The second vector.

Returns:

True if the elements in the *lhs* are all equal to the corresponding elements in *rhs*. Returns false if the lengths of the vectors don't match.

Time complexity: O(n), the length of the vectors.

igraph_vector_all_almost_e — Are all elements almost equal?

```
igraph_bool_t igraph_vector_all_almost_e(const igraph_vector_t *lhs,
                                         const igraph_vector_t *rhs,
                                         igraph_real_t eps);
```

Checks if the elements of two vectors are equal within a relative tolerance.

Arguments:

lhs: The first vector.

rhs: The second vector.

eps: Relative tolerance, see `igraph_almost_equals()` for details.

Returns:

True if the two vectors are almost equal, false if there is at least one differing element or if the vectors are not of the same size.

igraph_vector_all_l — Are all elements less?

```
igraph_bool_t igraph_vector_all_l(const igraph_vector_t *lhs,
                                  const igraph_vector_t *rhs);
```

Arguments:

lhs: The first vector.

rhs: The second vector.

Returns:

True if the elements in the *lhs* are all less than the corresponding elements in *rhs*. Returns false if the lengths of the vectors don't match. If any element is NaN, it will return false.

Time complexity: O(n), the length of the vectors.

igraph_vector_all_g — Are all elements greater?

```
igraph_bool_t igraph_vector_all_g(const igraph_vector_t *lhs,
                                  const igraph_vector_t *rhs);
```

Arguments:

lhs: The first vector.

rhs: The second vector.

Returns:

True if the elements in the *lhs* are all greater than the corresponding elements in *rhs*. Returns false if the lengths of the vectors don't match. If any element is NaN, it will return false.

Time complexity: O(n), the length of the vectors.

igraph_vector_all_le — Are all elements less or equal?

```
igraph_bool_t igraph_vector_all_le(const igraph_vector_t *lhs,
                                   const igraph_vector_t *rhs);
```

Arguments:

lhs: The first vector.

rhs: The second vector.

Returns:

True if the elements in the *lhs* are all less than or equal to the corresponding elements in *rhs*. Returns false if the lengths of the vectors don't match. If any element is NaN, it will return false.

Time complexity: O(n), the length of the vectors.

igraph_vector_all_ge — Are all elements greater or equal?

```
igraph_bool_t igraph_vector_all_ge(const igraph_vector_t *lhs,
                                   const igraph_vector_t *rhs);
```

Arguments:

lhs: The first vector.

rhs: The second vector.

Returns:

True if the elements in the *lhs* are all greater than or equal to the corresponding elements in *rhs*. Returns false if the lengths of the vectors don't match. If any element is NaN, it will return false.

Time complexity: O(n), the length of the vectors.

igraph_vector_is_equal — Are all elements equal?

```
igraph_bool_t igraph_vector_is_equal(const igraph_vector_t *lhs,
                                     const igraph_vector_t *rhs);
```

This is an alias of `igraph_vector_all_e()` with a more intuitive name.

Arguments:

lhs: The first vector.

rhs: The second vector.

Returns:

True if the elements in the *lhs* are all equal to the corresponding elements in *rhs*. Returns false if the lengths of the vectors don't match.

Time complexity: O(n), the length of the vectors.

igraph_vector_zapsmall — Replaces small elements of a vector by exact zeros.

```
igraph_error_t igraph_vector_zapsmall(igraph_vector_t *v, igraph_real_t tol);
```

Vector elements which are smaller in magnitude than the given absolute tolerance will be replaced by exact zeros. The default tolerance corresponds to two-thirds of the representable digits of igraph_real_t, i.e. `DBL_EPSILON^(2/3)` which is approximately `10^-10`.

Arguments:

v: The vector to process, it will be changed in-place.

tol: Tolerance value. Numbers smaller than this in magnitude will be replaced by zeros. Pass in zero to use the default tolerance. Must not be negative.

Returns:

Error code.

See also:

`igraph_vector_all_almost_e()` and `igraph_almost_equals()` to perform comparisons with relative tolerances.

igraph_vector_lex_cmp — Lexicographical comparison of two vectors (type-safe variant).

```
int igraph_vector_lex_cmp(
    const igraph_vector_t *lhs, const igraph_vector_t *rhs
);
```

If the elements of two vectors match but one is shorter, the shorter one comes first. Thus {1, 3} comes after {1, 2, 3}, but before {1, 3, 4}.

This function is typically used together with `igraph_vector_list_sort()`.

Arguments:

lhs: Pointer to the first vector.

rhs: Pointer to the second vector.

Returns:

-1 if *lhs* is lexicographically smaller, 0 if *lhs* and *rhs* are equal, else 1.

See also:

`igraph_vector_lex_cmp_untyped()` for an untyped variant of this function, or `igraph_vector_colex_cmp()` to compare vectors starting from the last element.

Time complexity: O(n), the number of elements in the smaller vector.

Example 7.2. File examples/simple/igraph_vector_int_list_sort.c

igraph_vector_lex_cmp_untyped — Lexicographical comparison of two vectors (non-type-safe).

```
int igraph_vector_lex_cmp_untyped(const void *lhs, const void *rhs);
```

If the elements of two vectors match but one is shorter, the shorter one comes first. Thus {1, 3} comes after {1, 2, 3}, but before {1, 3, 4}.

This function is typically used together with `igraph_vector_ptr_sort()`.

Arguments:

lhs: Pointer to a pointer to the first vector (interpreted as an `igraph_vector_t **`).

rhs: Pointer to a pointer to the second vector (interpreted as an `igraph_vector_t **`).

Returns:

-1 if *lhs* is lexicographically smaller, 0 if *lhs* and *rhs* are equal, else 1.

See also:

`igraph_vector_lex_cmp()` for a type-safe variant of this function, or `igraph_vector_colex_cmp_untyped()` to compare vectors starting from the last element.

Time complexity: O(n), the number of elements in the smaller vector.

`igraph_vector_colex_cmp` — Colexicographical comparison of two vectors.

```
int igraph_vector_colex_cmp(
    const igraph_vector_t *lhs, const igraph_vector_t *rhs
);
```

This comparison starts from the last element of both vectors and moves backward. If the elements of two vectors match but one is shorter, the shorter one comes first. Thus {1, 2} comes after {3, 2, 1}, but before {0, 1, 2}.

This function is typically used together with `igraph_vector_list_sort()`.

Arguments:

`lhs`: Pointer to a pointer to the first vector.

`rhs`: Pointer to a pointer to the second vector.

Returns:

-1 if *`lhs`* in reverse order is lexicographically smaller than the reverse of *`rhs`*, 0 if *`lhs`* and *`rhs`* are equal, else 1.

See also:

`igraph_vector_colex_cmp_untyped()` for an untyped variant of this function, or `igraph_vector_lex_cmp()` to compare vectors starting from the first element.

Time complexity: O(n), the number of elements in the smaller vector.

Example 7.3. File `examples/simple/igraph_vector_int_list_sort.c`

`igraph_vector_colex_cmp_untyped` — Colexicographical comparison of two vectors.

```
int igraph_vector_colex_cmp_untyped(const void *lhs, const void *rhs);
```

This comparison starts from the last element of both vectors and moves backward. If the elements of two vectors match but one is shorter, the shorter one comes first. Thus {1, 2} comes after {3, 2, 1}, but before {0, 1, 2}.

This function is typically used together with `igraph_vector_ptr_sort()`.

Arguments:

`lhs`: Pointer to a pointer to the first vector (interpreted as an `igraph_vector_t **`).

`rhs`: Pointer to a pointer to the second vector (interpreted as an `igraph_vector_t **`).

Returns:

-1 if *lhs* in reverse order is lexicographically smaller than the reverse of *rhs*, 0 if *lhs* and *rhs* are equal, else 1.

See also:

`igraph_vector_colex_cmp()` for a type-safe variant of this function, `igraph_vector_lex_cmp_untyped()` to compare vectors starting from the first element.

Time complexity: O(n), the number of elements in the smaller vector.

Finding minimum and maximum

`igraph_vector_min` — Smallest element of a vector.

```
igraph_real_t igraph_vector_min(const igraph_vector_t *v);
```

The vector must not be empty.

Arguments:

v: The input vector.

Returns:

The smallest element of *v*, or NaN if any element is NaN.

Time complexity: O(n), the number of elements.

`igraph_vector_max` — Largest element of a vector.

```
igraph_real_t igraph_vector_max(const igraph_vector_t *v);
```

The vector must not be empty.

Arguments:

v: The vector object.

Returns:

The maximum element of *v*, or NaN if any element is NaN.

Time complexity: O(n), the number of elements.

`igraph_vector_which_min` — Index of the smallest element.

```
igraph_int_t igraph_vector_which_min(const igraph_vector_t* v);
```

The vector must not be empty. If the smallest element is not unique, then the index of the first is returned. If the vector contains NaN values, the index of the first NaN value is returned.

Arguments:

v: The input vector.

Returns:

Index of the smallest element.

Time complexity: O(n), the number of elements.

`igraph_vector_which_max` — Gives the index of the maximum element of the vector.

```
igraph_int_t igraph_vector_which_max(const igraph_vector_t *v);
```

The vector must not be empty. If the largest element is not unique, then the index of the first is returned. If the vector contains NaN values, the index of the first NaN value is returned.

Arguments:

v: The vector object.

Returns:

The index of the first maximum element.

Time complexity: O(n), n is the size of the vector.

`igraph_vector_minmax` — Minimum and maximum elements of a vector.

```
void igraph_vector_minmax(const igraph_vector_t *v,
                          igraph_real_t *min, igraph_real_t *max);
```

Handy if you want to have both the smallest and largest element of a vector. The vector is only traversed once. The vector must be non-empty. If a vector contains at least one NaN, both `min` and `max` will be NaN.

Arguments:

v: The input vector. It must contain at least one element.

min: Pointer to a base type variable, the minimum is stored here.

max: Pointer to a base type variable, the maximum is stored here.

Time complexity: O(n), the number of elements.

`igraph_vector_which_minmax` — Index of the minimum and maximum elements.

```
void igraph_vector_which_minmax(const igraph_vector_t *v,
        igraph_int_t *which_min, igraph_int_t *which_max);
```

Handy if you need the indices of the smallest and largest elements. The vector is traversed only once. The vector must be non-empty. If the minimum or maximum is not unique, the index of the first

minimum or the first maximum is returned, respectively. If a vector contains at least one NaN, both `which_min` and `which_max` will point to the first NaN value.

Arguments:

v: The input vector. It must contain at least one element.

which_min: The index of the minimum element will be stored here.

which_max: The index of the maximum element will be stored here.

Time complexity: O(n), the number of elements.

Vector properties

`igraph_vector_empty` — Decides whether the size of the vector is zero.

```
igraph_bool_t igraph_vector_empty(const igraph_vector_t *v);
```

Arguments:

v: The vector object.

Returns:

True if the size of the vector is zero and false otherwise.

Time complexity: O(1).

`igraph_vector_size` — The size of the vector.

```
igraph_int_t igraph_vector_size(const igraph_vector_t *v);
```

Returns the number of elements stored in the vector.

Arguments:

v: The vector object

Returns:

The size of the vector.

Time complexity: O(1).

`igraph_vector_capacity` — Returns the allocated capacity of the vector.

```
igraph_int_t igraph_vector_capacity(const igraph_vector_t *v);
```

Note that this might be different from the size of the vector (as queried by `igraph_vector_size()`), and specifies how many elements the vector can hold, without reallocation.

Arguments:

v: Pointer to the (previously initialized) vector object to query.

Returns:

The allocated capacity.

See also:

`igraph_vector_size()`.

Time complexity: O(1).

`igraph_vector_sum` — Calculates the sum of the elements in the vector.

```
igraph_real_t igraph_vector_sum(const igraph_vector_t *v);
```

For the empty vector 0 is returned.

Arguments:

v: The vector object.

Returns:

The sum of the elements.

Time complexity: O(n), the size of the vector.

`igraph_vector_prod` — Calculates the product of the elements in the vector.

```
igraph_real_t igraph_vector_prod(const igraph_vector_t *v);
```

For the empty vector one (1) is returned.

Arguments:

v: The vector object.

Returns:

The product of the elements.

Time complexity: O(n), the size of the vector.

`igraph_vector_isininterval` — Checks if all elements of a vector are in the given interval.

```
igraph_bool_t igraph_vector_isininterval(const igraph_vector_t *v,
        igraph_real_t low,
```

```
        igraph_real_t high);
```

Arguments:

v: The vector object.

low: The lower limit of the interval (inclusive).

high: The higher limit of the interval (inclusive).

Returns:

True if the vector is empty or all vector elements are in the interval, false otherwise. If any element is NaN, it will return false.

Time complexity: O(n), the number of elements in the vector.

igraph_vector_maxdifference — The maximum absolute difference of *m1* and *m2*.

```
igraph_real_t igraph_vector_maxdifference(const igraph_vector_t *m1,
        const igraph_vector_t *m2);
```

The element with the largest absolute value in *m1* - *m2* is returned. Both vectors must be non-empty, but they not need to have the same length, the extra elements in the longer vector are ignored. If any value is NaN in the shorter vector, the result will be NaN.

Arguments:

m1: The first vector.

m2: The second vector.

Returns:

The maximum absolute difference of *m1* and *m2*.

Time complexity: O(n), the number of elements in the shorter vector.

igraph_vector_is_nan — Check for each element if it is NaN.

```
igraph_error_t igraph_vector_is_nan(const igraph_vector_t *v, igraph_vector_boo
```

Arguments:

v: The igraph_vector_t object to check.

is_nan: The resulting boolean vector indicating for each element whether it is NaN or not.

Returns:

Error code, IGRAPH_ENOMEM if there is not enough memory. Note that this function *never* returns an error if the vector *is_nan* will already be large enough.

Time complexity: O(n), the number of elements.

`igraph_vector_is_any_nan` — Check if any element is NaN.

```
igraph_bool_t igraph_vector_is_any_nan(const igraph_vector_t *v);
```

Arguments:

v: The igraph_vector_t object to check.

Returns:

True if any element is NaN, false otherwise.

Time complexity: O(n), the number of elements.

`igraph_vector_is_all_finite` — Check if all elements are finite.

```
igraph_bool_t igraph_vector_is_all_finite(const igraph_vector_t *v);
```

Arguments:

v: The igraph_vector_t object to check.

Returns:

True if none of the elements are infinite or NaN.

Time complexity: O(n), the number of elements.

Searching for elements

`igraph_vector_contains` — Linear search in a vector.

```
igraph_bool_t igraph_vector_contains(const igraph_vector_t *v,
        igraph_real_t what);
```

Check whether the supplied element is included in the vector, by linear search.

Arguments:

v: The input vector.

what: The element to look for.

Returns:

`true` if the element is found and `false` otherwise.

Time complexity: O(n), the length of the vector.

`igraph_vector_search` — Searches in a vector from a given position.

```
igraph_bool_t igraph_vector_search(const igraph_vector_t *v,
        igraph_int_t from, igraph_real_t what, igraph_int_t *pos);
```

The supplied element *what* is searched in vector *v*, starting from element index *from*. If found then the index of the first instance (after *from*) is stored in *pos*.

Arguments:

v: The input vector.

from: The index to start searching from. No range checking is performed.

what: The element to find.

pos: If not NULL then the index of the found element is stored here.

Returns:

Boolean, true if the element was found, false otherwise.

Time complexity: O(m), the number of elements to search, the length of the vector minus the *from* argument.

igraph_vector_binsearch — Finds an element by binary searching a sorted vector.

```
igraph_bool_t igraph_vector_binsearch(const igraph_vector_t *v,
        igraph_real_t what, igraph_int_t *pos);
```

It is assumed that the vector is sorted. If the specified element (*what*) is not in the vector, then the position of where it should be inserted (to keep the vector sorted) is returned. If the vector contains any NaN values, the returned value is undefined and *pos* may point to any position.

Arguments:

v: The igraph_vector_t object.

what: The element to search for.

pos: Pointer to an igraph_int_t. This is set to the position of an instance of *what* in the vector if it is present. If *v* does not contain *what* then *pos* is set to the position to which it should be inserted (to keep the vector sorted of course).

Returns:

True if *what* is found in the vector, false otherwise.

Time complexity: O(log(n)), n is the number of elements in *v*.

igraph_vector_binsearch_slice — Finds an element by binary searching a sorted slice of a vector.

```
igraph_bool_t igraph_vector_binsearch_slice(const igraph_vector_t *v,
        igraph_real_t what, igraph_int_t *pos, igraph_int_t start, igraph_int_t
```

It is assumed that the indicated slice of the vector, from *start* to *end*, is sorted. If the specified element (*what*) is not in the slice of the vector, then the position of where it should be inserted (to keep the *slice* sorted) is returned. Note that this means that the returned index will point *inside* the slice (including its endpoints), but will not evaluate values *outside* the slice. If the indicated slice contains any NaN values, the returned value is undefined and pos may point to any position within the slice.

Arguments:

v: The igraph_vector_t object.

what: The element to search for.

pos: Pointer to an igraph_int_t. This is set to the position of an instance of *what* in the slice of the vector if it is present. If *v* does not contain *what* then *pos* is set to the position to which it should be inserted (to keep the vector sorted).

start: The start position of the slice to search (inclusive).

end: The end position of the slice to search (exclusive).

Returns:

True if *what* is found in the vector, false otherwise.

Time complexity: O(log(n)), n is the number of elements in the slice of *v*, i.e. *end* - *start*.

igraph_vector_contains_sorted — Binary search in a sorted vector.

```
igraph_bool_t igraph_vector_contains_sorted(const igraph_vector_t *v, igraph_re
```

It is assumed that the vector is sorted.

Arguments:

v: The igraph_vector_t object.

what: The element to search for.

Returns:

True if *what* is found in the vector, false otherwise.

Time complexity: O(log(n)), n is the number of elements in *v*.

Resizing operations

igraph_vector_clear — Removes all elements from a vector.

```
void igraph_vector_clear(igraph_vector_t* v);
```

This function simply sets the size of the vector to zero, it does not free any allocated memory. For that you have to call igraph_vector_destroy().

Arguments:

v: The vector object.

Time complexity: O(1).

igraph_vector_reserve — Reserves memory for a vector.

```
igraph_error_t igraph_vector_reserve(igraph_vector_t *v, igraph_int_t capacity)
```

igraph vectors are flexible, they can grow and shrink. Growing however occasionally needs the data in the vector to be copied. In order to avoid this, you can call this function to reserve space for future growth of the vector.

Note that this function does *not* change the size of the vector. Let us see a small example to clarify things: if you reserve space for 100 elements and the size of your vector was (and still is) 60, then you can surely add additional 40 elements to your vector before it will be copied.

Arguments:

v: The vector object.

capacity: The new *allocated* size of the vector.

Returns:

Error code: `IGRAPH_ENOMEM` if there is not enough memory.

Time complexity: operating system dependent, should be around O(n), n is the new allocated size of the vector.

igraph_vector_resize — Resize the vector.

```
igraph_error_t igraph_vector_resize(igraph_vector_t* v, igraph_int_t new_size);
```

Note that this function does not free any memory, just sets the size of the vector to the given one. It can on the other hand allocate more memory if the new size is larger than the previous one. In this case the newly appeared elements in the vector are *not* set to zero, they are uninitialized.

Arguments:

v: The vector object

new_size: The new size of the vector.

Returns:

Error code, `IGRAPH_ENOMEM` if there is not enough memory. Note that this function *never* returns an error if the vector is made smaller.

See also:

`igraph_vector_reserve()` for allocating memory for future extensions of a vector. `igraph_vector_resize_min()` for deallocating the unnneded memory for a vector.

Time complexity: O(1) if the new size is smaller, operating system dependent if it is larger. In the latter case it is usually around O(n), n is the new size of the vector.

`igraph_vector_resize_min` — Deallocate the unused memory of a vector.

```
void igraph_vector_resize_min(igraph_vector_t *v);
```

This function attempts to deallocate the unused reserved storage of a vector. If it succeeds, `igraph_vector_size()` and `igraph_vector_capacity()` will be the same. The data in the vector is always preserved, even if deallocation is not successful.

Arguments:

v: Pointer to an initialized vector.

See also:

`igraph_vector_resize()`, `igraph_vector_reserve()`.

Time complexity: operating system dependent, O(n) at worst.

`igraph_vector_push_back` — Appends one element to a vector.

```
igraph_error_t igraph_vector_push_back(igraph_vector_t *v, igraph_real_t e);
```

This function resizes the vector to be one element longer and sets the very last element in the vector to *e*.

Arguments:

v: The vector object.

e: The element to append to the vector.

Returns:

Error code: `IGRAPH_ENOMEM`: not enough memory.

Time complexity: operating system dependent. What is important is that a sequence of n subsequent calls to this function has time complexity O(n), even if there hadn't been any space reserved for the new elements by `igraph_vector_reserve()`. This is implemented by a trick similar to the C ++ vector class: each time more memory is allocated for a vector, the size of the additionally allocated memory is the same as the vector's current length. (We assume here that the time complexity of memory allocation is at most linear.)

`igraph_vector_pop_back` — Removes and returns the last element of a vector.

```
igraph_real_t igraph_vector_pop_back(igraph_vector_t *v);
```

It is an error to call this function with an empty vector.

Arguments:

v: The vector object.

Returns:

The removed last element.

Time complexity: O(1).

`igraph_vector_insert` — Inserts a single element into a vector.

```
igraph_error_t igraph_vector_insert(
        igraph_vector_t *v, igraph_int_t pos, igraph_real_t value);
```

Note that this function does not do range checking. Insertion will shift the elements from the position given to the end of the vector one position to the right, and the new element will be inserted in the empty space created at the given position. The size of the vector will increase by one.

Arguments:

v: The vector object.

pos: The position where the new element is to be inserted.

value: The new element to be inserted.

`igraph_vector_remove` — Removes a single element from a vector.

```
void igraph_vector_remove(igraph_vector_t *v, igraph_int_t elem);
```

Note that this function does not do range checking.

Arguments:

v: The vector object.

elem: The position of the element to remove.

Time complexity: O(n-elem), n is the number of elements in the vector.

`igraph_vector_remove_section` — Deletes a section from a vector.

```
void igraph_vector_remove_section(
        igraph_vector_t *v, igraph_int_t from, igraph_int_t to);
```

Arguments:

v: The vector object.

from: The position of the first element to remove.

to: The position of the first element *not* to remove.

Time complexity: O(n-from), n is the number of elements in the vector.

Complex vector operations

`igraph_vector_complex_real` — Gives the real part of a complex vector.

```
igraph_error_t igraph_vector_complex_real(const igraph_vector_complex_t *v,
                                          igraph_vector_t *real);
```

Arguments:

`v`: Pointer to a complex vector.

`real`: Pointer to an initialized vector. The result will be stored here.

Returns:

Error code.

Time complexity: O(n), n is the number of elements in the vector.

`igraph_vector_complex_imag` — Gives the imaginary part of a complex vector.

```
igraph_error_t igraph_vector_complex_imag(const igraph_vector_complex_t *v,
                                          igraph_vector_t *imag);
```

Arguments:

`v`: Pointer to a complex vector.

`imag`: Pointer to an initialized vector. The result will be stored here.

Returns:

Error code.

Time complexity: O(n), n is the number of elements in the vector.

`igraph_vector_complex_realimag` — Gives the real and imaginary parts of a complex vector.

```
igraph_error_t igraph_vector_complex_realimag(const igraph_vector_complex_t *v,
                                              igraph_vector_t *real,
                                              igraph_vector_t *imag);
```

Arguments:

`v`: Pointer to a complex vector.

`real`: Pointer to an initialized vector. The real part will be stored here.

`imag`: Pointer to an initialized vector. The imaginary part will be stored here.

Returns:

Error code.

Time complexity: O(n), n is the number of elements in the vector.

`igraph_vector_complex_create` — Creates a complex vector from a real and imaginary part.

```
igraph_error_t igraph_vector_complex_create(igraph_vector_complex_t *v,
                                    const igraph_vector_t *real,
                                    const igraph_vector_t *imag);
```

Arguments:

v: Pointer to an uninitialized complex vector.

real: Pointer to the real part of the complex vector.

imag: Pointer to the imaginary part of the complex vector.

Returns:

Error code.

Time complexity: O(n), n is the number of elements in the vector.

`igraph_vector_complex_create_polar` — Creates a complex matrix from a magnitude and an angle.

```
igraph_error_t igraph_vector_complex_create_polar(igraph_vector_complex_t *v,
                                        const igraph_vector_t *r,
                                        const igraph_vector_t *theta);
```

Arguments:

v: Pointer to an uninitialized complex vector.

r: Pointer to a real vector containing magnitudes.

theta: Pointer to a real vector containing arguments (phase angles).

Returns:

Error code.

Time complexity: O(n), n is the number of elements in the vector.

`igraph_vector_complex_all_almost_e` — Are all elements almost equal?

```
igraph_bool_t igraph_vector_complex_all_almost_e(const igraph_vector_complex_t
                                                 const igraph_vector_complex_t
```

```
                                                    igraph_real_t eps);
```

Checks if the elements of two complex vectors are equal within a relative tolerance.

Arguments:

lhs: The first vector.

rhs: The second vector.

eps: Relative tolerance, see `igraph_complex_almost_equals()` for details.

Returns:

True if the two vectors are almost equal, false if there is at least one differing element or if the vectors are not of the same size.

`igraph_vector_complex_zapsmall` — Replaces small elements of a complex vector by exact zeros.

```
igraph_error_t igraph_vector_complex_zapsmall(igraph_vector_complex_t *v, igraph
```

Similarly to `igraph_vector_zapsmall()`, small elements will be replaced by zeros. The operation is performed separately on the real and imaginary parts of the numbers. This way, complex numbers with a large real part and tiny imaginary part will effectively be transformed to real numbers. The default tolerance corresponds to two-thirds of the representable digits of igraph_real_t, i.e. `DBL_EPSILON^(2/3)` which is approximately `10^-10`.

Arguments:

v: The vector to process, it will be changed in-place.

tol: Tolerance value. Real and imaginary parts smaller than this in magnitude will be replaced by zeros. Pass in zero to use the default tolerance. Must not be negative.

Returns:

Error code.

See also:

`igraph_vector_complex_all_almost_e()` and `igraph_complex_almost_equals()` to perform comparisons with relative tolerances.

Sorting

`igraph_vector_sort` — Sorts the elements of the vector into ascending order.

```
void igraph_vector_sort(igraph_vector_t *v);
```

If the vector contains any NaN values, the resulting ordering of NaN values is undefined and may appear anywhere in the vector.

Arguments:

v: Pointer to an initialized vector object.

Time complexity: O(n log n) for n elements.

`igraph_vector_reverse_sort` — Sorts the elements of the vector into descending order.

```
void igraph_vector_reverse_sort(igraph_vector_t *v);
```

If the vector contains any NaN values, the resulting ordering of NaN values is undefined and may appear anywhere in the vector.

Arguments:

v: Pointer to an initialized vector object.

Time complexity: O(n log n) for n elements.

`igraph_vector_sort_ind` — Returns a permutation of indices that sorts a vector.

```
igraph_error_t igraph_vector_sort_ind(
        const igraph_vector_t *v,
        igraph_vector_int_t *inds,
        igraph_order_t order);
```

Takes an unsorted array `v` as input and computes an array of indices *`inds`* such that `v[ inds[i] ]`, with `i` increasing from 0, is an ordered array (either ascending or descending, depending on *`order`*). The order of indices for identical elements is not defined. If the vector contains any NaN values, the ordering of NaN values is undefined.

Arguments:

v: the array to be sorted

`inds`: the output array of indices. This must be initialized, but will be resized

`order`: whether the output array should be sorted in ascending or descending order. Use `IGRAPH_ASCENDING` for ascending and `IGRAPH_DESCENDING` for descending order.

Returns:

Error code.

This routine uses igraph's built-in qsort routine. Algorithm: 1) create an array of pointers to the elements of v. 2) Pass this array to qsort. 3) after sorting the difference between the pointer value and the first pointer value gives its original position in the array. Use this to set the values of inds.

Set operations on sorted vectors

`igraph_vector_intersect_sorted` — Set intersection of two sorted vectors.

```
igraph_error_t igraph_vector_intersect_sorted(const igraph_vector_t *v1,
        const igraph_vector_t *v2, igraph_vector_t *result);
```

The elements that are contained in both vectors are stored in the result vector. All three vectors must be initialized.

For similar-size vectors, this function uses a straightforward linear scan. When the vector sizes differ substantially, it uses the set intersection method of Ricardo Baeza-Yates, which takes logarithmic time in the length of the larger vector.

The algorithm keeps the multiplicities of the elements: if an element appears `k1` times in the first vector and `k2` times in the second, the result will include that element `min(k1, k2)` times.

Reference:

Baeza-Yates R: A fast set intersection algorithm for sorted sequences. In: Lecture Notes in Computer Science, vol. 3109/2004, pp. 400--408, 2004. Springer Berlin/Heidelberg. https://doi.org/10.1007/978-3-540-27801-6_30

Arguments:

v1: The first vector

v2: The second vector

result: The result vector, which will also be sorted.

Returns:

Error code.

Time complexity: O(m log(n)) where m is the size of the smaller vector and n is the size of the larger one.

`igraph_vector_intersection_size_sorted` — Intersection size of two sorted vectors.

```
igraph_int_t igraph_vector_intersection_size_sorted(
        const igraph_vector_t *v1,
        const igraph_vector_t *v2);
```

Counts elements that are present in both vectors. This is particularly useful for counting common neighbours of two vertices.

For similar-size vectors, this function uses a straightforward linear scan. When the vector sizes differ substantially, it uses the set intersection method of Ricardo Baeza-Yates, which takes logarithmic time in the length of the larger vector.

The algorithm keeps the multiplicities of the elements: if an element appears `k1` times in the first vector and `k2` times in the second, the result will include that element `min(k1, k2)` times.

Reference:

Baeza-Yates R: A fast set intersection algorithm for sorted sequences. In: Lecture Notes in Computer Science, vol. 3109/2004, pp. 400--408, 2004. Springer Berlin/Heidelberg. https://doi.org/10.1007/978-3-540-27801-6_30

Arguments:

v1: The first vector

v2: The second vector

Returns:

The number of common elements.

Time complexity: O(m log(n)) where m is the size of the smaller vector and n is the size of the larger one.

`igraph_vector_difference_sorted` — Set difference of two sorted vectors.

```
igraph_error_t igraph_vector_difference_sorted(const igraph_vector_t *v1,
        const igraph_vector_t *v2, igraph_vector_t *result);
```

The elements that are contained in only the first vector but not the second are stored in the result vector. All three vectors must be initialized.

The algorithm keeps the multiplicities of the elements: if an element appears `k1` times in the first vector and `k2` times in the second, the result will include that element `max(0, k1-k2)` times.

Arguments:

v1: the first vector

v2: the second vector

result: the result vector

`igraph_vector_difference_and_intersection_sorted` — Simultaneous difference and intersection of two sorted vectors.

```
igraph_error_t igraph_vector_difference_and_intersection_sorted(
        const igraph_vector_t *v1, const igraph_vector_t *v2,
        igraph_vector_t *vdiff12, igraph_vector_t*vdiff21,
        igraph_vector_t*vint);
```

This function iterates over all the elements of the two input vectors and sorts them into three other vectors: elements that are in the first vector but not in the second, elements that are in the second vector but not in the first, and the intersection of the two vectors. The input vectors must be initialized. The output arguments can be `NULL`, but they must be initialized if they are not `NULL` and will be resized accordingly.

The multiplicities of the individual elements are treated consistently with `igraph_vector_difference_sorted()` and `igraph_vector_intersect_sorted()`: The algorithm keeps the multiplicities of the elements: if an element appears `k1` times in the first vector and `k2` times in the second, the intersection vector will include that element `min(k1, k2)` times, while the difference vectors will include that element `max(0, k1-k2)` and `max(0, k2-k1)` times, respectively.

Arguments:

v1: the first vector

v2: the second vector

vdiff12: output vector containing the elements that are in the first vector but not the second one, or NULL if not needed

vdiff21: output vector containing the elements that are in the second vector but not the first one, or NULL if not needed

vint: output vector containing the intersection, or NULL if not needed

Pointer vectors (igraph_vector_ptr_t)

The igraph_vector_ptr_t data type is very similar to the igraph_vector_t type, but it stores generic pointers instead of real numbers.

This type has the same space complexity as igraph_vector_t, and most implemented operations work the same way as for igraph_vector_t.

The same VECTOR macro used for ordinary vectors can be used for pointer vectors as well, please note that a typeless generic pointer will be provided by this macro and you may need to cast it to a specific pointer before starting to work with it.

Pointer vectors may have an associated item destructor function which takes a pointer and returns nothing. The item destructor will be called on each item in the pointer vector when it is destroyed by igraph_vector_ptr_destroy() or igraph_vector_ptr_destroy_all(), or when its elements are freed by igraph_vector_ptr_free_all(). Note that the semantics of an item destructor does not coincide with C++ destructors; for instance, when a pointer vector is resized to a smaller size, the extra items will *not* be destroyed automatically! Nevertheless, item destructors may become handy in many cases; for instance, a vector of graphs generated by some function can be destroyed with a single call to igraph_vector_ptr_destroy_all() if the item destructor is set to igraph_destroy().

igraph_vector_ptr_init — Initialize a pointer vector (constructor).

```
igraph_error_t igraph_vector_ptr_init(igraph_vector_ptr_t* v, igraph_int_t size
```

This is the constructor of the pointer vector data type. All pointer vectors constructed this way should be destroyed via calling igraph_vector_ptr_destroy().

Arguments:

v: Pointer to an uninitialized igraph_vector_ptr_t object, to be created.

size: Integer, the size of the pointer vector.

Returns:

Error code: IGRAPH_ENOMEM if out of memory

Time complexity: operating system dependent, the amount of “time” required to allocate *size* elements.

igraph_vector_ptr_init_copy — Initializes a pointer vector from another one (constructor).

```
igraph_error_t igraph_vector_ptr_init_copy(igraph_vector_ptr_t *to, const igraph
```

This function creates a pointer vector by copying another one. This is shallow copy, only the pointers in the vector will be copied.

It is potentially dangerous to copy a pointer vector with an associated item destructor. The copied vector will inherit the item destructor, which may cause problems when both vectors are destroyed as the items might get destroyed twice. Make sure you know what you are doing when copying a pointer vector with an item destructor, or unset the item destructor on one of the vectors later.

Arguments:

to: Pointer to an uninitialized pointer vector object.

from: A pointer vector object.

Returns:

Error code: `IGRAPH_ENOMEM` if out of memory

Time complexity: O(n) if allocating memory for n elements can be done in O(n) time.

`igraph_vector_ptr_destroy` — Destroys a pointer vector.

```
void igraph_vector_ptr_destroy(igraph_vector_ptr_t* v);
```

The destructor for pointer vectors.

Arguments:

v: Pointer to the pointer vector to destroy.

Time complexity: operating system dependent, the “time” required to deallocate O(n) bytes, n is the number of elements allocated for the pointer vector (not necessarily the number of elements in the vector).

`igraph_vector_ptr_free_all` — Frees all the elements of a pointer vector.

```
void igraph_vector_ptr_free_all(igraph_vector_ptr_t* v);
```

If an item destructor is set for this pointer vector, this function will first call the destructor on all elements of the vector and then free all the elements using `igraph_free()`. If an item destructor is not set, the elements will simply be freed.

Arguments:

v: Pointer to the pointer vector whose elements will be freed.

Time complexity: operating system dependent, the “time” required to call the destructor n times and then deallocate O(n) pointers, each pointing to a memory area of arbitrary size. n is the number of elements in the pointer vector.

`igraph_vector_ptr_destroy_all` — Frees all the elements and destroys the pointer vector.

```
void igraph_vector_ptr_destroy_all(igraph_vector_ptr_t* v);
```

This function is equivalent to `igraph_vector_ptr_free_all()` followed by `igraph_vector_ptr_destroy()`.

Arguments:

v: Pointer to the pointer vector to destroy.

Time complexity: operating system dependent, the "time" required to deallocate O(n) pointers, each pointing to a memory area of arbitrary size, plus the "time" required to deallocate O(n) bytes, n being the number of elements allocated for the pointer vector (not necessarily the number of elements in the vector).

`igraph_vector_ptr_size` — Gives the number of elements in the pointer vector.

```
igraph_int_t igraph_vector_ptr_size(const igraph_vector_ptr_t* v);
```

Arguments:

v: The pointer vector object.

Returns:

The size of the object, i.e. the number of pointers stored.

Time complexity: O(1).

`igraph_vector_ptr_capacity` — Returns the allocated capacity of the pointer vector.

```
igraph_int_t igraph_vector_ptr_capacity(const igraph_vector_ptr_t* v);
```

Arguments:

v: The pointer vector object.

Returns:

The allocated capacity.

Time complexity: O(1).

`igraph_vector_ptr_clear` — Removes all elements from a pointer vector.

```
void igraph_vector_ptr_clear(igraph_vector_ptr_t* v);
```

This function resizes a pointer to vector to zero length. Note that the pointed objects are *not* deallocated, you should call `igraph_free()` on them, or make sure that their allocated memory is freed in

some other way, you'll get memory leaks otherwise. If you have set up an item destructor earlier, the destructor will be called on every element.

Note that the current implementation of this function does *not* deallocate the memory required for storing the pointers, so making a pointer vector smaller this way does not give back any memory. This behavior might change in the future.

Arguments:

v: The pointer vector to clear.

Time complexity: O(1).

igraph_vector_ptr_reserve — Reserves memory for a pointer vector for later use.

```
igraph_error_t igraph_vector_ptr_reserve(igraph_vector_ptr_t* v, igraph_int_t ca
```

Returns:

Error code.

igraph_vector_ptr_resize — Resizes a pointer vector.

```
igraph_error_t igraph_vector_ptr_resize(igraph_vector_ptr_t* v, igraph_int_t new
```

Note that if a vector is made smaller the pointed object are not deallocated by this function and the item destructor is not called on the extra elements.

Arguments:

v: A pointer vector.

newsize: The new size of the pointer vector.

Returns:

Error code.

Time complexity: O(1) if the vector if made smaller. Operating system dependent otherwise, the amount of “time” needed to allocate the memory for the vector elements.

igraph_vector_ptr_resize_min — Deallocate the unused memory of a pointer vector.

```
void igraph_vector_ptr_resize_min(igraph_vector_ptr_t* v);
```

This function attempts to deallocate the unused reserved storage of a pointer vector. If it succeeds, `igraph_vector_ptr_size()` and `igraph_vector_ptr_capacity()` will be the same. The data in the pointer vector is always preserved, even if deallocation is not successful.

Arguments:

v: Pointer to an initialized pointer vector.

See also:

`igraph_vector_ptr_resize()`, `igraph_vector_ptr_reserve()`.

Time complexity: operating system dependent, O(n) at worst.

`igraph_vector_ptr_push_back` — Appends an element to the back of a pointer vector.

```
igraph_error_t igraph_vector_ptr_push_back(igraph_vector_ptr_t* v, void* e);
```

Arguments:

v: The pointer vector.

e: The new element to include in the pointer vector.

Returns:

Error code.

See also:

`igraph_vector_push_back()` for the corresponding operation of the ordinary vector type.

Time complexity: O(1) or O(n), n is the number of elements in the vector. The pointer vector implementation ensures that n subsequent push_back operations need O(n) time to complete.

`igraph_vector_ptr_pop_back` — Removes and returns the last element of a pointer vector.

```
void *igraph_vector_ptr_pop_back(igraph_vector_ptr_t *v);
```

It is an error to call this function with an empty vector.

Arguments:

v: The pointer vector.

Returns:

The removed last element.

Time complexity: O(1).

`igraph_vector_ptr_insert` — Inserts a single element into a pointer vector.

```
igraph_error_t igraph_vector_ptr_insert(igraph_vector_ptr_t* v, igraph_int_t pos
```

Note that this function does not do range checking. Insertion will shift the elements from the position given to the end of the vector one position to the right, and the new element will be inserted in the empty space created at the given position. The size of the vector will increase by one.

Arguments:

v: The pointer vector object.

pos: The position where the new element is inserted.

e: The inserted element

igraph_vector_ptr_get — Access an element of a pointer vector.

```
void *igraph_vector_ptr_get(const igraph_vector_ptr_t* v, igraph_int_t pos);
```

Arguments:

v: Pointer to a pointer vector.

pos: The index of the pointer to return.

Returns:

The pointer at *pos* position.

Time complexity: O(1).

igraph_vector_ptr_set — Assign to an element of a pointer vector.

```
void igraph_vector_ptr_set(igraph_vector_ptr_t* v, igraph_int_t pos, void* valu
```

Arguments:

v: Pointer to a pointer vector.

pos: The index of the pointer to update.

value: The new pointer to set in the vector.

Time complexity: O(1).

igraph_vector_ptr_sort — Sorts the pointer vector based on an external comparison function.

```
void igraph_vector_ptr_sort(igraph_vector_ptr_t *v, int (*compar)(const void*, c
```

Sometimes it is necessary to sort the pointers in the vector based on the property of the element being referenced by the pointer. This function allows us to sort the vector based on an arbitrary external comparison function which accepts two void * pointers `p1` and `p2` and returns an integer less than,

equal to or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. `p1` and `p2` will point to the pointer in the vector, so they have to be double-dereferenced if one wants to get access to the underlying object the address of which is stored in `v`.

Arguments:

v: The pointer vector to be sorted.

compar: A qsort-compatible comparison function. It must take pointers to the elements of the pointer vector. For example, if the pointer vector contains `igraph_vector_t *` pointers, then the comparison function must interpret its arguments as `igraph_vector_t **`.

`igraph_vector_ptr_sort_ind` — Returns a permutation of indices that sorts a vector of pointers.

```
igraph_error_t igraph_vector_ptr_sort_ind(igraph_vector_ptr_t *v,
        igraph_vector_int_t *inds, cmp_t *cmp);
```

Takes an unsorted array `v` as input and computes an array of indices inds such that v[inds[i]], with i increasing from 0, is an ordered array (either ascending or descending, depending on \v order). The order of indices for identical elements is not defined.

Arguments:

v: the array to be sorted

inds: the output array of indices. This must be initialized, but will be resized

cmp: a comparator function that takes two elements of the pointer vector being sorted (these are constant pointers on their own) and returns a negative value if the item *"pointed* to" by the first pointer is smaller than the item *"pointed* to" by the second pointer, a positive value if it is larger, or zero if the two items are equal

Returns:

Error code.

This routine uses the C library qsort routine. Algorithm: 1) create an array of pointers to the elements of v. 2) Pass this array to qsort. 3) after sorting the difference between the pointer value and the first pointer value gives its original position in the array. Use this to set the values of inds.

`igraph_vector_ptr_permute` — Permutes the elements of a pointer vector in place according to an index vector.

```
igraph_error_t igraph_vector_ptr_permute(igraph_vector_ptr_t* v, const igraph_v
```

This function takes a vector `v` and a corresponding index vector `ind`, and permutes the elements of `v` such that `v`[ind[i]] is moved to become `v`[i] after the function is executed.

It is an error to call this function with an index vector that does not represent a valid permutation. Each element in the index vector must be between 0 and the length of the vector minus one (inclusive), and each such element must appear only once. The function does not attempt to validate the index vector.

The index vector that this function takes is compatible with the index vector returned from `igraph_vector_ptr_sort_ind()`; passing in the index vector from `igraph_vector_ptr_sort_ind()` will sort the original vector.

As a special case, this function allows the index vector to be *shorter* than the vector being permuted, in which case the elements whose indices do not occur in the index vector will be removed from the vector.

Arguments:

v: the vector to permute

ind: the index vector

Returns:

Error code: `IGRAPH_ENOMEM` if there is not enough memory.

Time complexity: O(n), the size of the vector.

`igraph_vector_ptr_get_item_destructor` — Gets the current item destructor for this pointer vector.

```
igraph_finally_func_t* igraph_vector_ptr_get_item_destructor(const igraph_vecto
```

The item destructor is a function which will be called on every non-null pointer stored in this vector when `igraph_vector_ptr_destroy()`, igraph_vector_ptr_destroy_all() or `igraph_vector_ptr_free_all()` is called.

Returns:

The current item destructor.

Time complexity: O(1).

`igraph_vector_ptr_set_item_destructor` — Sets the item destructor for this pointer vector.

```
igraph_finally_func_t* igraph_vector_ptr_set_item_destructor(
    igraph_vector_ptr_t *v, igraph_finally_func_t *func);
```

The item destructor is a function which will be called on every non-null pointer stored in this vector when `igraph_vector_ptr_destroy()`, igraph_vector_ptr_destroy_all() or `igraph_vector_ptr_free_all()` is called.

Returns:

The old item destructor.

Time complexity: O(1).

`IGRAPH_VECTOR_PTR_SET_ITEM_DESTRUCTOR` — Sets the item destructor for this pointer vector (macro version).

```
#define IGRAPH_VECTOR_PTR_SET_ITEM_DESTRUCTOR(v, func)
```

This macro is expanded to `igraph_vector_ptr_set_item_destructor()`, the only difference is that the second argument is automatically cast to an `igraph_finally_func_t*`. The cast is necessary in most cases as the destructor functions we use (such as `igraph_vector_destroy()`) take a pointer to some concrete igraph data type, while `igraph_finally_func_t` expects `void*`

Matrices

About igraph_matrix_t objects

This type is just an interface to igraph_vector_t.

The igraph_matrix_t type usually stores n elements in O(n) space, but not always. See the documentation of the vector type.

Matrix constructors and destructors

`igraph_matrix_init` — Initializes a matrix.

```
igraph_error_t igraph_matrix_init(
        igraph_matrix_t *m, igraph_int_t nrow, igraph_int_t ncol);
```

Every matrix needs to be initialized before using it. This is done by calling this function. A matrix has to be destroyed if it is not needed any more; see `igraph_matrix_destroy()`.

Arguments:

m: Pointer to a not yet initialized matrix object to be initialized.

nrow: The number of rows in the matrix.

ncol: The number of columns in the matrix.

Returns:

Error code.

Time complexity: usually O(n), n is the number of elements in the matrix.

`igraph_matrix_init_array` — Initializes a matrix from an ordinary C array (constructor).

```
igraph_error_t igraph_matrix_init_array(
        igraph_matrix_t *m, const igraph_real_t *data,
        igraph_int_t nrow, igraph_int_t ncol,
        igraph_matrix_storage_t storage);
```

The array is assumed to store the matrix data contiguously, either in a column-major or row-major format. In other words, *data* may store concatenated matrix columns or concatenated matrix rows. Constructing a matrix from column-major data is faster, as this is igraph's native storage format.

Arguments:

v: Pointer to an uninitialized matrix object.

data: A regular C array, storing the elements of the matrix in column-major order, i.e. the elements of the first column are stored first, followed by the second column and so on.

nrow: The number of rows in the matrix.

ncol: The number of columns in the matrix.

storage: IGRAPH_ROW_MAJOR if the array is in row-major format, IGRAPH_COLUMN_MAJOR if the array is in column-major format.

Returns:

Error code: IGRAPH_ENOMEM if there is not enough memory.

Time complexity: operating system specific, usually O(*nrow ncol*).

igraph_matrix_init_copy — Copies a matrix.

```
igraph_error_t igraph_matrix_init_copy(igraph_matrix_t *to, const igraph_matrix
```

Creates a matrix object by copying from an existing matrix.

Arguments:

to: Pointer to an uninitialized matrix object.

from: The initialized matrix object to copy.

Returns:

Error code, IGRAPH_ENOMEM if there isn't enough memory to allocate the new matrix.

Time complexity: O(n), the number of elements in the matrix.

igraph_matrix_destroy — Destroys a matrix object.

```
void igraph_matrix_destroy(igraph_matrix_t *m);
```

This function frees all the memory allocated for a matrix object. The destroyed object needs to be reinitialized before using it again.

Arguments:

m: The matrix to destroy.

Time complexity: operating system dependent.

Initializing elements

igraph_matrix_null — Sets all elements in a matrix to zero.

```
void igraph_matrix_null(igraph_matrix_t *m);
```

Arguments:

m: Pointer to an initialized matrix object.

Time complexity: O(n), n is the number of elements in the matrix.

`igraph_matrix_fill` — Fill with an element.

```
void igraph_matrix_fill(igraph_matrix_t *m, igraph_real_t e);
```

Set the matrix to a constant matrix.

Arguments:

m: The input matrix.

e: The element to set.

Time complexity: O(mn), the number of elements.

Accessing elements of a matrix

`MATRIX` — Accessing an element of a matrix.

```
#define MATRIX(m,i,j)
```

Note that there are no range checks right now. This functionality might be redefined as a proper function later.

Arguments:

m: The matrix object.

i: The index of the row, starting with zero.

j: The index of the column, starting with zero.

Time complexity: O(1).

`igraph_matrix_get` — Extract an element from a matrix.

```
igraph_real_t igraph_matrix_get(const igraph_matrix_t *m,
                                igraph_int_t row, igraph_int_t col);
```

Use this if you need a function for some reason and cannot use the `MATRIX` macro. Note that no range checking is performed.

Arguments:

m: The input matrix.

row: The row index.

col: The column index.

Returns:

The element in the given row and column.

Time complexity: O(1).

`igraph_matrix_get_ptr` — Pointer to an element of a matrix.

```
igraph_real_t* igraph_matrix_get_ptr(const igraph_matrix_t *m,
                                     igraph_int_t row, igraph_int_t col);
```

The function returns a pointer to an element. No range checking is performed.

Arguments:

m: The input matrix.

row: The row index.

col: The column index.

Returns:

Pointer to the element in the given row and column.

Time complexity: O(1).

`igraph_matrix_set` — Set an element.

```
void igraph_matrix_set(
        igraph_matrix_t* m, igraph_int_t row, igraph_int_t col,
        igraph_real_t value);
```

Set an element of a matrix. No range checking is performed.

Arguments:

m: The input matrix.

row: The row index.

col: The column index.

value: The new value of the element.

Time complexity: O(1).

Matrix views

`igraph_matrix_view` — Creates a matrix view into an existing array.

```
igraph_matrix_t igraph_matrix_view(
        const igraph_real_t *data,
        igraph_int_t nrow, igraph_int_t ncol);
```

This function lets you treat an existing C array as a matrix. The elements of the matrix are assumed to be stored in column-major order in the array, i.e. the elements of the first column are stored first, followed by the second column and so on.

Since this function creates a view into an existing array, you must *not* destroy the igraph_matrix_t object when you are done with it. Similarly, you must *not* call any function on it that may attempt to modify the size of the matrix. Modifying an element in the matrix will modify the underlying array as the two share the same memory block.

Arguments:

data: The raw array that the matrix provides a view into.

nrow: The number of rows in the matrix.

ncol: The number of columns in the matrix.

Returns:

The matrix object providing the view into the array.

Time complexity: O(1).

igraph_matrix_view_from_vector — Creates a matrix view that treats an existing vector as a matrix.

```
igraph_matrix_t igraph_matrix_view_from_vector(
    const igraph_vector_t *v,
    igraph_int_t nrow
);
```

This function lets you treat an existing igraph vector as a matrix. The elements of the matrix are assumed to be stored in column-major order in the vector, i.e. the elements of the first column are stored first, followed by the second column and so on.

Since this function creates a view into an existing vector, you must *not* destroy the `igraph_matrix_t` object when you are done with it. Similarly, you must *not* call any function on it that may attempt to modify the size of the vector. Modifying an element in the matrix will modify the underlying vector as the two share the same memory block.

Additionally, you must *not* attempt to grow the underlying vector by any vector operation as that may result in a re-allocation of the backing memory block of the vector, and the matrix view will not be informed about the re-allocation so it will point to an invalid memory area afterwards.

Arguments:

v: The vector that the matrix will provide a view into.

nrow: The number of rows in the matrix. The number of columns will be derived implicitly from the size of the vector. If the number of items in the vector is not divisible by the number of rows, the last few elements of the vector will not be covered by the view.

Returns:

The matrix object providing the view into the vector.

Time complexity: O(1).

Copying matrices

igraph_matrix_copy_to — Copies a matrix to a regular C array.

```
void igraph_matrix_copy_to(const igraph_matrix_t *m, igraph_real_t *to, igraph_
```

The C array should be of sufficient size; there are (of course) no range checks.

Arguments:

m: Pointer to an initialized matrix object.

to: Pointer to a C array; the place to copy the data to.

storage: IGRAPH_ROW_MAJOR to write the data in row-major format, IGRAPH_COLUMN_MAJOR to write it in column-major format. Currently igraph uses column-major storage internally, thus IGRAPH_COLUMN_MAJOR is much faster.

Returns:

Error code.

Time complexity: O(n), n is the number of elements in the matrix.

igraph_matrix_update — Update from another matrix.

```
igraph_error_t igraph_matrix_update(igraph_matrix_t *to,
                                    const igraph_matrix_t *from);
```

This function replicates *from* in the matrix *to*. Note that *to* must be already initialized.

Arguments:

to: The result matrix.

from: The matrix to replicate; it is left unchanged.

Returns:

Error code.

Time complexity: O(mn), the number of elements.

igraph_matrix_swap — Swap two matrices.

```
void igraph_matrix_swap(igraph_matrix_t *m1, igraph_matrix_t *m2);
```

The contents of the two matrices will be swapped.

Arguments:

m1: The first matrix.

m2: The second matrix.

Time complexity: O(1).

Operations on rows and columns

igraph_matrix_get_row — Extract a row.

```
igraph_error_t igraph_matrix_get_row(const igraph_matrix_t *m,
                                     igraph_vector_t *res, igraph_int_t index);
```

Extract a row from a matrix and return it as a vector.

Arguments:

m: The input matrix.

res: Pointer to an initialized vector; it will be resized if needed.

index: The index of the row to select.

Returns:

Error code.

Time complexity: O(n), the number of columns in the matrix.

igraph_matrix_get_col — Select a column.

```
igraph_error_t igraph_matrix_get_col(const igraph_matrix_t *m,
                                     igraph_vector_t *res,
                                     igraph_int_t index);
```

Extract a column of a matrix and return it as a vector.

Arguments:

m: The input matrix.

res: The result will we stored in this vector. It should be initialized and will be resized as needed.

index: The index of the column to select.

Returns:

Error code.

Time complexity: O(n), the number of rows in the matrix.

igraph_matrix_set_row — Set a row from a vector.

```
igraph_error_t igraph_matrix_set_row(igraph_matrix_t *m,
                                     const igraph_vector_t *v, igraph_int_t ind
```

Sets the elements of a row with the given vector. This has the effect of setting row `index` to have the elements in the vector `v`. The length of the vector and the number of columns in the matrix must match, otherwise an error is triggered.

Arguments:

m: The input matrix.

v: The vector containing the new elements of the row.

index: Index of the row to set.

Returns:

Error code.

Time complexity: O(n), the number of columns in the matrix.

`igraph_matrix_set_col` — Set a column from a vector.

```
igraph_error_t igraph_matrix_set_col(igraph_matrix_t *m,
                                     const igraph_vector_t *v, igraph_int_t inde
```

Sets the elements of a column with the given vector. In effect, column `index` will be set with elements from the vector `v`. The length of the vector and the number of rows in the matrix must match, otherwise an error is triggered.

Arguments:

`m`: The input matrix.

`v`: The vector containing the new elements of the column.

`index`: Index of the column to set.

Returns:

Error code.

Time complexity: O(m), the number of rows in the matrix.

`igraph_matrix_swap_rows` — Swap two rows.

```
igraph_error_t igraph_matrix_swap_rows(igraph_matrix_t *m,
                                       igraph_int_t i, igraph_int_t j);
```

Swap two rows in the matrix.

Arguments:

`m`: The input matrix.

`i`: The index of the first row.

`j`: The index of the second row.

Returns:

Error code.

Time complexity: O(n), the number of columns.

`igraph_matrix_swap_cols` — Swap two columns.

```
igraph_error_t igraph_matrix_swap_cols(igraph_matrix_t *m,
                                       igraph_int_t i, igraph_int_t j);
```

Swap two columns in the matrix.

Arguments:

m: The input matrix.

i: The index of the first column.

j: The index of the second column.

Returns:

Error code.

Time complexity: O(m), the number of rows.

igraph_matrix_select_rows — Select some rows of a matrix.

```
igraph_error_t igraph_matrix_select_rows(const igraph_matrix_t *m,
        igraph_matrix_t *res,
        const igraph_vector_int_t *rows);
```

This function selects some rows of a matrix and returns them in a new matrix. The result matrix should be initialized before calling the function.

Arguments:

m: The input matrix.

res: The result matrix. It should be initialized and will be resized as needed.

rows: Vector; it contains the row indices (starting with zero) to extract. Note that no range checking is performed.

Returns:

Error code.

Time complexity: O(nm), n is the number of rows, m the number of columns of the result matrix.

igraph_matrix_select_cols — Select some columns of a matrix.

```
igraph_error_t igraph_matrix_select_cols(const igraph_matrix_t *m,
        igraph_matrix_t *res,
        const igraph_vector_int_t *cols);
```

This function selects some columns of a matrix and returns them in a new matrix. The result matrix should be initialized before calling the function.

Arguments:

m: The input matrix.

res: The result matrix. It should be initialized and will be resized as needed.

cols: Vector; it contains the column indices (starting with zero) to extract. Note that no range checking is performed.

Returns:

Error code.

Time complexity: O(nm), n is the number of rows, m the number of columns of the result matrix.

`igraph_matrix_select_rows_cols` — Select some rows and columns of a matrix.

```
igraph_error_t igraph_matrix_select_rows_cols(const igraph_matrix_t *m,
        igraph_matrix_t *res,
        const igraph_vector_int_t *rows,
        const igraph_vector_int_t *cols);
```

This function selects some rows and columns of a matrix and returns them in a new matrix. The result matrix should be initialized before calling the function.

Arguments:

m: The input matrix.

res: The result matrix. It should be initialized and will be resized as needed.

rows: Vector; it contains the row indices (starting with zero) to extract. Note that no range checking is performed.

cols: Vector; it contains the column indices (starting with zero) to extract. Note that no range checking is performed.

Returns:

Error code.

Time complexity: O(nm), n is the number of rows, m the number of columns of the result matrix.

Matrix operations

`igraph_matrix_add_constant` — Add a constant to every element.

```
void igraph_matrix_add_constant(igraph_matrix_t *m, igraph_real_t plus);
```

Arguments:

m: The input matrix.

plud: The constant to add.

Time complexity: O(mn), the number of elements.

`igraph_matrix_scale` — Multiplies each element of the matrix by a constant.

```
void igraph_matrix_scale(igraph_matrix_t *m, igraph_real_t by);
```

Arguments:

m: The matrix.

by: The constant.

Added in version 0.2.

Time complexity: O(n), the number of elements in the matrix.

igraph_matrix_add — Add two matrices.

```
igraph_error_t igraph_matrix_add(igraph_matrix_t *m1,
                                 const igraph_matrix_t *m2);
```

Add *m2* to *m1*, and store the result in *m1*. The dimensions of the matrices must match.

Arguments:

m1: The first matrix; the result will be stored here.

m2: The second matrix; it is left unchanged.

Returns:

Error code.

Time complexity: O(mn), the number of elements.

igraph_matrix_sub — Difference of two matrices.

```
igraph_error_t igraph_matrix_sub(igraph_matrix_t *m1,
                                 const igraph_matrix_t *m2);
```

Subtract *m2* from *m1* and store the result in *m1*. The dimensions of the two matrices must match.

Arguments:

m1: The first matrix; the result is stored here.

m2: The second matrix; it is left unchanged.

Returns:

Error code.

Time complexity: O(mn), the number of elements.

igraph_matrix_mul_elements — Elementwise matrix multiplication.

```
igraph_error_t igraph_matrix_mul_elements(igraph_matrix_t *m1,
```

```
        const igraph_matrix_t *m2);
```

Multiply *m1* by *m2* elementwise and store the result in *m1*. The dimensions of the two matrices must match.

Arguments:

m1: The first matrix; the result is stored here.

m2: The second matrix; it is left unchanged.

Returns:

Error code.

Time complexity: O(mn), the number of elements.

igraph_matrix_div_elements — Elementwise division.

```
igraph_error_t igraph_matrix_div_elements(igraph_matrix_t *m1,
        const igraph_matrix_t *m2);
```

Divide *m1* by *m2* elementwise and store the result in *m1*. The dimensions of the two matrices must match.

Arguments:

m1: The dividend. The result is store here.

m2: The divisor. It is left unchanged.

Returns:

Error code.

Time complexity: O(mn), the number of elements.

igraph_matrix_sum — Sum of elements.

```
igraph_real_t igraph_matrix_sum(const igraph_matrix_t *m);
```

Returns the sum of the elements of a matrix.

Arguments:

m: The input matrix.

Returns:

The sum of the elements.

Time complexity: O(mn), the number of elements in the matrix.

igraph_matrix_prod — Product of all matrix elements.

```
igraph_real_t igraph_matrix_prod(const igraph_matrix_t *m);
```

Note that this function can result in overflow easily, even for not too big matrices. Overflow is not checked.

Arguments:

m: The input matrix.

Returns:

The product of the elements.

Time complexity: O(mn), the number of elements.

`igraph_matrix_rowsum` — Rowwise sum.

```
igraph_error_t igraph_matrix_rowsum(const igraph_matrix_t *m,
                                    igraph_vector_t *res);
```

Calculate the sum of the elements in each row.

Arguments:

m: The input matrix.

res: Pointer to an initialized vector; the result is stored here. It will be resized if necessary.

Returns:

Error code.

Time complexity: O(mn), the number of elements in the matrix.

`igraph_matrix_colsum` — Columnwise sum.

```
igraph_error_t igraph_matrix_colsum(const igraph_matrix_t *m,
                                    igraph_vector_t *res);
```

Calculate the sum of the elements in each column.

Arguments:

m: The input matrix.

res: Pointer to an initialized vector; the result is stored here. It will be resized if necessary.

Returns:

Error code.

Time complexity: O(mn), the number of elements in the matrix.

`igraph_matrix_transpose` — Transpose of a matrix.

```
igraph_error_t igraph_matrix_transpose(igraph_matrix_t *m);
```

Calculates the transpose of a matrix. When the matrix is non-square, this function reallocates the storage used for the matrix.

Arguments:

m: The input (and output) matrix.

Returns:

Error code.

Time complexity: O(mn), the number of elements in the matrix.

Matrix comparisons

igraph_matrix_all_e — Are all elements equal?

```
igraph_bool_t igraph_matrix_all_e(const igraph_matrix_t *lhs,
        const igraph_matrix_t *rhs);
```

Checks element-wise equality of two matrices. For matrices containing floating point values, consider using `igraph_matrix_all_almost_e()`.

Arguments:

lhs: The first matrix.

rhs: The second matrix.

Returns:

True if the elements in the *lhs* are all equal to the corresponding elements in *rhs*. Returns false if the dimensions of the matrices don't match.

Time complexity: O(nm), the size of the matrices.

igraph_matrix_all_almost_e — Are all elements almost equal?

```
igraph_bool_t igraph_matrix_all_almost_e(const igraph_matrix_t *lhs,
                                         const igraph_matrix_t *rhs,
                                         igraph_real_t eps);
```

Checks if the elements of two matrices are equal within a relative tolerance.

Arguments:

lhs: The first matrix.

rhs: The second matrix.

eps: Relative tolerance, see `igraph_almost_equals()` for details.

Returns:

True if the two matrices are almost equal, false if there is at least one differing element or if the matrices are not of the same dimensions.

igraph_matrix_all_l — Are all elements less?

```
igraph_bool_t igraph_matrix_all_l(const igraph_matrix_t *lhs,
        const igraph_matrix_t *rhs);
```

Arguments:

lhs: The first matrix.

rhs: The second matrix.

Returns:

True if the elements in the *lhs* are all less than the corresponding elements in *rhs*. Returns false if the dimensions of the matrices don't match.

Time complexity: O(nm), the size of the matrices.

igraph_matrix_all_g — Are all elements greater?

```
igraph_bool_t igraph_matrix_all_g(const igraph_matrix_t *lhs,
        const igraph_matrix_t *rhs);
```

Arguments:

lhs: The first matrix.

rhs: The second matrix.

Returns:

True if the elements in the *lhs* are all greater than the corresponding elements in *rhs*. Returns false if the dimensions of the matrices don't match.

Time complexity: O(nm), the size of the matrices.

igraph_matrix_all_le — Are all elements less or equal?

```
igraph_bool_t
igraph_matrix_all_le(const igraph_matrix_t *lhs,
                                const igraph_matrix_t *rhs);
```

Arguments:

lhs: The first matrix.

rhs: The second matrix.

Returns:

True if the elements in the *lhs* are all less than or equal to the corresponding elements in *rhs*. Returns false if the dimensions of the matrices don't match.

Time complexity: O(nm), the size of the matrices.

igraph_matrix_all_ge — Are all elements greater or equal?

```
igraph_bool_t
igraph_matrix_all_ge(const igraph_matrix_t *lhs,
                                 const igraph_matrix_t *rhs);
```

Arguments:

lhs: The first matrix.

rhs: The second matrix.

Returns:

True if the elements in the *lhs* are all greater than or equal to the corresponding elements in *rhs*. Returns false if the dimensions of the matrices don't match.

Time complexity: O(nm), the size of the matrices.

igraph_matrix_zapsmall — Replaces small elements of a matrix by exact zeros.

```
igraph_error_t igraph_matrix_zapsmall(igraph_matrix_t *m, igraph_real_t tol);
```

Matrix elements which are smaller in magnitude than the given absolute tolerance will be replaced by exact zeros. The default tolerance corresponds to two-thirds of the representable digits of igraph_real_t, i.e. `DBL_EPSILON^(2/3)` which is approximately `10^-10`.

Arguments:

m: The matrix to process, it will be changed in-place.

tol: Tolerance value. Numbers smaller than this in magnitude will be replaced by zeros. Pass in zero to use the default tolerance. Must not be negative.

Returns:

Error code.

See also:

`igraph_matrix_all_almost_e()` and `igraph_almost_equals()` to perform comparisons with relative tolerances.

Combining matrices

igraph_matrix_rbind — Combine two matrices rowwise.

```
igraph_error_t igraph_matrix_rbind(igraph_matrix_t *to,
                                   const igraph_matrix_t *from);
```

This function places the rows of *from* below the rows of to and stores the result in *to*. The number of columns in the two matrices must match.

Arguments:

to: The upper matrix; the result is also stored here.

from: The lower matrix. It is left unchanged.

Returns:

Error code.

Time complexity: O(mn), the number of elements in the newly created matrix.

igraph_matrix_cbind — Combine matrices columnwise.

```
igraph_error_t igraph_matrix_cbind(igraph_matrix_t *to,
                                   const igraph_matrix_t *from);
```

This function places the columns of *from* on the right of *to*, and stores the result in *to*.

Arguments:

to: The left matrix; the result is stored here too.

from: The right matrix. It is left unchanged.

Returns:

Error code.

Time complexity: O(mn), the number of elements on the new matrix.

Finding minimum and maximum

igraph_matrix_min — Smallest element of a matrix.

```
igraph_real_t igraph_matrix_min(const igraph_matrix_t *m);
```

The matrix must be non-empty.

Arguments:

m: The input matrix.

Returns:

The smallest element of *m*, or NaN if any element is NaN.

Time complexity: O(mn), the number of elements in the matrix.

igraph_matrix_max — Largest element of a matrix.

```
igraph_real_t igraph_matrix_max(const igraph_matrix_t *m);
```

If the matrix is empty, an arbitrary number is returned.

Arguments:

m: The matrix object.

Returns:

The maximum element of *m*, or NaN if any element is NaN.

Added in version 0.2.

Time complexity: O(mn), the number of elements in the matrix.

igraph_matrix_which_min — Indices of the smallest element.

```
void igraph_matrix_which_min(
        const igraph_matrix_t *m, igraph_int_t *i, igraph_int_t *j);
```

The matrix must be non-empty. If the smallest element is not unique, then the indices of the first such element are returned. If the matrix contains NaN values, the indices of the first NaN value are returned.

Arguments:

m: The matrix.

i: Pointer to an igraph_int_t. The row index of the minimum is stored here.

j: Pointer to an igraph_int_t. The column index of the minimum is stored here.

Time complexity: O(mn), the number of elements.

igraph_matrix_which_max — Indices of the largest element.

```
void igraph_matrix_which_max(
        const igraph_matrix_t *m, igraph_int_t *i, igraph_int_t *j);
```

The matrix must be non-empty. If the largest element is not unique, then the indices of the first such element are returned. If the matrix contains NaN values, the indices of the first NaN value are returned.

Arguments:

m: The matrix.

i: Pointer to an igraph_int_t. The row index of the maximum is stored here.

j: Pointer to an igraph_int_t. The column index of the maximum is stored here.

Time complexity: O(mn), the number of elements.

igraph_matrix_minmax — Minimum and maximum elements of a matrix.

```
void igraph_matrix_minmax(const igraph_matrix_t *m,
                                    igraph_real_t *min, igraph_real_t *max);
```

Handy if you want to have both the smallest and largest element of a matrix. The matrix is only traversed once. The matrix must be non-empty. If a matrix contains at least one NaN, both `min` and `max` will be NaN.

Arguments:

m: The input matrix. It must contain at least one element.

min: Pointer to a base type variable. The minimum is stored here.

max: Pointer to a base type variable. The maximum is stored here.

Time complexity: O(mn), the number of elements.

igraph_matrix_which_minmax — Indices of the minimum and maximum elements.

```
void igraph_matrix_which_minmax(const igraph_matrix_t *m,
        igraph_int_t *imin, igraph_int_t *jmin,
        igraph_int_t *imax, igraph_int_t *jmax);
```

Handy if you need the indices of the smallest and largest elements. The matrix is traversed only once. The matrix must be non-empty. If the minimum or maximum is not unique, the index of the first minimum or the first maximum is returned, respectively. If a matrix contains at least one NaN, both `which_min` and `which_max` will point to the first NaN value.

Arguments:

m: The input matrix.

imin: Pointer to an igraph_int_t, the row index of the minimum is stored here.

jmin: Pointer to an igraph_int_t, the column index of the minimum is stored here.

imax: Pointer to an igraph_int_t, the row index of the maximum is stored here.

jmax: Pointer to an igraph_int_t, the column index of the maximum is stored here.

Time complexity: O(mn), the number of elements.

Matrix properties

igraph_matrix_empty — Is the matrix empty?

```
igraph_bool_t igraph_matrix_empty(const igraph_matrix_t *m);
```

It is possible to have a matrix with zero rows or zero columns, or even both. This functions checks for these.

Arguments:

m: The input matrix.

Returns:

Boolean, `true` if the matrix contains zero elements, and `false` otherwise.

Time complexity: O(1).

`igraph_matrix_isnull` — Checks for a null matrix.

```
igraph_bool_t igraph_matrix_isnull(const igraph_matrix_t *m);
```

Checks whether all elements are zero.

Arguments:

m: The input matrix.

Returns:

Boolean, `true` is *m* contains only zeros and `false` otherwise.

Time complexity: O(mn), the number of elements.

`igraph_matrix_size` — The number of elements in a matrix.

```
igraph_int_t igraph_matrix_size(const igraph_matrix_t *m);
```

Arguments:

m: Pointer to an initialized matrix object.

Returns:

The size of the matrix.

Time complexity: O(1).

`igraph_matrix_capacity` — Returns the number of elements allocated for a matrix.

```
igraph_int_t igraph_matrix_capacity(const igraph_matrix_t *m);
```

Note that this might be different from the size of the matrix (as queried by `igraph_matrix_size()`, and specifies how many elements the matrix can hold, without reallocation.

Arguments:

v: Pointer to the (previously initialized) matrix object to query.

Returns:

The allocated capacity.

See also:

`igraph_matrix_size()`, `igraph_matrix_nrow()`, `igraph_matrix_ncol()`.

Time complexity: O(1).

`igraph_matrix_nrow` — The number of rows in a matrix.

```
igraph_int_t igraph_matrix_nrow(const igraph_matrix_t *m);
```

Arguments:

m: Pointer to an initialized matrix object.

Returns:

The number of rows in the matrix.

Time complexity: O(1).

`igraph_matrix_ncol` — The number of columns in a matrix.

```
igraph_int_t igraph_matrix_ncol(const igraph_matrix_t *m);
```

Arguments:

m: Pointer to an initialized matrix object.

Returns:

The number of columns in the matrix.

Time complexity: O(1).

`igraph_matrix_is_symmetric` — Is the matrix symmetric?

```
igraph_bool_t igraph_matrix_is_symmetric(const igraph_matrix_t *m);
```

A non-square matrix is not symmetric by definition.

Arguments:

m: The input matrix.

Returns:

Boolean, `true` if the matrix is square and symmetric, `false` otherwise.

Time complexity: O(mn), the number of elements. O(1) for non-square matrices.

`igraph_matrix_maxdifference` — Maximum absolute difference between two matrices.

```
igraph_real_t igraph_matrix_maxdifference(const igraph_matrix_t *m1,
        const igraph_matrix_t *m2);
```

Calculate the maximum absolute difference of two matrices. Both matrices must be non-empty. If their dimensions differ then a warning is given and the comparison is performed by vectors columnwise from both matrices. The remaining elements in the larger vector are ignored.

Arguments:

m1: The first matrix.

m2: The second matrix.

Returns:

The element with the largest absolute value in `m1 - m2`.

Time complexity: O(mn), the elements in the smaller matrix.

Searching for elements

`igraph_matrix_contains` — Search for an element.

```
igraph_bool_t igraph_matrix_contains(const igraph_matrix_t *m,
        igraph_real_t e);
```

Search for the given element in the matrix.

Arguments:

m: The input matrix.

e: The element to search for.

Returns:

Boolean, `true` if the matrix contains *e*, `false` otherwise.

Time complexity: O(mn), the number of elements.

`igraph_matrix_search` — Search from a given position.

```
igraph_bool_t igraph_matrix_search(const igraph_matrix_t *m,
        igraph_int_t from, igraph_real_t what, igraph_int_t *pos,
        igraph_int_t *row, igraph_int_t *col);
```

Search for an element in a matrix and start the search from the given position. The search is performed columnwise.

Arguments:

m: The input matrix.

from: The position to search from, the positions are enumerated columnwise.

what: The element to search for.

pos: Pointer to an igraph_int_t. If the element is found, then this is set to the position of its first appearance.

row: Pointer to an igraph_int_t. If the element is found, then this is set to its row index.

col: Pointer to an igraph_int_t. If the element is found, then this is set to its column index.

Returns:

Boolean, `true` if the element is found, `false` otherwise.

Time complexity: O(mn), the number of elements.

Resizing operations

`igraph_matrix_resize` — Resizes a matrix.

```
igraph_error_t igraph_matrix_resize(igraph_matrix_t *m, igraph_int_t nrow, igra
```

This function resizes a matrix by adding more elements to it. The matrix contains arbitrary data after resizing it. That is, after calling this function you cannot expect that element (i,j) in the matrix remains the same as before.

Arguments:

m: Pointer to an already initialized matrix object.

nrow: The number of rows in the resized matrix.

ncol: The number of columns in the resized matrix.

Returns:

Error code.

Time complexity: O(1) if the matrix gets smaller, usually O(n) if it gets larger, n is the number of elements in the resized matrix.

`igraph_matrix_resize_min` — Deallocates unused memory for a matrix.

```
void igraph_matrix_resize_min(igraph_matrix_t *m);
```

This function attempts to deallocate the unused reserved storage of a matrix.

Arguments:

m: Pointer to an initialized matrix.

See also:

`igraph_matrix_resize()`.

Time complexity: operating system dependent, O(n) at worst.

`igraph_matrix_add_rows` — Adds rows to a matrix.

```
igraph_error_t igraph_matrix_add_rows(igraph_matrix_t *m, igraph_int_t n);
```

Arguments:

m: The matrix object.

n: The number of rows to add.

Returns:

Error code, `IGRAPH_ENOMEM` if there isn't enough memory for the operation.

Time complexity: linear with the number of elements of the new, resized matrix.

`igraph_matrix_add_cols` — Adds columns to a matrix.

```
igraph_error_t igraph_matrix_add_cols(igraph_matrix_t *m, igraph_int_t n);
```

Arguments:

m: The matrix object.

n: The number of columns to add.

Returns:

Error code, `IGRAPH_ENOMEM` if there is not enough memory to perform the operation.

Time complexity: linear with the number of elements of the new, resized matrix.

`igraph_matrix_remove_row` — Remove a row.

```
igraph_error_t igraph_matrix_remove_row(igraph_matrix_t *m, igraph_int_t row);
```

A row is removed from the matrix.

Arguments:

m: The input matrix.

row: The index of the row to remove.

Returns:

Error code.

Time complexity: O(mn), the number of elements in the matrix.

`igraph_matrix_remove_col` — Removes a column from a matrix.

```
igraph_error_t igraph_matrix_remove_col(igraph_matrix_t *m, igraph_int_t col);
```

Arguments:

m: The matrix object.

col: The column to remove.

Returns:

Error code, always returns with success.

Time complexity: linear with the number of elements of the new, resized matrix.

Complex matrix operations

igraph_matrix_complex_real — Gives the real part of a complex matrix.

```
igraph_error_t igraph_matrix_complex_real(const igraph_matrix_complex_t *m,
                                          igraph_matrix_t *real);
```

Arguments:

m: Pointer to a complex matrix.

real: Pointer to an initialized matrix. The result will be stored here.

Returns:

Error code.

Time complexity: O(n), n is the number of elements in the matrix.

igraph_matrix_complex_imag — Gives the imaginary part of a complex matrix.

```
igraph_error_t igraph_matrix_complex_imag(const igraph_matrix_complex_t *m,
                                          igraph_matrix_t *imag);
```

Arguments:

m: Pointer to a complex matrix.

imag: Pointer to an initialized matrix. The result will be stored here.

Returns:

Error code.

Time complexity: O(n), n is the number of elements in the matrix.

igraph_matrix_complex_realimag — Gives the real and imaginary parts of a complex matrix.

```
igraph_error_t igraph_matrix_complex_realimag(const igraph_matrix_complex_t *m,
                                      igraph_matrix_t *real,
                                      igraph_matrix_t *imag);
```

Arguments:

m: Pointer to a complex matrix.

real: Pointer to an initialized matrix. The real part will be stored here.

imag: Pointer to an initialized matrix. The imaginary part will be stored here.

Returns:

Error code.

Time complexity: O(n), n is the number of elements in the matrix.

`igraph_matrix_complex_create` — Creates a complex matrix from a real and imaginary part.

```
igraph_error_t igraph_matrix_complex_create(igraph_matrix_complex_t *m,
                                    const igraph_matrix_t *real,
                                    const igraph_matrix_t *imag);
```

Arguments:

m: Pointer to an uninitialized complex matrix.

real: Pointer to the real part of the complex matrix.

imag: Pointer to the imaginary part of the complex matrix.

Returns:

Error code.

Time complexity: O(n), n is the number of elements in the matrix.

`igraph_matrix_complex_create_polar` — Creates a complex matrix from a magnitude and an angle.

```
igraph_error_t igraph_matrix_complex_create_polar(igraph_matrix_complex_t *m,
                                        const igraph_matrix_t *r,
                                        const igraph_matrix_t *theta);
```

Arguments:

m: Pointer to an uninitialized complex matrix.

r: Pointer to a real matrix containing magnitudes.

theta: Pointer to a real matrix containing arguments (phase angles).

Returns:

Error code.

Time complexity: O(n), n is the number of elements in the matrix.

igraph_matrix_complex_all_almost_e — Are all elements almost equal?

```
igraph_bool_t igraph_matrix_complex_all_almost_e(igraph_matrix_complex_t *lhs,
                                                 igraph_matrix_complex_t *rhs,
                                                 igraph_real_t eps);
```

Checks if the elements of two complex matrices are equal within a relative tolerance.

Arguments:

lhs: The first matrix.

rhs: The second matrix.

eps: Relative tolerance, see `igraph_complex_almost_equals()` for details.

Returns:

True if the two matrices are almost equal, false if there is at least one differing element or if the matrices are not of the same dimensions.

igraph_matrix_complex_zapsmall — Replaces small elements of a complex matrix by exact zeros.

```
igraph_error_t igraph_matrix_complex_zapsmall(igraph_matrix_complex_t *m, igraph
```

Similarly to `igraph_matrix_zapsmall()`, small elements will be replaced by zeros. The operation is performed separately on the real and imaginary parts of the numbers. This way, complex numbers with a large real part and tiny imaginary part will effectively be transformed to real numbers. The default tolerance corresponds to two-thirds of the representable digits of igraph_real_t, i.e. `DBL_EPSILON^(2/3)` which is approximately `10^-10`.

Arguments:

m: The matrix to process, it will be changed in-place.

tol: Tolerance value. Real and imaginary parts smaller than this in magnitude will be replaced by zeros. Pass in zero to use the default tolerance. Must not be negative.

Returns:

Error code.

See also:

`igraph_matrix_complex_all_almost_e()` and `igraph_complex_almost_equals()` to perform comparisons with relative tolerances.

Sparse matrices

About sparse matrices

The `igraph_sparsemat_t` data type stores sparse matrices, i.e. matrices in which the majority of the elements are zero.

The data type is essentially a wrapper to some of the functions in the CXSparse library, by Tim Davis, see http://faculty.cse.tamu.edu/davis/suitesparse.html

Matrices can be stored in two formats: triplet and column-compressed. The triplet format is intended for sparse matrix initialization, as it is easy to add new (non-zero) elements to it. Most of the computations are done on sparse matrices in column-compressed format, after the user has converted the triplet matrix to column-compressed, via `igraph_sparsemat_compress()`.

Both formats are dynamic, in the sense that new elements can be added to them, possibly resulting the allocation of more memory.

Row and column indices follow the C convention and are zero-based.

Example 7.4. File `examples/simple/igraph_sparsemat.c`

Example 7.5. File `examples/simple/igraph_sparsemat3.c`

Example 7.6. File `examples/simple/igraph_sparsemat4.c`

Example 7.7. File `examples/simple/igraph_sparsemat6.c`

Example 7.8. File `examples/simple/igraph_sparsemat7.c`

Example 7.9. File `examples/simple/igraph_sparsemat8.c`

Creating sparse matrix objects

`igraph_sparsemat_init` — Initializes a sparse matrix, in triplet format.

```
igraph_error_t igraph_sparsemat_init(igraph_sparsemat_t *A, igraph_int_t rows,
        igraph_int_t cols, igraph_int_t nzmax);
```

This is the most common way to create a sparse matrix, together with the `igraph_sparsemat_entry()` function, which can be used to add the non-zero elements one by one. Once done, the user can call `igraph_sparsemat_compress()` to convert the matrix to column-compressed, to allow computations with it.

The user must call `igraph_sparsemat_destroy()` on the matrix to deallocate the memory, once the matrix is no more needed.

Arguments:

A: Pointer to a not yet initialized sparse matrix.

rows: The number of rows in the matrix.

cols: The number of columns.

nzmax: The maximum number of non-zero elements in the matrix. It is not compulsory to get this right, but it is useful for the allocation of the proper amount of memory.

Returns:

Error code.

Time complexity: TODO.

igraph_sparsemat_init_copy — Copies a sparse matrix.

```
igraph_error_t igraph_sparsemat_init_copy(
    igraph_sparsemat_t *to, const igraph_sparsemat_t *from
);
```

Create a sparse matrix object, by copying another one. The source matrix can be either in triplet or column-compressed format.

Exactly the same amount of memory will be allocated to the copy matrix, as it is currently for the original one.

Arguments:

to: Pointer to an uninitialized sparse matrix, the copy will be created here.

from: The sparse matrix to copy.

Returns:

Error code.

Time complexity: O(n+nzmax), the number of columns plus the maximum number of non-zero elements.

igraph_sparsemat_init_diag — Creates a sparse diagonal matrix.

```
igraph_error_t igraph_sparsemat_init_diag(
    igraph_sparsemat_t *A, igraph_int_t nzmax, const igraph_vector_t *values,
    igraph_bool_t compress
);
```

Arguments:

A: An uninitialized sparse matrix, the result is stored here.

nzmax: The maximum number of non-zero elements, this essentially gives the amount of memory that will be allocated for matrix elements.

values: The values to store in the diagonal, the size of the matrix defined by the length of this vector.

compress: Whether to create a column-compressed matrix. If false, then a triplet matrix is created.

Returns:

Error code.

Time complexity: O(n), the length of the diagonal vector.

igraph_sparsemat_init_eye — Creates a sparse identity matrix.

```
igraph_error_t igraph_sparsemat_init_eye(
    igraph_sparsemat_t *A, igraph_int_t n, igraph_int_t nzmax,
    igraph_real_t value, igraph_bool_t compress
);
```

Arguments:

A: An uninitialized sparse matrix, the result is stored here.

n: The number of rows and number of columns in the matrix.

nzmax: The maximum number of non-zero elements, this essentially gives the amount of memory that will be allocated for matrix elements.

value: The value to store in the diagonal.

compress: Whether to create a column-compressed matrix. If false, then a triplet matrix is created.

Returns:

Error code.

Time complexity: O(n).

igraph_sparsemat_realloc — Allocates more (or less) memory for a sparse matrix.

```
igraph_error_t igraph_sparsemat_realloc(igraph_sparsemat_t *A, igraph_int_t nzma
```

Sparse matrices automatically allocate more memory, as needed. To control memory allocation, the user can call this function, to allocate memory for a given number of non-zero elements.

Arguments:

A: The sparse matrix, it can be in triplet or column-compressed format.

nzmax: The new maximum number of non-zero elements.

Returns:

Error code.

Time complexity: TODO.

`igraph_sparsemat_destroy` — Deallocates memory used by a sparse matrix.

```
void igraph_sparsemat_destroy(igraph_sparsemat_t *A);
```

One destroyed, the sparse matrix must be initialized again, before calling any other operation on it.

Arguments:

A: The sparse matrix to destroy.

Time complexity: O(1).

Query properties of a sparse matrix

`igraph_sparsemat_index` — Extracts a submatrix or a single element.

```
igraph_error_t igraph_sparsemat_index(const igraph_sparsemat_t *A,
                           const igraph_vector_int_t *p,
                           const igraph_vector_int_t *q,
                           igraph_sparsemat_t *res,
                           igraph_real_t *constres);
```

This function indexes into a sparse matrix. It serves two purposes. First, it can extract submatrices from a sparse matrix. Second, as a special case, it can extract a single element from a sparse matrix.

Arguments:

A: The input matrix, it must be in column-compressed format.

p: An integer vector, or a null pointer. The selected row index or indices. A null pointer selects all rows.

q: An integer vector, or a null pointer. The selected column index or indices. A null pointer selects all columns.

res: Pointer to an uninitialized sparse matrix, or a null pointer. If not a null pointer, then the selected submatrix is stored here.

constres: Pointer to a real variable or a null pointer. If not a null pointer, then the first non-zero element in the selected submatrix is stored here, if there is one. Otherwise zero is stored here. This behavior is handy if one wants to select a single entry from the matrix.

Returns:

Error code.

Time complexity: TODO.

`igraph_sparsemat_nrow` — Number of rows.

```
igraph_int_t igraph_sparsemat_nrow(const igraph_sparsemat_t *A);
```

Arguments:

A: The input matrix, in triplet or column-compressed format.

Returns:

The number of rows in the *A* matrix.

Time complexity: O(1).

`igraph_sparsemat_ncol` — Number of columns.

```
igraph_int_t igraph_sparsemat_ncol(const igraph_sparsemat_t *A);
```

Arguments:

A: The input matrix, in triplet or column-compressed format.

Returns:

The number of columns in the *A* matrix.

Time complexity: O(1).

`igraph_sparsemat_type` — Type of a sparse matrix (triplet or column-compressed).

```
igraph_sparsemat_type_t igraph_sparsemat_type(const igraph_sparsemat_t *A);
```

Gives whether a sparse matrix is stored in the triplet format or in column-compressed format.

Arguments:

A: The input matrix.

Returns:

Either `IGRAPH_SPARSEMAT_CC` or `IGRAPH_SPARSEMAT_TRIPLET`.

Time complexity: O(1).

`igraph_sparsemat_is_triplet` — Is this sparse matrix in triplet format?

```
igraph_bool_t igraph_sparsemat_is_triplet(const igraph_sparsemat_t *A);
```

Decides whether a sparse matrix is in triplet format.

Arguments:

A: The input matrix.

Returns:

One if the input matrix is in triplet format, zero otherwise.

Time complexity: O(1).

`igraph_sparsemat_is_cc` — Is this sparse matrix in column-compressed format?

```
igraph_bool_t igraph_sparsemat_is_cc(const igraph_sparsemat_t *A);
```

Decides whether a sparse matrix is in column-compressed format.

Arguments:

A: The input matrix.

Returns:

One if the input matrix is in column-compressed format, zero otherwise.

Time complexity: O(1).

`igraph_sparsemat_is_symmetric` — Returns whether a sparse matrix is symmetric.

```
igraph_error_t igraph_sparsemat_is_symmetric(const igraph_sparsemat_t *A, igraph
```

Arguments:

A: The input matrix

result: Pointer to an `igraph_bool_t` ; the result is provided here.

Returns:

Error code.

`igraph_sparsemat_get` — Return the value of a single element from a sparse matrix.

```
igraph_real_t igraph_sparsemat_get(
    const igraph_sparsemat_t *A, igraph_int_t row, igraph_int_t col
);
```

Arguments:

A: The input matrix, in triplet or column-compressed format.

row: The row index

col: The column index

Returns:

The value of the cell with the given row and column indices in the matrix; zero if the indices are out of bounds.

Time complexity: TODO.

`igraph_sparsemat_getelements` — Returns all elements of a sparse matrix.

```
igraph_error_t igraph_sparsemat_getelements(const igraph_sparsemat_t *A,
                                    igraph_vector_int_t *i,
                                    igraph_vector_int_t *j,
                                    igraph_vector_t *x);
```

This function will return the elements of a sparse matrix in three vectors. Two vectors will indicate where the elements are located, and one will specify the elements themselves.

Arguments:

A: A sparse matrix in either triplet or compressed form.

i: An initialized integer vector. This will store the rows of the returned elements.

j: An initialized integer vector. For a triplet matrix this will store the columns of the returned elements. For a compressed matrix, if the column index is `k`, then `j[k]` is the index in *x* of the start of the `k-th` column, and the last element of `j` is the total number of elements. The total number of elements in the `k-th` column is therefore `j[k+1] - j[k]`. For example, if there is one element in the first column, and five in the second, `j` will be set to `{0, 1, 6}`.

x: An initialized vector. The elements will be placed here.

Returns:

Error code.

Time complexity: O(n), the number of stored elements in the sparse matrix.

`igraph_sparsemat_getelements_sorted` — Returns all elements of a sparse matrix, sorted by row and column indices.

```
igraph_error_t igraph_sparsemat_getelements_sorted(const igraph_sparsemat_t *A,
                                        igraph_vector_int_t *i,
                                        igraph_vector_int_t *j,
                                        igraph_vector_t *x);
```

This function will sort a sparse matrix and return the elements in three vectors. Two vectors will indicate where the elements are located, and one will specify the elements themselves.

Sorting is done based on the *indices* of the elements, not their numeric values. The returned entries will be sorted by column indices; entries in the same column are then sorted by row indices.

Arguments:

A: A sparse matrix in either triplet or compressed form.

i: An initialized integer vector. This will store the rows of the returned elements.

j: An initialized integer vector. For a triplet matrix this will store the columns of the returned elements. For a compressed matrix, if the column index is `k`, then `j[k]` is the index in *x* of the start of the `k-th` column, and the last element of `j` is the total number of elements. The total number of elements in the `k-th` column is therefore `j[k+1] - j[k]`. For example, if there is one element in the first column, and five in the second, `j` will be set to `{0, 1, 6}`.

x: An initialized vector. The elements will be placed here.

Returns:

Error code.

Time complexity: TODO.

`igraph_sparsemat_min` — Minimum of a sparse matrix.

```
igraph_real_t igraph_sparsemat_min(igraph_sparsemat_t *A);
```

Arguments:

A: The input matrix, column-compressed.

Returns:

The minimum in the input matrix, or `IGRAPH_INFINITY` if the matrix has zero elements.

Time complexity: TODO.

`igraph_sparsemat_max` — Maximum of a sparse matrix.

```
igraph_real_t igraph_sparsemat_max(igraph_sparsemat_t *A);
```

Arguments:

A: The input matrix, column-compressed.

Returns:

The maximum in the input matrix, or `-IGRAPH_INFINITY` if the matrix has zero elements.

Time complexity: TODO.

`igraph_sparsemat_minmax` — Minimum and maximum of a sparse matrix.

```
igraph_error_t igraph_sparsemat_minmax(igraph_sparsemat_t *A,
                            igraph_real_t *min, igraph_real_t *max);
```

Arguments:

A: The input matrix, column-compressed.

min: The minimum in the input matrix is stored here, or `IGRAPH_INFINITY` if the matrix has zero elements.

max: The maximum in the input matrix is stored here, or `-IGRAPH_INFINITY` if the matrix has zero elements.

Returns:

Error code.

Time complexity: TODO.

`igraph_sparsemat_count_nonzero` — Counts nonzero elements of a sparse matrix.

```
igraph_int_t igraph_sparsemat_count_nonzero(igraph_sparsemat_t *A);
```

Arguments:

A: The input matrix, column-compressed.

Returns:

Error code.

Time complexity: TODO.

`igraph_sparsemat_count_nonzerotol` — Counts nonzero elements of a sparse matrix, ignoring elements close to zero.

```
igraph_int_t igraph_sparsemat_count_nonzerotol(igraph_sparsemat_t *A,
        igraph_real_t tol);
```

Count the number of matrix entries that are closer to zero than *tol*.

Arguments:

A: The input matrix, column-compressed.

tol: The tolerance for zero comparisons.

Returns:

Error code.

Time complexity: TODO.

`igraph_sparsemat_rowsums` — Row-wise sums.

```
igraph_error_t igraph_sparsemat_rowsums(const igraph_sparsemat_t *A,
                                        igraph_vector_t *res);
```

Arguments:

A: The input matrix, in triplet or column-compressed format.

res: An initialized vector, the result is stored here. It will be resized as needed.

Returns:

Error code.

Time complexity: O(nz), the number of non-zero elements.

igraph_sparsemat_colsums — Column-wise sums.

```
igraph_error_t igraph_sparsemat_colsums(const igraph_sparsemat_t *A,
                                        igraph_vector_t *res);
```

Arguments:

A: The input matrix, in triplet or column-compressed format.

res: An initialized vector, the result is stored here. It will be resized as needed.

Returns:

Error code.

Time complexity: O(nz) for triplet matrices, O(nz+n) for column-compressed ones, nz is the number of non-zero elements, n is the number of columns.

igraph_sparsemat_nonzero_storage — Returns number of stored entries of a sparse matrix.

```
igraph_int_t igraph_sparsemat_nonzero_storage(const igraph_sparsemat_t *A);
```

This function will return the number of stored entries of a sparse matrix. These entries can be zero, and multiple entries can be at the same position. Use `igraph_sparsemat_dupl()` to sum duplicate entries, and `igraph_sparsemat_dropzeros()` to remove zeros.

Arguments:

A: A sparse matrix in either triplet or compressed form.

Returns:

Number of stored entries.

Time complexity: O(1).

Operations on sparse matrices

igraph_sparsemat_entry — Adds an element to a sparse matrix.

```
igraph_error_t igraph_sparsemat_entry(igraph_sparsemat_t *A,
```

```
        igraph_int_t row, igraph_int_t col, igraph_real_t elem);
```

This function can be used to add the entries to a sparse matrix, after initializing it with `igraph_sparsemat_init()`. If you add multiple entries in the same position, they will all be saved, and the resulting value is the sum of all entries in that position.

Arguments:

A: The input matrix, it must be in triplet format.

row: The row index of the entry to add.

col: The column index of the entry to add.

elem: The value of the entry.

Returns:

Error code.

Time complexity: O(1) on average.

`igraph_sparsemat_fkeep` — Filters the elements of a sparse matrix.

```
igraph_error_t igraph_sparsemat_fkeep(
    igraph_sparsemat_t *A,
    igraph_int_t (*fkeep)(igraph_int_t, igraph_int_t, igraph_real_t, void*),
    void *other
);
```

This function can be used to filter the (non-zero) elements of a sparse matrix. For all entries, it calls the supplied function and depending on the return values either keeps, or deleted the element from the matrix.

Arguments:

A: The input matrix, in column-compressed format.

fkeep: The filter function. It must take four arguments: the first is an `igraph_int_t`, the row index of the entry, the second is another `igraph_int_t`, the column index. The third is `igraph_real_t`, the value of the entry. The fourth element is a `void` pointer, the *`other`* argument is passed here. The function must return an `int`. If this is zero, then the entry is deleted, otherwise it is kept.

other: A `void` pointer that is passed to the filtering function.

Returns:

Error code.

Time complexity: TODO.

`igraph_sparsemat_dropzeros` — Drops the zero elements from a sparse matrix.

```
igraph_error_t igraph_sparsemat_dropzeros(igraph_sparsemat_t *A);
```

As a result of matrix operations, some of the entries in a sparse matrix might be zero. This function removes these entries.

Arguments:

A: The input matrix, it must be in column-compressed format.

Returns:

Error code.

Time complexity: TODO.

`igraph_sparsemat_droptol` — Drops the almost zero elements from a sparse matrix.

```
igraph_error_t igraph_sparsemat_droptol(igraph_sparsemat_t *A, igraph_real_t to
```

This function is similar to `igraph_sparsemat_dropzeros()`, but it also drops entries that are closer to zero than the given tolerance threshold.

Arguments:

A: The input matrix, it must be in column-compressed format.

tol: Real number, giving the tolerance threshold.

Returns:

Error code.

Time complexity: TODO.

`igraph_sparsemat_scale` — Scales a sparse matrix.

```
igraph_error_t igraph_sparsemat_scale(igraph_sparsemat_t *A, igraph_real_t by);
```

Multiplies all elements of a sparse matrix, by the given factor.

Arguments:

A: The input matrix.

by: The scaling factor.

Returns:

Error code.

Time complexity: O(nz), the number of non-zero elements in the matrix.

`igraph_sparsemat_permute` — Permutes the rows and columns of a sparse matrix.

```
igraph_error_t igraph_sparsemat_permute(const igraph_sparsemat_t *A,
                                        const igraph_vector_int_t *p,
                                        const igraph_vector_int_t *q,
                                        igraph_sparsemat_t *res);
```

Arguments:

A: The input matrix, it must be in column-compressed format.

p: Integer vector, giving the permutation of the rows.

q: Integer vector, the permutation of the columns.

res: Pointer to an uninitialized sparse matrix, the result is stored here.

Returns:

Error code.

Time complexity: O(m+n+nz), the number of rows plus the number of columns plus the number of non-zero elements in the matrix.

`igraph_sparsemat_transpose` — Transposes a sparse matrix.

```
igraph_error_t igraph_sparsemat_transpose(
    const igraph_sparsemat_t *A, igraph_sparsemat_t *res
);
```

Arguments:

A: The input matrix, column-compressed or triple format.

res: Pointer to an uninitialized sparse matrix, the result is stored here.

Returns:

Error code.

Time complexity: TODO.

`igraph_sparsemat_add` — Sum of two sparse matrices.

```
igraph_error_t igraph_sparsemat_add(const igraph_sparsemat_t *A,
                          const igraph_sparsemat_t *B,
                          igraph_real_t alpha,
                          igraph_real_t beta,
                          igraph_sparsemat_t *res);
```

Arguments:

A: The first input matrix, in column-compressed format.

B: The second input matrix, in column-compressed format.

alpha: Real value, *A* is multiplied by *alpha* before the addition.

beta: Real value, *B* is multiplied by *beta* before the addition.

res: Pointer to an uninitialized sparse matrix, the result is stored here.

Returns:

Error code.

Time complexity: TODO.

igraph_sparsemat_multiply — Matrix multiplication.

```
igraph_error_t igraph_sparsemat_multiply(const igraph_sparsemat_t *A,
                                         const igraph_sparsemat_t *B,
                                         igraph_sparsemat_t *res);
```

Multiplies two sparse matrices.

Arguments:

A: The first input matrix (left hand side), in column-compressed format.

B: The second input matrix (right hand side), in column-compressed format.

res: Pointer to an uninitialized sparse matrix, the result is stored here.

Returns:

Error code.

Time complexity: TODO.

igraph_sparsemat_gaxpy — Matrix-vector product, added to another vector.

```
igraph_error_t igraph_sparsemat_gaxpy(const igraph_sparsemat_t *A,
                                      const igraph_vector_t *x,
                                      igraph_vector_t *res);
```

Arguments:

A: The input matrix, in column-compressed format.

x: The input vector, its size must match the number of columns in *A*.

res: This vector is added to the matrix-vector product and it is overwritten by the result.

Returns:

Error code.

Time complexity: TODO.

igraph_sparsemat_add_rows — Adds rows to a sparse matrix.

```
igraph_error_t igraph_sparsemat_add_rows(igraph_sparsemat_t *A, igraph_int_t n)
```

The current matrix elements are retained and all elements in the new rows are zero.

Arguments:

A: The input matrix, in triplet or column-compressed format.

n: The number of rows to add.

Returns:

Error code.

Time complexity: O(1).

`igraph_sparsemat_add_cols` — Adds columns to a sparse matrix.

```
igraph_error_t igraph_sparsemat_add_cols(igraph_sparsemat_t *A, igraph_int_t n)
```

The current matrix elements are retained, and all elements in the new columns are zero.

Arguments:

A: The input matrix, in triplet or column-compressed format.

n: The number of columns to add.

Returns:

Error code.

Time complexity: TODO.

`igraph_sparsemat_resize` — Resizes a sparse matrix and clears all the elements.

```
igraph_error_t igraph_sparsemat_resize(igraph_sparsemat_t *A, igraph_int_t nrow,
                                       igraph_int_t ncol, igraph_int_t nzmax);
```

This function resizes a sparse matrix. The resized sparse matrix will become empty, even if it contained nonzero entries.

Arguments:

A: The initialized sparse matrix to resize.

nrow: The new number of rows.

ncol: The new number of columns.

nzmax: The new maximum number of elements.

Returns:

Error code.

Time complexity: O(nzmax), the maximum number of non-zero elements.

`igraph_sparsemat_sort` — Sorts all elements of a sparse matrix by row and column indices.

```
igraph_error_t igraph_sparsemat_sort(const igraph_sparsemat_t *A,
                                     igraph_sparsemat_t *sorted);
```

This function will sort the elements of a sparse matrix such that iterating over the entries will return them sorted by column indices; elements in the same column are then sorted by row indices.

Arguments:

A: A sparse matrix in either triplet or compressed form.

sorted: An uninitialized sparse matrix; the result will be returned here. The result will be in triplet form if the input was in triplet form, otherwise it will be in compressed form. Note that sorting is more efficient when the matrix is already in compressed form.

Returns:

Error code.

Time complexity: TODO

Operations on sparse matrix iterators

`igraph_sparsemat_iterator_init` — Initialize a sparse matrix iterator.

```
igraph_error_t igraph_sparsemat_iterator_init(
    igraph_sparsemat_iterator_t *it, const igraph_sparsemat_t *sparsemat
);
```

Arguments:

it: A pointer to an uninitialized sparse matrix iterator.

sparsemat: Pointer to the sparse matrix.

Returns:

Error code. This will always return `IGRAPH_SUCCESS`

Time complexity: O(n), the number of columns of the sparse matrix.

`igraph_sparsemat_iterator_reset` — Reset a sparse matrix iterator to the first element.

```
igraph_error_t igraph_sparsemat_iterator_reset(igraph_sparsemat_iterator_t *it)
```

Arguments:

it: A pointer to the sparse matrix iterator.

Returns:

Error code. This will always return `IGRAPH_SUCCESS`

Time complexity: O(n), the number of columns of the sparse matrix.

`igraph_sparsemat_iterator_end` — Query if the iterator is past the last element.

```
igraph_bool_t
igraph_sparsemat_iterator_end(const igraph_sparsemat_iterator_t *it);
```

Arguments:

it: A pointer to the sparse matrix iterator.

Returns:

true if the iterator is past the last element, false if it points to an element in a sparse matrix.

Time complexity: O(1).

`igraph_sparsemat_iterator_row` — Return the row of the iterator.

```
igraph_int_t igraph_sparsemat_iterator_row(const igraph_sparsemat_iterator_t *i
```

Arguments:

it: A pointer to the sparse matrix iterator.

Returns:

The row of the element at the current iterator position.

Time complexity: O(1).

`igraph_sparsemat_iterator_col` — Return the column of the iterator.

```
igraph_int_t igraph_sparsemat_iterator_col(const igraph_sparsemat_iterator_t *i
```

Arguments:

it: A pointer to the sparse matrix iterator.

Returns:

The column of the element at the current iterator position.

Time complexity: O(1).

`igraph_sparsemat_iterator_get` — Return the element at the current iterator position.

```
igraph_real_t
igraph_sparsemat_iterator_get(const igraph_sparsemat_iterator_t *it);
```

Arguments:

`it`: A pointer to the sparse matrix iterator.

Returns:

The value of the element at the current iterator position.

Time complexity: O(1).

`igraph_sparsemat_iterator_next` — Let a sparse matrix iterator go to the next element.

```
igraph_int_t igraph_sparsemat_iterator_next(igraph_sparsemat_iterator_t *it);
```

Arguments:

`it`: A pointer to the sparse matrix iterator.

Returns:

The position of the iterator in the element vector.

Time complexity: O(n), the number of columns of the sparse matrix.

`igraph_sparsemat_iterator_idx` — Returns the element vector index of a sparse matrix iterator.

```
igraph_int_t igraph_sparsemat_iterator_idx(const igraph_sparsemat_iterator_t *i
```

Arguments:

`it`: A pointer to the sparse matrix iterator.

Returns:

The position of the iterator in the element vector.

Time complexity: O(1).

Operations that change the internal representation

`igraph_sparsemat_compress` — Converts a sparse matrix to column-compressed format.

```
igraph_error_t igraph_sparsemat_compress(const igraph_sparsemat_t *A,
                                         igraph_sparsemat_t *res);
```

Converts a sparse matrix from triplet format to column-compressed format. Almost all sparse matrix operations require that the matrix is in column-compressed format.

Arguments:

A: The input matrix, it must be in triplet format.

res: Pointer to an uninitialized sparse matrix object, the compressed version of *A* is stored here.

Returns:

Error code.

Time complexity: O(nz) where `nz` is the number of non-zero elements.

`igraph_sparsemat_dupl` — Removes duplicate elements from a sparse matrix.

```
igraph_error_t igraph_sparsemat_dupl(igraph_sparsemat_t *A);
```

It is possible that a column-compressed sparse matrix stores a single matrix entry in multiple pieces. The entry is then the sum of all its pieces. (Some functions create matrices like this.) This function eliminates the multiple pieces.

Arguments:

A: The input matrix, in column-compressed format.

Returns:

Error code.

Time complexity: TODO.

Decompositions and solving linear systems

`igraph_sparsemat_symblu` — Symbolic LU decomposition.

```
igraph_error_t igraph_sparsemat_symblu(igraph_int_t order, const igraph_sparsema
                                       igraph_sparsemat_symbolic_t *dis);
```

LU decomposition of sparse matrices involves two steps, the first is calling this function, and then `igraph_sparsemat_lu()`.

Arguments:

order: The ordering to use: 0 means natural ordering, 1 means minimum degree ordering of A +A', 2 is minimum degree ordering of A'A after removing the dense rows from A, and 3 is the minimum degree ordering of A'A.

A: The input matrix, in column-compressed format.

dis: The result of the symbolic analysis is stored here. Once not needed anymore, it must be destroyed by calling `igraph_sparsemat_symbolic_destroy()`.

Returns:

Error code.

Time complexity: TODO.

`igraph_sparsemat_symbqr` — Symbolic QR decomposition.

```
igraph_error_t igraph_sparsemat_symbqr(igraph_int_t order, const igraph_sparsema
                            igraph_sparsemat_symbolic_t *dis);
```

QR decomposition of sparse matrices involves two steps, the first is calling this function, and then `igraph_sparsemat_qr()`.

Arguments:

order: The ordering to use: 0 means natural ordering, 1 means minimum degree ordering of A +A', 2 is minimum degree ordering of A'A after removing the dense rows from A, and 3 is the minimum degree ordering of A'A.

A: The input matrix, in column-compressed format.

dis: The result of the symbolic analysis is stored here. Once not needed anymore, it must be destroyed by calling `igraph_sparsemat_symbolic_destroy()`.

Returns:

Error code.

Time complexity: TODO.

`igraph_sparsemat_lsolve` — Solves a lower-triangular linear system.

```
igraph_error_t igraph_sparsemat_lsolve(const igraph_sparsemat_t *L,
                            const igraph_vector_t *b,
                            igraph_vector_t *res);
```

Solve the Lx=b linear equation system, where the L coefficient matrix is square and lower-triangular, with a zero-free diagonal.

Arguments:

L: The input matrix, in column-compressed format.

b: The right hand side of the linear system.

res: An initialized vector, the result is stored here.

Returns:

Error code.

Time complexity: TODO.

igraph_sparsemat_ltsolve — Solves an upper-triangular linear system.

```
igraph_error_t igraph_sparsemat_ltsolve(const igraph_sparsemat_t *L,
                                        const igraph_vector_t *b,
                                        igraph_vector_t *res);
```

Solve the L'x=b linear equation system, where the L matrix is square and lower-triangular, with a zero-free diagonal.

Arguments:

L: The input matrix, in column-compressed format.

b: The right hand side of the linear system.

res: An initialized vector, the result is stored here.

Returns:

Error code.

Time complexity: TODO.

igraph_sparsemat_usolve — Solves an upper-triangular linear system.

```
igraph_error_t igraph_sparsemat_usolve(const igraph_sparsemat_t *U,
                                       const igraph_vector_t *b,
                                       igraph_vector_t *res);
```

Solves the Ux=b upper triangular system.

Arguments:

U: The input matrix, in column-compressed format.

b: The right hand side of the linear system.

res: An initialized vector, the result is stored here.

Returns:

Error code.

Time complexity: TODO.

igraph_sparsemat_utsolve — Solves a lower-triangular linear system.

```
igraph_error_t igraph_sparsemat_utsolve(const igraph_sparsemat_t *U,
                                        const igraph_vector_t *b,
                                        igraph_vector_t *res);
```

This is the same as `igraph_sparsemat_usolve()`, but U'x=b is solved, where the apostrophe denotes the transpose.

Arguments:

U: The input matrix, in column-compressed format.

b: The right hand side of the linear system.

res: An initialized vector, the result is stored here.

Returns:

Error code.

Time complexity: TODO.

`igraph_sparsemat_cholsol` — Solves a symmetric linear system via Cholesky decomposition.

```
igraph_error_t igraph_sparsemat_cholsol(const igraph_sparsemat_t *A,
                                        const igraph_vector_t *b,
                                        igraph_vector_t *res,
                                        igraph_int_t order);
```

Solve Ax=b, where A is a symmetric positive definite matrix.

Arguments:

A: The input matrix, in column-compressed format.

b: The right hand side.

res: An initialized vector, the result is stored here.

order: An integer giving the ordering method to use for the factorization. Zero is the natural ordering; if it is one, then the fill-reducing minimum-degree ordering of A+A' is used.

Returns:

Error code.

Time complexity: TODO.

`igraph_sparsemat_lusol` — Solves a linear system via LU decomposition.

```
igraph_error_t igraph_sparsemat_lusol(const igraph_sparsemat_t *A,
                                      const igraph_vector_t *b,
                                      igraph_vector_t *res,
                                      igraph_int_t order,
```

```
                                igraph_real_t tol);
```

Solve Ax=b, via LU factorization of A.

Arguments:

A: The input matrix, in column-compressed format.

b: The right hand side of the equation.

res: An initialized vector, the result is stored here.

order: The ordering method to use, zero means the natural ordering, one means the fill-reducing minimum-degree ordering of A+A', two means the ordering of A'*A, after removing the dense rows from A. Three means the ordering of A'*A.

tol: Real number, the tolerance limit to use for the numeric LU factorization.

Returns:

Error code.

Time complexity: TODO.

`igraph_sparsemat_lu` — LU decomposition of a sparse matrix.

```
igraph_error_t igraph_sparsemat_lu(const igraph_sparsemat_t *A,
                        const igraph_sparsemat_symbolic_t *dis,
                        igraph_sparsemat_numeric_t *din, double tol);
```

Performs numeric sparse LU decomposition of a matrix.

Arguments:

A: The input matrix, in column-compressed format.

dis: The symbolic analysis for LU decomposition, coming from a call to the `igraph_sparsemat_symblu()` function.

din: The numeric decomposition, the result is stored here. It can be used to solve linear systems with changing right hand side vectors, by calling `igraph_sparsemat_luresol()`. Once not needed any more, it must be destroyed by calling `igraph_sparsemat_symbolic_destroy()` on it.

tol: The tolerance for the numeric LU decomposition.

Returns:

Error code.

Time complexity: TODO.

`igraph_sparsemat_qr` — QR decomposition of a sparse matrix.

```
igraph_error_t igraph_sparsemat_qr(const igraph_sparsemat_t *A,
                        const igraph_sparsemat_symbolic_t *dis,
                        igraph_sparsemat_numeric_t *din);
```

Numeric QR decomposition of a sparse matrix.

Arguments:

A: The input matrix, in column-compressed format.

dis: The result of the symbolic QR analysis, from the function `igraph_sparsemat_symbqr()`.

din: The result of the decomposition is stored here, it can be used to solve many linear systems with the same coefficient matrix and changing right hand sides, using the `igraph_sparsemat_qrresol()` function. Once not needed any more, one should call `igraph_sparsemat_numeric_destroy()` on it to free the allocated memory.

Returns:

Error code.

Time complexity: TODO.

`igraph_sparsemat_luresol` — Solves a linear system using a precomputed LU decomposition.

```
igraph_error_t igraph_sparsemat_luresol(const igraph_sparsemat_symbolic_t *dis,
                                        const igraph_sparsemat_numeric_t *din,
                                        const igraph_vector_t *b,
                                        igraph_vector_t *res);
```

Uses the LU decomposition of a matrix to solve linear systems.

Arguments:

dis: The symbolic analysis of the coefficient matrix, the result of `igraph_sparsemat_symblu()`.

din: The LU decomposition, the result of a call to `igraph_sparsemat_lu()`.

b: A vector that defines the right hand side of the linear equation system.

res: An initialized vector, the solution of the linear system is stored here.

Returns:

Error code.

Time complexity: TODO.

`igraph_sparsemat_qrresol` — Solves a linear system using a precomputed QR decomposition.

```
igraph_error_t igraph_sparsemat_qrresol(const igraph_sparsemat_symbolic_t *dis,
                                        const igraph_sparsemat_numeric_t *din,
                                        const igraph_vector_t *b,
                                        igraph_vector_t *res);
```

Solves a linear system using a QR decomposition of its coefficient matrix.

Arguments:

dis: Symbolic analysis of the coefficient matrix, the result of `igraph_sparsemat_symbqr()`.

din: The QR decomposition of the coefficient matrix, the result of `igraph_sparsemat_qr()`.

b: Vector, giving the right hand side of the linear equation system.

res: An initialized vector, the solution is stored here. It is resized as needed.

Returns:

Error code.

Time complexity: TODO.

`igraph_sparsemat_symbolic_destroy` — Deallocates memory after a symbolic decomposition.

```
void igraph_sparsemat_symbolic_destroy(igraph_sparsemat_symbolic_t *dis);
```

Frees the memory allocated by `igraph_sparsemat_symbqr()` or `igraph_sparsemat_symblu()`.

Arguments:

dis: The symbolic analysis.

Time complexity: O(1).

`igraph_sparsemat_numeric_destroy` — Deallocates memory after a numeric decomposition.

```
void igraph_sparsemat_numeric_destroy(igraph_sparsemat_numeric_t *din);
```

Frees the memoty allocated by `igraph_sparsemat_qr()` or `igraph_sparsemat_lu()`.

Arguments:

din: The LU or QR decomposition.

Time complexity: O(1).

Eigenvalues and eigenvectors

`igraph_sparsemat_arpack_rssolve` — Eigenvalues and eigenvectors of a symmetric sparse matrix via ARPACK.

```
igraph_error_t igraph_sparsemat_arpack_rssolve(const igraph_sparsemat_t *A,
                                       igraph_arpack_options_t *options,
                                       igraph_arpack_storage_t *storage,
                                       igraph_vector_t *values,
                                       igraph_matrix_t *vectors,
                                       igraph_sparsemat_solve_t solvemethod);
```

Arguments:

A: The input matrix, must be column-compressed.

options: It is passed to `igraph_arpack_rssolve()`. Supply `NULL` here to use the defaults. See `igraph_arpack_options_t` for the details. If `mode` is 1, then ARPACK uses regular mode, if `mode` is 3, then shift and invert mode is used and the `sigma` structure member defines the shift.

storage: Storage for ARPACK. See `igraph_arpack_rssolve()` and `igraph_arpack_storage_t` for details.

values: An initialized vector or a null pointer, the eigenvalues are stored here.

vectors: An initialised matrix, or a null pointer, the eigenvectors are stored here, in the columns.

solvemethod: The method to solve the linear system, if `mode` is 3, i.e. the shift and invert mode is used. Possible values:

`IGRAPH_SPARSEMAT_SOLVE_LU`	The linear system is solved using LU decomposition.
`IGRAPH_SPARSEMAT_SOLVE_QR`	The linear system is solved using QR decomposition.

Returns:

Error code.

Time complexity: TODO.

`igraph_sparsemat_arpack_rnsolve` — Eigenvalues and eigenvectors of a nonsymmetric sparse matrix via ARPACK.

```
igraph_error_t igraph_sparsemat_arpack_rnsolve(const igraph_sparsemat_t *A,
                                       igraph_arpack_options_t *options,
                                       igraph_arpack_storage_t *storage,
                                       igraph_matrix_t *values,
                                       igraph_matrix_t *vectors);
```

Eigenvalues and/or eigenvectors of a nonsymmetric sparse matrix.

Arguments:

A: The input matrix, in column-compressed mode.

options: ARPACK options, it is passed to `igraph_arpack_rnsolve()`. Supply `NULL` here to use the defaults. See also `igraph_arpack_options_t` for details.

storage: Storage for ARPACK, this is passed to `igraph_arpack_rnsolve()`. See `igraph_arpack_storage_t` for details.

values: An initialized matrix, or a null pointer. If not a null pointer, then the eigenvalues are stored here, the first column is the real part, the second column is the imaginary part.

vectors: An initialized matrix, or a null pointer. If not a null pointer, then the eigenvectors are stored here, please see `igraph_arpack_rnsolve()` for the format.

Returns:

Error code.

Time complexity: TODO.

Conversion to other data types

`igraph_matrix_as_sparsemat` — Converts a dense matrix to a sparse matrix.

```
igraph_error_t igraph_matrix_as_sparsemat(igraph_sparsemat_t *res,
                                const igraph_matrix_t *mat,
                                igraph_real_t tol);
```

Arguments:

res: An uninitialized sparse matrix, the result is stored here.

mat: The dense input matrix.

tol: The tolerance for zero comparisons. Values closer than *tol* to zero are considered as zero, and will not be included in the sparse matrix.

Returns:

Error code.

See also:

`igraph_sparsemat_as_matrix()` for the reverse conversion.

Time complexity: O(mn), the number of elements in the dense matrix.

`igraph_sparsemat_as_matrix` — Converts a sparse matrix to a dense matrix.

```
igraph_error_t igraph_sparsemat_as_matrix(igraph_matrix_t *res,
                                const igraph_sparsemat_t *spmat);
```

Arguments:

res: Pointer to an initialized matrix, the result is stored here. It will be resized to the required size.

spmat: The input sparse matrix, in triplet or column-compressed format.

Returns:

Error code.

See also:

`igraph_matrix_as_sparsemat()` for the reverse conversion.

Time complexity: O(mn), the number of elements in the dense matrix.

Writing to a file, or to the screen

`igraph_sparsemat_print` — Prints a sparse matrix to a file.

```
igraph_error_t igraph_sparsemat_print(const igraph_sparsemat_t *A,
                                      FILE *outstream);
```

Only the non-zero entries are printed. This function serves more as a debugging utility, as currently there is no function that could read back the printed matrix from the file.

Arguments:

A: The input matrix, triplet or column-compressed format.

outstream: The stream to print it to.

Returns:

Error code.

Time complexity: O(nz) for triplet matrices, O(n+nz) for column-compressed matrices. nz is the number of non-zero elements, n is the number columns in the matrix.

Stacks

`igraph_stack_init` — Initializes a stack.

```
igraph_error_t igraph_stack_init(igraph_stack_t* s, igraph_int_t capacity);
```

The initialized stack is always empty.

Arguments:

s: Pointer to an uninitialized stack.

capacity: The number of elements to allocate memory for.

Returns:

Error code.

Time complexity: O(*size*).

`igraph_stack_destroy` — Destroys a stack object.

```
void igraph_stack_destroy(igraph_stack_t* s);
```

Deallocate the memory used for a stack. It is possible to reinitialize a destroyed stack again by `igraph_stack_init()`.

Arguments:

s: The stack to destroy.

Time complexity: O(1).

`igraph_stack_reserve` — Reserve memory.

```
igraph_error_t igraph_stack_reserve(igraph_stack_t* s, igraph_int_t capacity);
```

Reserve memory for future use. The actual size of the stack is unchanged.

Arguments:

s: The stack object.

size: The number of elements to reserve memory for. If it is not bigger than the current size then nothing happens.

Returns:

Error code.

Time complexity: should be around O(n), the new allocated size of the stack.

`igraph_stack_empty` — Decides whether a stack object is empty.

```
igraph_bool_t igraph_stack_empty(igraph_stack_t* s);
```

Arguments:

s: The stack object.

Returns:

Boolean, `true` if the stack is empty, `false` otherwise.

Time complexity: O(1).

`igraph_stack_size` — Returns the number of elements in a stack.

```
igraph_int_t igraph_stack_size(const igraph_stack_t* s);
```

Arguments:

s: The stack object.

Returns:

The number of elements in the stack.

Time complexity: O(1).

`igraph_stack_clear` — Removes all elements from a stack.

```
void igraph_stack_clear(igraph_stack_t* s);
```

Arguments:

s: The stack object.

Time complexity: O(1).

`igraph_stack_push` — Places an element on the top of a stack.

```
igraph_error_t igraph_stack_push(igraph_stack_t* s, igraph_real_t elem);
```

The capacity of the stack is increased, if needed.

Arguments:

s: The stack object.

elem: The element to push.

Returns:

Error code.

Time complexity: O(1) is no reallocation is needed, O(n) otherwise, but it is ensured that n push operations are performed in O(n) time.

`igraph_stack_pop` — Removes and returns an element from the top of a stack.

```
igraph_real_t igraph_stack_pop(igraph_stack_t* s);
```

The stack must contain at least one element, call `igraph_stack_empty()` to make sure of this.

Arguments:

s: The stack object.

Returns:

The removed top element.

Time complexity: O(1).

`igraph_stack_top` — Query top element.

```
igraph_real_t igraph_stack_top(const igraph_stack_t* s);
```

Returns the top element of the stack, without removing it. The stack must be non-empty.

Arguments:

s: The stack.

Returns:

The top element.

Time complexity: O(1).

Double-ended queues

This is the classic data type of the double ended queue. Most of the time it is used if a First-In-First-Out (FIFO) behavior is needed. See the operations below.

Example 7.10. File `examples/simple/dqueue.c`

`igraph_dqueue_init` — Initialize a double ended queue (deque).

```
igraph_error_t igraph_dqueue_init(igraph_dqueue_t* q, igraph_int_t capacity);
```

The queue will be always empty.

Arguments:

q: Pointer to an uninitialized deque.

capacity: How many elements to allocate memory for.

Returns:

Error code.

Time complexity: O(*capacity*).

`igraph_dqueue_destroy` — Destroy a double ended queue.

```
void igraph_dqueue_destroy(igraph_dqueue_t* q);
```

Arguments:

q: The queue to destroy.

Time complexity: O(1).

`igraph_dqueue_empty` — Decide whether the queue is empty.

```
igraph_bool_t igraph_dqueue_empty(const igraph_dqueue_t* q);
```

Arguments:

q: The queue.

Returns:

Boolean, true if *q* contains at least one element, false otherwise.

Time complexity: O(1).

`igraph_dqueue_full` — Check whether the queue is full.

```
igraph_bool_t igraph_dqueue_full(igraph_dqueue_t* q);
```

If a queue is full the next `igraph_dqueue_push()` operation will allocate more memory.

Arguments:

q: The queue.

Returns:

`true` if *q* is full, `false` otherwise.

Time complecity: O(1).

`igraph_dqueue_clear` — Remove all elements from the queue.

```
void igraph_dqueue_clear(igraph_dqueue_t* q);
```

Arguments:

q: The queue.

Time complexity: O(1).

`igraph_dqueue_size` — Number of elements in the queue.

```
igraph_int_t igraph_dqueue_size(const igraph_dqueue_t* q);
```

Arguments:

q: The queue.

Returns:

Integer, the number of elements currently in the queue.

Time complexity: O(1).

igraph_dqueue_head — Head of the queue.

```
igraph_real_t igraph_dqueue_head(const igraph_dqueue_t* q);
```

The queue must contain at least one element.

Arguments:

q: The queue.

Returns:

The first element in the queue.

Time complexity: O(1).

igraph_dqueue_back — Tail of the queue.

```
igraph_real_t igraph_dqueue_back(const igraph_dqueue_t* q);
```

The queue must contain at least one element.

Arguments:

q: The queue.

Returns:

The last element in the queue.

Time complexity: O(1).

igraph_dqueue_get — Access an element in a queue.

```
igraph_real_t igraph_dqueue_get(const igraph_dqueue_t *q, igraph_int_t idx);
```

Arguments:

q: The queue.

idx: The index of the element within the queue.

Returns:

The desired element.

Time complexity: O(1).

igraph_dqueue_pop — Remove the head.

```
igraph_real_t igraph_dqueue_pop(igraph_dqueue_t* q);
```

Removes and returns the first element in the queue. The queue must be non-empty.

Arguments:

q: The input queue.

Returns:

The first element in the queue.

Time complexity: O(1).

igraph_dqueue_pop_back — Removes the tail.

```
igraph_real_t igraph_dqueue_pop_back(igraph_dqueue_t* q);
```

Removes and returns the last element in the queue. The queue must be non-empty.

Arguments:

q: The queue.

Returns:

The last element in the queue.

Time complexity: O(1).

igraph_dqueue_push — Appends an element.

```
igraph_error_t igraph_dqueue_push(igraph_dqueue_t* q, igraph_real_t elem);
```

Append an element to the end of the queue.

Arguments:

q: The queue.

elem: The element to append.

Returns:

Error code.

Time complexity: O(1) if no memory allocation is needed, O(n), the number of elements in the queue otherwise. But note that by allocating always twice as much memory as the current size of the queue we ensure that n push operations can always be done in at most O(n) time. (Assuming memory allocation is at most linear.)

Maximum and minimum heaps

`igraph_heap_init` — Initializes an empty heap object.

```
igraph_error_t igraph_heap_init(igraph_heap_t* h, igraph_int_t capacity);
```

Creates an *empty* heap, and also allocates memory for some elements.

Arguments:

h: Pointer to an uninitialized heap object.

capacity: Number of elements to allocate memory for.

Returns:

Error code.

Time complexity: O(*alloc_size*), assuming memory allocation is a linear operation.

`igraph_heap_init_array` — Build a heap from an array.

```
igraph_error_t igraph_heap_init_array(igraph_heap_t *h, const igraph_real_t *da
```

Initializes a heap object from an array. The heap is also built of course (constructor).

Arguments:

h: Pointer to an uninitialized heap object.

data: Pointer to an array of base data type.

len: The length of the array at *data*.

Returns:

Error code.

Time complexity: O(n), the number of elements in the heap.

`igraph_heap_destroy` — Destroys an initialized heap object.

```
void igraph_heap_destroy(igraph_heap_t* h);
```

Arguments:

h: The heap object.

Time complexity: O(1).

`igraph_heap_clear` — Removes all elements from a heap.

```
void igraph_heap_clear(igraph_heap_t* h);
```

This function simply sets the size of the heap to zero, it does not free any allocated memory. For that you have to call `igraph_heap_destroy()`.

Arguments:

h: The heap object.

Time complexity: O(1).

`igraph_heap_empty` — Decides whether a heap object is empty.

```
igraph_bool_t igraph_heap_empty(const igraph_heap_t* h);
```

Arguments:

h: The heap object.

Returns:

`true` if the heap is empty, `false` otherwise.

TIme complexity: O(1).

`igraph_heap_push` — Add an element.

```
igraph_error_t igraph_heap_push(igraph_heap_t* h, igraph_real_t elem);
```

Adds an element to the heap.

Arguments:

h: The heap object.

elem: The element to add.

Returns:

Error code.

Time complexity: O(log n), n is the number of elements in the heap if no reallocation is needed, O(n) otherwise. It is ensured that n push operations are performed in O(n log n) time.

`igraph_heap_top` — Top element.

```
igraph_real_t igraph_heap_top(const igraph_heap_t* h);
```

For maximum heaps this is the largest, for minimum heaps the smallest element of the heap.

Arguments:

h: The heap object.

Returns:

The top element.

Time complexity: O(1).

`igraph_heap_delete_top` — Removes and returns the top element.

```
igraph_real_t igraph_heap_delete_top(igraph_heap_t* h);
```

Removes and returns the top element of the heap. For maximum heaps this is the largest, for minimum heaps the smallest element.

Arguments:

h: The heap object.

Returns:

The top element.

Time complexity: O(log n), n is the number of elements in the heap.

`igraph_heap_size` — Number of elements in the heap.

```
igraph_int_t igraph_heap_size(const igraph_heap_t* h);
```

Gives the number of elements in a heap.

Arguments:

h: The heap object.

Returns:

The number of elements in the heap.

Time complexity: O(1).

`igraph_heap_reserve` — Reserves memory for a heap.

```
igraph_error_t igraph_heap_reserve(igraph_heap_t* h, igraph_int_t capacity);
```

Allocates memory for future use. The size of the heap is unchanged. If the heap is larger than the *`capacity`* parameter then nothing happens.

Arguments:

`h`: The heap object.

`capacity`: The number of elements to allocate memory for.

Returns:

Error code.

Time complexity: O(*`capacity`*) if *`capacity`* is larger than the current number of elements. O(1) otherwise.

String vectors

The igraph_strvector_t type is a vector of null-terminated strings. It is used internally for storing graph attribute names as well as string attributes in the C attribute handler.

This container automatically manages the memory of its elements. The strings within an igraph_strvector_t should be considered constant, and not modified directly. Functions that add new elements always make copies of the string passed to them.

Example 7.11. File `examples/simple/igraph_strvector.c`

`igraph_strvector_init` — Initializes a string vector.

```
igraph_error_t igraph_strvector_init(igraph_strvector_t *sv, igraph_int_t size)
```

Reserves memory for the string vector, a string vector must be first initialized before calling other functions on it. All elements of the string vector are set to the empty string.

Arguments:

`sv`: Pointer to an initialized string vector.

`size`: The (initial) length of the string vector.

Returns:

Error code.

Time complexity: O(*`len`*).

`igraph_strvector_init_copy` — Initialization by copying.

```
igraph_error_t igraph_strvector_init_copy(igraph_strvector_t *to,
                                          const igraph_strvector_t *from);
```

Initializes a string vector by copying another string vector.

Arguments:

to: Pointer to an uninitialized string vector.

from: The other string vector, to be copied.

Returns:

Error code.

Time complexity: O(l), the total length of the strings in *from*.

igraph_strvector_destroy — Frees the memory allocated for the string vector.

```
void igraph_strvector_destroy(igraph_strvector_t *sv);
```

Destroy a string vector. It may be reinitialized with `igraph_strvector_init()` later.

Arguments:

sv: The string vector.

Time complexity: O(l), the total length of the strings, maybe less depending on the memory manager.

STR — Indexing string vectors.

```
#define STR(sv,i)
```

This is a macro that allows to query the elements of a string vector, just like `igraph_strvector_get()`. Note this macro cannot be used to set an element. Use `igraph_strvector_set()` to set an element instead.

Arguments:

sv: The string vector

i: The index of the element.

Returns:

The element at position *i*.

Time complexity: O(1).

Warning

Deprecated since version 0.10.9. Please do not use this function in new code; use `igraph_strvector_get()` instead.

`igraph_strvector_get` — Retrieves an element of a string vector.

```
const char *igraph_strvector_get(const igraph_strvector_t *sv, igraph_int_t idx
```

Query an element of a string vector. The returned string must not be modified.

Arguments:

sv: The input string vector.

idx: The index of the element to query.

Time complexity: O(1).

`igraph_strvector_set` — Sets an element of the string vector from a string.

```
igraph_error_t igraph_strvector_set(igraph_strvector_t *sv, igraph_int_t idx,
                          const char *value);
```

The provided *value* is copied into the *idx* position in the string vector.

Arguments:

sv: The string vector.

idx: The position to set.

value: The new value.

Returns:

Error code.

Time complexity: O(l), the length of the new string. Maybe more, depending on the memory management, if reallocation is needed.

`igraph_strvector_set_len` — Sets an element of the string vector given a buffer and its size.

```
igraph_error_t igraph_strvector_set_len(igraph_strvector_t *sv, igraph_int_t id
                           const char *value, size_t len);
```

This is almost the same as `igraph_strvector_set`, but the new value is not a zero terminated string, but its length is given.

Arguments:

sv: The string vector.

idx: The position to set.

value: The new value.

len: The length of the new value.

Returns:

Error code.

Time complexity: O(l), the length of the new string. Maybe more, depending on the memory management, if reallocation is needed.

igraph_strvector_push_back — Adds an element to the back of a string vector.

```
igraph_error_t igraph_strvector_push_back(igraph_strvector_t *sv, const char *va
```

Arguments:

sv: The string vector.

value: The string to add; it will be copied.

Returns:

Error code.

Time complexity: O(n+l), n is the total number of strings, l is the length of the new string.

igraph_strvector_push_back_len — Adds a string of the given length to the back of a string vector.

```
igraph_error_t igraph_strvector_push_back_len(
        igraph_strvector_t *sv,
        const char *value, size_t len);
```

Arguments:

sv: The string vector.

value: The start of the string to add. At most *len* characters will be copied.

len: The length of the string.

Returns:

Error code.

Time complexity: O(n+l), n is the total number of strings, l is the length of the new string.

igraph_strvector_swap_elements — Swap two elements in a string vector.

```
void igraph_strvector_swap_elements(igraph_strvector_t *sv, igraph_int_t i, igra
```

Note that currently no range checking is performed.

Arguments:

sv: The string vector.

i: Index of the first element.

j: Index of the second element (may be the same as the first one).

Time complexity: O(1).

igraph_strvector_remove — Removes a single element from a string vector.

```
void igraph_strvector_remove(igraph_strvector_t *sv, igraph_int_t elem);
```

The string will be one shorter.

Arguments:

sv: The string vector.

elem: The index of the element to remove.

Time complexity: O(n), the length of the string.

igraph_strvector_remove_section — Removes a section from a string vector.

```
void igraph_strvector_remove_section(
        igraph_strvector_t *sv, igraph_int_t from, igraph_int_t to);
```

This function removes the range `[from, to)` from the string vector.

Arguments:

sv: The string vector.

from: The position of the first element to remove.

to: The position of the first element *not* to remove.

igraph_strvector_append — Concatenates two string vectors.

```
igraph_error_t igraph_strvector_append(igraph_strvector_t *to,
                                       const igraph_strvector_t *from);
```

Appends the contents of the *from* vector to the *to* vector. If the *from* vector is no longer needed after this operation, use `igraph_strvector_merge()` for better performance.

Arguments:

to: The first string vector, the result is stored here.

from: The second string vector, it is kept unchanged.

Returns:

Error code.

See also:

`igraph_strvector_merge()`

Time complexity: O(n+l2), n is the number of strings in the new string vector, l2 is the total length of strings in the *from* string vector.

`igraph_strvector_merge` — Moves the contents of a string vector to the end of another.

```
igraph_error_t igraph_strvector_merge(igraph_strvector_t *to, igraph_strvector_
```

Transfers the contents of the *from* vector to the end of *to*, clearing *from* in the process. If this operation fails, both vectors are left intact. This function does not copy or reallocate individual strings, therefore it performs better than `igraph_strvector_append()`.

Arguments:

to: The target vector. The contents of *from* will be appended to it.

from: The source vector. It will be cleared.

Returns:

Error code.

See also:

`igraph_strvector_append()`

Time complexity: O(l2) if *to* has sufficient capacity, O(2*l1+l2) otherwise, where l1 and l2 are the lengths of *to* and \from respectively.

`igraph_strvector_swap` — Swaps all elements of two string vectors.

```
void igraph_strvector_swap(igraph_strvector_t *v1, igraph_strvector_t *v2);
```

Arguments:

v1: The first string vector.

v2: The second string vector.

Time complexity: O(1).

igraph_strvector_update — Updates a string vector from another one.

```
igraph_error_t igraph_strvector_update(
    igraph_strvector_t *to, const igraph_strvector_t *from
);
```

After this operation the contents of *to* will be exactly the same as that of *from*. The vector *to* will be resized if it was originally shorter or longer than *from*.

Arguments:

to: The string vector to update.

from: The string vector to update from.

Returns:

Error code.

igraph_strvector_clear — Removes all elements from a string vector.

```
void igraph_strvector_clear(igraph_strvector_t *sv);
```

After this operation the string vector will be empty.

Arguments:

sv: The string vector.

Time complexity: O(l), the total length of strings, maybe less, depending on the memory manager.

igraph_strvector_resize — Resizes a string vector.

```
igraph_error_t igraph_strvector_resize(igraph_strvector_t *sv, igraph_int_t news
```

If the new size is bigger then empty strings are added, if it is smaller then the unneeded elements are removed.

Arguments:

sv: The string vector.

newsize: The new size.

Returns:

Error code.

Time complexity: O(n), the number of strings if the vector is made bigger, O(l), the total length of the deleted strings if it is made smaller, maybe less, depending on memory management.

igraph_strvector_reserve — Reserves memory for a string vector.

```
igraph_error_t igraph_strvector_reserve(igraph_strvector_t *sv, igraph_int_t ca
```

igraph string vectors are flexible, they can grow and shrink. Growing however occasionally needs the data in the vector to be copied. In order to avoid this, you can call this function to reserve space for future growth of the vector.

Note that this function does *not* change the size of the string vector. Let us see a small example to clarify things: if you reserve space for 100 strings and the size of your vector was (and still is) 60, then you can surely add additional 40 strings to your vector before it will be copied.

Arguments:

sv: The string vector object.

capacity: The new *allocated* size of the string vector.

Returns:

Error code: IGRAPH_ENOMEM if there is not enough memory.

Time complexity: operating system dependent, should be around O(n), n is the new allocated size of the vector.

igraph_strvector_resize_min — Deallocates the unused memory of a string vector.

```
void igraph_strvector_resize_min(igraph_strvector_t *sv);
```

This function attempts to deallocate the unused reserved storage of a string vector. If it succeeds, igraph_strvector_size() and igraph_strvector_capacity() will be the same. The data in the string vector is always preserved, even if deallocation is not successful.

Arguments:

sv: The string vector.

Time complexity: Operating system dependent, at most O(n).

igraph_strvector_size — Returns the size of a string vector.

```
igraph_int_t igraph_strvector_size(const igraph_strvector_t *sv);
```

Arguments:

sv: The string vector.

Returns:

The length of the string vector.

Time complexity: O(1).

`igraph_strvector_capacity` — Returns the capacity of a string vector.

```
igraph_int_t igraph_strvector_capacity(const igraph_strvector_t *sv);
```

Arguments:

sv: The string vector.

Returns:

The capacity of the string vector.

Time complexity: O(1).

Lists of vectors, matrices and graphs

About igraph_vector_list_t objects

The igraph_vector_list_t data type is essentially a list of igraph_vector_t objects with automatic memory management. It is something similar to (but much simpler than) the vector template in the C++ standard library where the elements are vectors themselves.

There are multiple variants of igraph_vector_list_t; the basic variant stores vectors of doubles (i.e. each item is an `igraph_vector_t`), but there is also igraph_vector_int_list_t for integers (where each item is an igraph_vector_int_t), igraph_matrix_list_t for matrices of doubles and so on. The following list summarizes the variants that are currently available in the library:

- igraph_vector_list_t for lists of vectors of floating-point numbers (igraph_vector_t)
- igraph_vector_int_list_t for lists of integer vectors (igraph_vector_int_t)
- igraph_matrix_list_t for lists of matrices of floating-point numbers (igraph_matrix_t)
- igraph_graph_list_t for lists of graphs (igraph_t)

Lists of vectors are used in **igraph** in many cases, e.g., when returning lists of paths, cliques or vertex sets. Functions that expect or return a list of numeric vectors typically use igraph_vector_list_t or igraph_vector_int_list_t to achieve this. Lists of integer vectors are used when the vectors in the list are supposed to hold vertex or edge identifiers, while lists of floating-point vectors are used when the vectors are expected to hold fractional numbers or infinities.

The elements in an igraph_vector_list_t object and its variants are indexed from zero, we follow the usual C convention here.

Almost all of the functions described below for igraph_vector_list_t also exist for all the other vector list variants. These variants are not documented separately; you can simply replace `vector_list` with, say, `vector_int_list` if you need a function for another variant. For instance, to initialize a list of integer vectors, you need to use `igraph_vector_int_list_init`() and not `igraph_vector_list_init()`.

Before diving into a detailed description of the functions related to lists of vectors, we must also talk about the *ownership* rules of these objects. The most important rule is that the vectors in the list are owned by the list itself, meaning that the user is *not* responsible for allocating memory for the vectors or for freeing the memory associated to the vectors. It is the responsibility of the list to allocate and initialize the vectors when new items are created in the list, and it is also the responsibility of the list to destroy the items when they are removed from the list without passing on their ownership to the user. As a consequence, the list may not contain "uninitialized" or "null" items; each item is initialized when it comes to existence. If you create a list containing one million vectors, you are not only allocating memory for one million `igraph_vector_t` object but you are also initializing one million vectors. Also, if you have a list containing one million vectors and you clear the list by calling `igraph_vector_list_clear()`, the list will implicitly destroy these lists, and any pointers that you may hold to the items become invalid at once.

Speaking about pointers, the typical way of working with vectors in a list is to obtain a pointer to one of the items via the `igraph_vector_list_get_ptr()` method and then passing this pointer onwards to functions that manipulate `igraph_vector_t` objects. However, note that the pointers are *ephemeral* in the sense that they may be invalidated any time when the list is modified because a modification may involve the re-allocation of the internal storage of the list if more space is needed, and the pointers that you obtained will not follow the reallocation. This limitation does not appear often in real-world usage of `igraph_vector_list_t` and in general, the advantages of the automatic memory management outweigh this limitation.

Constructors and destructors

igraph_vector_list_t objects have to be initialized before using them, this is analogous to calling a constructor on them. `igraph_vector_list_init()` is the basic constructor; it creates a list of the given length and also initializes each vector in the newly created list to zero length.

If an igraph_vector_list_t object is not needed any more, it should be destroyed to free its allocated memory by calling the igraph_vector_list_t destructor, `igraph_vector_list_destroy()`. Calling the destructor also destroys all the vectors inside the vector list due to the ownership rules. If you want to keep a few of the vectors in the vector list, you need to copy them with `igraph_vector_init_copy()` or `igraph_vector_update()`, or you need to remove them from the list and take ownership by calling `igraph_vector_list_pop_back()`, `igraph_vector_list_remove()` or `igraph_vector_list_remove_fast()` .

`igraph_vector_list_init` — Initializes a list of vectors (constructor).

```
igraph_error_t igraph_vector_list_init(igraph_vector_list_t *v, igraph_int_t si
```

This function constructs a list of vectors of the given size, and initializes each vector in the newly created list to become an empty vector.

Vector objects initialized by this function are *owned* by the list, and they will be destroyed automatically when the list is destroyed with `igraph_vector_list_destroy()`.

Arguments:

v: Pointer to a not yet initialized list of vectors.

size: The size of the list.

Returns:

error code: `IGRAPH_ENOMEM` if there is not enough memory.

Time complexity: operating system dependent, the amount of “time” required to allocate O(n) elements and initialize the corresponding vectors; n is the number of elements.

igraph_vector_list_init_copy — Initializes a list of vectors from another list of vectors (constructor).

```
igraph_error_t igraph_vector_list_init_copy(igraph_vector_list_t* to, const igra
```

The contents of the existing list will be copied to the new one.

Arguments:

to: Pointer to a not yet initialized list of vectors.

from: The original list of vectors to copy.

Returns:

Error code: IGRAPH_ENOMEM if there is not enough memory.

Time complexity: operating system dependent, usually O(nm), n is the size of the list, m is the size of each element in the list, assuming that copying a single item takes O(m) time.

igraph_vector_list_destroy — Destroys a list of vectors object.

```
void igraph_vector_list_destroy(igraph_vector_list_t *v);
```

All lists initialized by igraph_vector_list_init() should be properly destroyed by this function. A destroyed list of vectors needs to be reinitialized by igraph_vector_list_init() if you want to use it again.

Vectors that are in the list when it is destroyed are also destroyed implicitly.

Arguments:

v: Pointer to the (previously initialized) list object to destroy.

Time complexity: operating system dependent.

Accessing elements

Elements of a vector list may be accessed with the igraph_vector_list_get_ptr() function. The function returns a *pointer* to the vector with a given index inside the list, and you may then pass this pointer onwards to other functions that can query or manipulate vectors. The pointer itself is guaranteed to stay valid as long as the list itself is not modified; however, *any* modification to the list will invalidate the pointer, even modifications that are seemingly unrelated to the vector that the pointer points to (such as adding a new vector at the end of the list). This is because the list data structure may be forced to re-allocate its internal storage if a new element does not fit into the already allocated space, and there are no guarantees that the re-allocated block remains at the same memory location (typically it gets moved elsewhere).

Note that the standard VECTOR macro that works for ordinary vectors does not work for lists of vectors to access the i-th element (but of course you can use it to index into an existing vector that you

retrieved from the vector list with `igraph_vector_list_get_ptr()`). This is because the macro notation would allow one to overwrite the vector in the list with another one without the list knowing about it, so the list would not be able to destroy the vector that was overwritten by a new one.

`igraph_vector_list_tail_ptr()` returns a pointer to the last vector in the list, or `NULL` if the list is empty. There is no `igraph_vector_list_head_ptr()`, however, as it is easy to write `igraph_vector_list_get_ptr(v, 0)` instead.

`igraph_vector_list_get_ptr` — The address of a vector in the vector list.

```
igraph_vector_t *igraph_vector_list_get_ptr(const igraph_vector_list_t *v, igra
```

Arguments:

v: The list object.

pos: The position of the vector in the list. The position of the first vector is zero.

Returns:

A pointer to the vector. It remains valid as long as the underlying list of vectors is not modified.

Time complexity: O(1).

`igraph_vector_list_tail_ptr` — The address of the last vector in the vector list.

```
igraph_vector_t *igraph_vector_list_tail_ptr(const igraph_vector_list_t *v);
```

Arguments:

v: The list object.

Returns:

A pointer to the last vector in the list, or `NULL` if the list is empty.

Time complexity: O(1).

`igraph_vector_list_set` — Sets the vector at the given index in the list.

```
void igraph_vector_list_set(igraph_vector_list_t *v, igraph_int_t pos, igraph_v
```

This function destroys the vector that is already at the given index *pos* in the list, and replaces it with the vector pointed to by *e*. The ownership of the vector pointed to by *e* is taken by the list so the user is not responsible for destroying *e* any more; it will be destroyed when the list itself is destroyed or if *e* gets removed from the list without passing on the ownership to somewhere else.

Arguments:

v: The list object.

pos: The index to modify in the list.

e: The vector to set in the list.

Time complexity: O(1).

igraph_vector_list_replace — Replaces the vector at the given index in the list with another one.

```
void igraph_vector_list_replace(igraph_vector_list_t *v, igraph_int_t pos, igra
```

This function replaces the vector that is already at the given index *pos* in the list with the vector pointed to by *e*. The ownership of the vector pointed to by *e* is taken by the list so the user is not responsible for destroying *e* any more. At the same time, the ownership of the vector that *was* in the list at position *pos* will be transferred to the caller and *e* will be updated to point to it, so the caller becomes responsible for destroying it when it does not need the vector any more.

Arguments:

v: The list object.

pos: The index to modify in the list.

e: The vector to swap with the one already in the list.

Time complexity: O(1).

Vector properties

igraph_vector_list_empty — Decides whether the size of the list is zero.

```
igraph_bool_t igraph_vector_list_empty(const igraph_vector_list_t *v);
```

Arguments:

v: The list object.

Returns:

True if the size of the list is zero and false otherwise.

Time complexity: O(1).

igraph_vector_list_size — The size of the vector list.

```
igraph_int_t igraph_vector_list_size(const igraph_vector_list_t *v);
```

Returns the number of vectors stored in the list.

Arguments:

v: The list object

Returns:

The size of the list.

Time complexity: O(1).

`igraph_vector_list_capacity` — Returns the allocated capacity of the list.

```
igraph_int_t igraph_vector_list_capacity(const igraph_vector_list_t *v);
```

Note that this might be different from the size of the list (as queried by `igraph_vector_list_size()`), and specifies how many vectors the list can hold, without reallocation.

Arguments:

v: Pointer to the (previously initialized) list object to query.

Returns:

The allocated capacity.

See also:

`igraph_vector_list_size()`.

Time complexity: O(1).

Resizing operations

`igraph_vector_list_clear` — Removes all elements from a list of vectors.

```
void igraph_vector_list_clear(igraph_vector_list_t *v);
```

This function sets the size of the list to zero, and it also destroys all the vectors that were placed in the list before clearing it.

Arguments:

v: The list object.

Time complexity: O(n), n is the number of items being deleted.

`igraph_vector_list_reserve` — Reserves memory for a list.

```
igraph_error_t igraph_vector_list_reserve(igraph_vector_list_t *v, igraph_int_t
```

igraph lists are flexible, they can grow and shrink. Growing however occasionally needs the data in the list to be copied. In order to avoid this, you can call this function to reserve space for future growth of the list.

Note that this function does *not* change the size of the list, neither does it initialize any new vectors. Let us see a small example to clarify things: if you reserve space for 100 elements and the size of your list was (and still is) 60, then you can surely add additional 40 new vectors to your list before it will be copied.

Arguments:

v: The list object.

capacity: The new *allocated* size of the list.

Returns:

Error code: `IGRAPH_ENOMEM` if there is not enough memory.

Time complexity: operating system dependent, should be around O(n), n is the new allocated size of the list.

`igraph_vector_list_resize` — Resizes the list of vectors.

```
igraph_error_t igraph_vector_list_resize(igraph_vector_list_t *v, igraph_int_t
```

Note that this function does not free any memory, just sets the size of the list to the given one. It can on the other hand allocate more memory if the new size is larger than the previous one.

When the new size is larger than the current size, the newly added vectors in the list are initialized to empty vectors. When the new size is smaller than the current size, the vectors that were removed from the end of the list are destroyed automatically.

Arguments:

v: The list object

new_size: The new size of the list.

Returns:

Error code, `IGRAPH_ENOMEM` if there is not enough memory. Note that this function *never* returns an error if the list is made smaller.

See also:

`igraph_vector_list_reserve()` for allocating memory for future extensions of a list.

Time complexity: O(m) if the new size is smaller (m is the number of items that were removed from the list), operating system dependent if the new size is larger. In the latter case it is usually around O(n), where n is the new size of the vector.

`igraph_vector_list_push_back` — Appends an existing vector to the list, transferring ownership.

```
igraph_error_t igraph_vector_list_push_back(igraph_vector_list_t *v, igraph_vec
```

This function resizes the list to be one element longer, and sets the very last element in the list to the specified vector *e* . The list takes ownership of the vector so the user is not responsible for freeing *e*

any more; the vector will be destroyed when the list itself is destroyed or if *e* gets removed from the list without passing on the ownership to somewhere else.

Arguments:

v: The list object.

e: Pointer to the vector to append to the list.

Returns:

Error code: `IGRAPH_ENOMEM`: not enough memory.

Time complexity: operating system dependent. What is important is that a sequence of n subsequent calls to this function has time complexity O(n), even if there hadn't been any space reserved for the new elements by `igraph_vector_list_reserve()`. This is implemented by a trick similar to the C++ vector class: each time more memory is allocated for a vector, the size of the additionally allocated memory is the same as the vector's current length. (We assume here that the time complexity of memory allocation is at most linear).

`igraph_vector_list_push_back_copy` — Appends the copy of a vector to the list.

```
igraph_error_t igraph_vector_list_push_back_copy(igraph_vector_list_t *v, const
```

This function resizes the list to be one element longer, and copies the specified vector given as an argument to the last element. The newly added element is owned by the list, but the ownership of the original vector is retained at the caller.

Arguments:

v: The list object.

e: Pointer to the vector to copy to the end of the list.

Returns:

Error code: `IGRAPH_ENOMEM`: not enough memory.

Time complexity: same as `igraph_vector_list_push_back()` plus the time needed to copy the vector (which is O(n) for n elements in the vector).

`igraph_vector_list_push_back_new` — Appends a new vector to the list.

```
igraph_error_t igraph_vector_list_push_back_new(igraph_vector_list_t *v, igraph
```

This function resizes the list to be one element longer. The newly added element will be an empty vector that is owned by the list. A pointer to the newly added element is returned in the last argument if it is not `NULL` .

Arguments:

v: The list object.

result: Pointer to a vector pointer; this will be updated to point to the newly added vector. May be `NULL` if you do not need a pointer to the newly added vector.

Returns:

Error code: `IGRAPH_ENOMEM`: not enough memory.

Time complexity: same as `igraph_vector_list_push_back()`.

`igraph_vector_list_pop_back` — Removes the last item from the vector list and transfer ownership to the caller.

```
igraph_vector_t igraph_vector_list_pop_back(igraph_vector_list_t *v);
```

This function removes the last vector from the list. The vector that was removed from the list is returned and its ownership is passed back to the caller; in other words, the caller becomes responsible for destroying the vector when it is not needed any more.

It is an error to call this function with an empty vector.

Arguments:

v: The list object.

result: Pointer to an `igraph_vector_t` object; it will be updated to the item that was removed from the list. Ownership of this vector is passed on to the caller.

Time complexity: O(1).

`igraph_vector_list_insert` — Inserts an existing vector into the list, transferring ownership.

```
igraph_error_t igraph_vector_list_insert(igraph_vector_list_t *v, igraph_int_t
```

This function inserts *e* into the list at the given index, moving other items towards the end of the list as needed. The list takes ownership of the vector so the user is not responsible for freeing *e* any more; the vector will be destroyed when the list itself is destroyed or if *e* gets removed from the list without passing on the ownership to somewhere else.

Arguments:

v: The list object.

pos: The position where the new element is to be inserted.

e: Pointer to the vector to insert into the list.

Returns:

Error code: `IGRAPH_ENOMEM`: not enough memory.

Time complexity: O(n).

`igraph_vector_list_insert_copy` — Inserts the copy of a vector to the list.

```
igraph_error_t igraph_vector_list_insert_copy(igraph_vector_list_t *v, igraph_i
```

This function inserts a copy of *e* into the list at the given index, moving other items towards the end of the list as needed. The newly added element is owned by the list, but the ownership of the original vector is retained at the caller.

Arguments:

v: The list object.

pos: The position where the new element is to be inserted.

e: Pointer to the vector to copy to the end of the list.

Returns:

Error code: `IGRAPH_ENOMEM`: not enough memory.

Time complexity: same as `igraph_vector_list_insert()` plus the time needed to copy the vector (which is O(n) for n elements in the vector).

`igraph_vector_list_insert_new` — Inserts a new vector into the list.

```
igraph_error_t igraph_vector_list_insert_new(igraph_vector_list_t *v, igraph_in
```

This function inserts a newly created empty vector into the list at the given index, moving other items towards the end of the list as needed. The newly added vector is owned by the list. A pointer to the new element is returned in the last argument if it is not `NULL` .

Arguments:

v: The list object.

pos: The position where the new element is to be inserted.

result: Pointer to a vector pointer; this will be updated to point to the newly added vector. May be `NULL` if you do not need a pointer to the newly added vector.

Returns:

Error code: `IGRAPH_ENOMEM`: not enough memory.

Time complexity: same as `igraph_vector_list_push_back()`.

`igraph_vector_list_remove` — Removes the item at the given index from the vector list and transfer ownership to the caller.

```
igraph_error_t igraph_vector_list_remove(igraph_vector_list_t *v, igraph_int_t
```

This function removes the vector at the given index from the list, and moves all subsequent items in the list by one slot to the left to fill the gap. The vector that was removed from the list is returned in *e* and its ownership is passed back to the caller; in other words, the caller becomes responsible for destroying the vector when it is not needed any more.

Arguments:

v: The list object.

index: Index of the item to be removed.

result: Pointer to an `igraph_vector_t` object; it will be updated to the item that was removed from the list. Ownership of this vector is passed on to the caller. It is an error to supply a null pointer here.

See also:

`igraph_vector_list_discard()` if you are not interested in the item that was removed, `igraph_vector_list_remove_fast()` if you do not care about the order of the items in the list.

Time complexity: O(n), where n is the number of items in the list.

`igraph_vector_list_remove_fast` — Removes the item at the given index in the vector list, move the last item to its place and transfer ownership to the caller.

```
igraph_error_t igraph_vector_list_remove_fast(igraph_vector_list_t *v, igraph_i
```

This function removes the vector at the given index from the list, moves the last item in the list to *index* to fill the gap, and then transfers ownership of the removed vector back to the caller; in other words, the caller becomes responsible for destroying the vector when it is not needed any more.

Arguments:

v: The list object.

index: Index of the item to be removed.

result: Pointer to an `igraph_vector_t` object; it will be updated to the item that was removed from the list. Ownership of this vector is passed on to the caller. It is an error to supply a null pointer here.

See also:

`igraph_vector_list_remove()` if you want to preserve the order of the items in the list, `igraph_vector_list_discard_fast()` if you are not interested in the item that was removed.

Time complexity: O(1).

`igraph_vector_list_discard` — Discards the item at the given index in the vector list.

```
void igraph_vector_list_discard(igraph_vector_list_t *v, igraph_int_t index);
```

This function removes the vector at the given index from the list, and moves all subsequent items in the list by one slot to the left to fill the gap. The vector that was removed from the list is destroyed automatically.

Arguments:

v: The list object.

index: Index of the item to be discarded and destroyed.

See also:

`igraph_vector_list_discard_fast()` if you do not care about the order of the items in the list, `igraph_vector_list_remove()` if you want to gain ownership of the item that was removed instead of destroying it.

Time complexity: O(n), where n is the number of items in the list.

`igraph_vector_list_discard_back` — Discards the last item in the vector list.

```
void igraph_vector_list_discard_back(igraph_vector_list_t *v);
```

This function removes the last vector from the list and destroys it.

Arguments:

v: The list object.

Time complexity: O(1).

`igraph_vector_list_discard_fast` — Discards the item at the given index in the vector list and moves the last item to its place.

```
void igraph_vector_list_discard_fast(igraph_vector_list_t *v, igraph_int_t inde
```

This function removes the vector at the given index from the list, and moves the last item in the list to *`index`* to fill the gap. The vector that was removed from the list is destroyed automatically.

Arguments:

v: The list object.

`index`: Index of the item to be discarded and destroyed.

See also:

`igraph_vector_list_discard()` if you want to preserve the order of the items in the list, `igraph_vector_list_remove_fast()` if you want to gain ownership of the item that was removed instead of destroying it.

Time complexity: O(1).

Sorting and reordering

`igraph_vector_list_permute` — Permutes the elements of a list in place according to an index vector.

```
igraph_error_t igraph_vector_list_permute(igraph_vector_list_t *v, const igraph_
```

This function takes a list `v` and a corresponding index vector `index`, and permutes the elements of `v` such that `v`[index[i]] is moved to become `v`[i] after the function is executed.

It is an error to call this function with an index vector that does not represent a valid permutation. Each element in the index vector must be between 0 and the length of the list minus one (inclusive), and each such element must appear only once. The function does not attempt to validate the index vector. Memory may be leaked if the index vector does not satisfy these conditions.

The index vector that this function takes is compatible with the index vector returned from `igraph_vector_list_sort_ind()`; passing in the index vector from `igraph_vector_list_sort_ind()` will sort the original vector.

Arguments:

v: the list to permute

index: the index vector

Time complexity: O(n), the number of items in the list.

`igraph_vector_list_sort` — Sorts the elements of the list into ascending order.

```
void igraph_vector_list_sort(igraph_vector_list_t *v, int (*cmp)(const igraph_v
```

Arguments:

v: Pointer to an initialized list object.

cmp: A comparison function that takes pointers to two vectors and returns zero if the two vectors are considered equal, any negative number if the first vector is smaller and any positive number if the second vector is smaller.

Returns:

Error code.

Time complexity: O(n log n) for n elements.

`igraph_vector_list_sort_ind` — Returns a permutation of indices that sorts the list.

```
igraph_error_t igraph_vector_list_sort_ind(
    igraph_vector_list_t *v, igraph_vector_int_t *inds,
    int (*cmp)(const igraph_vector_t*, const igraph_vector_t*)
);
```

Takes an unsorted list *v* as input and computes an array of indices *inds* such that v[inds[i]], with i increasing from 0, is an ordered array according to the comparison function *cmp*. The order of indices for identical elements is not defined.

Arguments:

v: the list to be sorted

inds: the output array of indices. This must be initialized, but will be resized

cmp: A comparison function that takes pointers to two vectors and returns zero if the two vectors are considered equal, any negative number if the first vector is smaller and any positive number if the second vector is smaller.

Returns:

Error code.

Time complexity: O(n log n) for n elements.

`igraph_vector_list_swap` — Swaps all elements of two vector lists.

```
void igraph_vector_list_swap(igraph_vector_list_t *v1, igraph_vector_list_t *v2
```

Arguments:

v1: The first list.

v2: The second list.

Returns:

Error code.

Time complexity: O(1).

`igraph_vector_list_swap_elements` — Swap two elements in a vector list.

```
void igraph_vector_list_swap_elements(igraph_vector_list_t *v1, igraph_int_t i,
```

Note that currently no range checking is performed.

Arguments:

v: The input list.

i: Index of the first element.

j: Index of the second element (may be the same as the first one).

Returns:

Error code, currently always `IGRAPH_SUCCESS`.

Time complexity: O(1).

Adjacency lists

Sometimes it is easier to work with a graph which is in adjacency list format: a list of vectors; each vector contains the neighbor vertices or incident edges of a given vertex. Typically, this representation is good if we need to iterate over the neighbors of all vertices many times. E.g. when finding the shortest paths between all pairs of vertices or calculating closeness centrality for all the vertices.

The igraph_adjlist_t stores the adjacency lists of a graph. After creation it is independent of the original graph, it can be modified freely with the usual vector operations, the graph is not affected. E.g. the adjacency list can be used to rewire the edges of a graph efficiently. If one used the straightforward `igraph_delete_edges()` and `igraph_add_edges()` combination for this that needs O(|V|

+|E|) time for every single deletion and insertion operation, it is thus very slow if many edges are rewired. Extracting the graph into an adjacency list, do all the rewiring operations on the vectors of the adjacency list and then creating a new graph needs (depending on how exactly the rewiring is done) typically O(|V|+|E|) time for the whole rewiring process.

Lazy adjacency lists are a bit different. When creating a lazy adjacency list, the neighbors of the vertices are not queried, only some memory is allocated for the vectors. When `igraph_lazy_adjlist_get()` is called for vertex v the first time, the neighbors of v are queried and stored in a vector of the adjacency list, so they don't need to be queried again. Lazy adjacency lists are handy if you have an at least linear operation (because initialization is generally linear in terms of the number of vertices), but you don't know how many vertices you will visit during the computation.

Example 7.12. File `examples/simple/adjlist.c`

Adjacent vertices

`igraph_adjlist_init` — Constructs an adjacency list of vertices from a given graph.

```
igraph_error_t igraph_adjlist_init(const igraph_t *graph, igraph_adjlist_t *al,
                        igraph_neimode_t mode, igraph_loops_t loops,
                        igraph_bool_t multiple);
```

Creates a list of vectors containing the neighbors of all vertices in a graph. The adjacency list is independent of the graph after creation, e.g. the graph can be destroyed and modified, the adjacency list contains the state of the graph at the time of its initialization.

This function returns each neighbor list in sorted order, just like `igraph_neighbors()`. However, adjacency lists *"in* general" are not guaranteed to be sorted, and we reserve the right to change the ordering of vertices in the result in the future without considering this a breaking change. If you need to ensure that the adjacency lists are sorted, you can use `igraph_adjlist_sort()` to sort all the adjacency lists, or call `igraph_vector_int_sort()` on the individual adjacency vectors after the initialization.

As of igraph 0.10, there is a small performance cost to setting *`loops`* to a different value than `IGRAPH_LOOPS_TWICE` or setting *`multiple`* to a different value from `IGRAPH_MULTIPLE`.

Arguments:

`graph`: The input graph.

`al`: Pointer to an uninitialized igraph_adjlist_t object.

`mode`: Constant specifying whether to include only outgoing (`IGRAPH_OUT`), only incoming (`IGRAPH_IN`), or both (`IGRAPH_ALL`) types of neighbors in the adjacency list. It is ignored for undirected graphs.

`loops`: Specifies how to treat loop edges. `IGRAPH_NO_LOOPS` removes loop edges from the adjacency list. `IGRAPH_LOOPS_ONCE` makes each loop edge appear only once in the adjacency list of the corresponding vertex. `IGRAPH_LOOPS_TWICE` makes loop edges appear *twice* in the adjacency list of the corresponding vertex, but only if the graph is undirected or *`mode`* is set to `IGRAPH_ALL`.

`multiple`: Specifies how to treat multiple (parallel) edges. `IGRAPH_NO_MULTIPLE` collapses parallel edges into a single one; `IGRAPH_MULTIPLE` keeps the multiplicities of parallel edges so the same vertex will appear as many times in the adjacency list of another vertex as the number of parallel edges going between the two vertices.

Returns:

Error code.

See also:

`igraph_neighbors()` for getting the neighbor lists of individual vertices.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

`igraph_adjlist_init_empty` — Initializes an empty adjacency list.

```
igraph_error_t igraph_adjlist_init_empty(igraph_adjlist_t *al, igraph_int_t no_
```

Creates a list of vectors, one for each vertex. This is useful when you are *constructing* a graph using an adjacency list representation as it does not require your graph to exist yet.

Arguments:

al: Pointer to an uninitialized igraph_adjlist_t object.

no_of_nodes: The number of vertices

Returns:

Error code.

Time complexity: O(n), linear in the number of vertices.

`igraph_adjlist_init_complementer` — Adjacency lists for the complementer graph.

```
igraph_error_t igraph_adjlist_init_complementer(const igraph_t *graph,
                                        igraph_adjlist_t *al,
                                        igraph_neimode_t mode,
                                        igraph_loops_t loops);
```

This function creates adjacency lists for the complementer of the input graph. In the complementer graph all edges are present which are not present in the original graph. Multiple edges in the input graph are ignored.

This function returns each neighbor list in sorted order.

Arguments:

graph: The input graph.

al: Pointer to a not yet initialized adjacency list.

mode: Constant specifying whether outgoing (`IGRAPH_OUT`), incoming (`IGRAPH_IN`), or both (`IGRAPH_ALL`) types of neighbors (in the complementer graph) to include in the adjacency list. It is ignored for undirected networks.

loops: Specifies how to treat loop edges. `IGRAPH_NO_LOOPS` will not include loops edges in the returned adjacency list. `IGRAPH_LOOPS_ONCE` will include vertex *i* in the adjacency

list of vetex *i* *once* if the original graph did not have a loop edge incident on vertex *i*, while IGRAPH_LOOPS_TWICE will include vertex *i* *twice if* *mode* is set to IGRAPH_ALL (otherwise it is treated the same way as IGRAPH_LOOPS_ONCE).

Returns:

Error code.

See also:

igraph_adjlist_init(), igraph_complementer()

Time complexity: O(|V|^2+|E|), quadratic in the number of vertices.

igraph_adjlist_init_from_inclist — Constructs an adjacency list of vertices from an incidence list.

```
igraph_error_t igraph_adjlist_init_from_inclist(
    const igraph_t *graph, igraph_adjlist_t *al, const igraph_inclist_t *il);
```

In some algorithms it is useful to have an adjacency list *and* an incidence list representation of the same graph, and in many cases it is the most useful if they are consistent with each other, i.e. if can be guaranteed that the vertex ID in the i-th entry of the adjacency list of vertex v is the *other* endpoint of the edge in the i-th entry of the incidence list of the same vertex. This function creates such an adjacency list from the corresponding incidence list by looking up the endpoints of each edge in the incidence list and constructing the corresponding adjacenecy vectors.

The adjacency list is independent of the graph or the incidence list after creation; in other words, modifications that are made to the graph or the incidence list are not reflected in the adjacency list.

Arguments:

graph: The input graph.

al: Pointer to an uninitialized igraph_adjlist_t object.

il: Pointer to an *initialized* igraph_inclist_t object that will be converted into an adjacency list.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

igraph_adjlist_destroy — Deallocates an adjacency list.

```
void igraph_adjlist_destroy(igraph_adjlist_t *al);
```

Free all memory allocated for an adjacency list.

Arguments:

al: The adjacency list to destroy.

Time complexity: O(n), where n is the size of the adjacency list.

igraph_adjlist_get — Query a vector in an adjacency list.

```
#define igraph_adjlist_get(al,no)
```

Returns a pointer to an `igraph_vector_int_t` object from an adjacency list. The vector can be modified as desired.

Arguments:

al: The adjacency list object.

no: The vertex whose adjacent vertices will be returned.

Returns:

Pointer to the `igraph_vector_int_t` object.

Time complexity: O(1).

`igraph_adjlist_size` — Returns the number of vertices in an adjacency list.

```
igraph_int_t igraph_adjlist_size(const igraph_adjlist_t *al);
```

Arguments:

al: The adjacency list.

Returns:

The number of vertices in the adjacency list.

Time complexity: O(1).

`igraph_adjlist_clear` — Removes all edges from an adjacency list.

```
void igraph_adjlist_clear(igraph_adjlist_t *al);
```

The size of the adjacency list stays unchanged, but all adjacent vertices will be removed.

Arguments:

al: The adjacency list.

Time complexity: O(n), where n is the size of the adjacency list.

`igraph_adjlist_sort` — Sorts each vector in an adjacency list.

```
void igraph_adjlist_sort(igraph_adjlist_t *al);
```

Sorts every vector of the adjacency list. Note that `igraph_adjlist_init()` already produces sorted neighbor lists. This function is useful when the adjacency list is produced in a different manner, or is modified in a way that does not preserve the sorted order.

Arguments:

al: The adjacency list.

Time complexity: O(m log m), m is the total number of neighbors stored in the adjacency list.

igraph_adjlist_simplify — Simplifies an adjacency list.

```
igraph_error_t igraph_adjlist_simplify(igraph_adjlist_t *al);
```

Simplifies an adjacency list, i.e. removes loop and multiple edges.

When the adjacency list is created with `igraph_adjlist_init()`, use the `loops` and `multiple` parameters of that function instead.

Arguments:

al: The adjacency list.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of edges and vertices.

Incident edges

igraph_inclist_init — Initializes an incidence list.

```
igraph_error_t igraph_inclist_init(const igraph_t *graph,
                        igraph_inclist_t *il,
                        igraph_neimode_t mode,
                        igraph_loops_t loops);
```

Creates a list of vectors containing the incident edges for all vertices. The incidence list is independent of the graph after creation, subsequent changes of the graph object do not update the incidence list, and changes to the incidence list do not update the graph.

When *mode* is `IGRAPH_IN` or `IGRAPH_OUT`, each edge ID will appear in the incidence list *once.* When *mode* is `IGRAPH_ALL`, each edge ID will appear in the incidence list *twice*, once for the source vertex and once for the target edge. It also means that the edge IDs of loop edges may potentially appear *twice* for the *same* vertex. Use the *loops* argument to control whether this will be the case (`IGRAPH_LOOPS_TWICE`) or not (`IGRAPH_LOOPS_ONCE` or `IGRAPH_NO_LOOPS`).

As of igraph 0.10, there is a small performance cost to setting *loops* to a different value than `IGRAPH_LOOPS_TWICE`.

Arguments:

graph: The input graph.

il: Pointer to an uninitialized incidence list.

mode: Constant specifying whether incoming edges (`IGRAPH_IN`), outgoing edges (`IGRAPH_OUT`) or both (`IGRAPH_ALL`) to include in the incidence lists of directed graphs. It is ignored for undirected graphs.

loops: Specifies how to treat loop edges. `IGRAPH_NO_LOOPS` removes loop edges from the incidence list. `IGRAPH_LOOPS_ONCE` makes each loop edge appear only once in the

incidence list of the corresponding vertex. `IGRAPH_LOOPS_TWICE` makes loop edges appear *twice* in the incidence list of the corresponding vertex, but only if the graph is undirected or `mode` is set to `IGRAPH_ALL`.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

`igraph_inclist_destroy` — Frees all memory allocated for an incidence list.

```
void igraph_inclist_destroy(igraph_inclist_t *il);
```

Arguments:

`il`: The incidence list to destroy.

Time complexity: O(n), where n is the size of the incidence list.

`igraph_inclist_get` — Query a vector in an incidence list.

```
#define igraph_inclist_get(il,no)
```

Returns a pointer to an igraph_vector_int_t object from an incidence list containing edge IDs. The vector can be modified, resized, etc. as desired.

Arguments:

`il`: Pointer to the incidence list.

`no`: The vertex for which the incident edges are returned.

Returns:

Pointer to an igraph_vector_int_t object.

Time complexity: O(1).

`igraph_inclist_size` — Returns the number of vertices in an incidence list.

```
igraph_int_t igraph_inclist_size(const igraph_inclist_t *il);
```

Arguments:

`il`: The incidence list.

Returns:

The number of vertices in the incidence list.

Time complexity: O(1).

`igraph_inclist_clear` — Removes all edges from an incidence list.

```
void igraph_inclist_clear(igraph_inclist_t *il);
```

The size of the incidence list stays unchanged, but all incident edges will be removed.

Arguments:

`il`: The incidence list.

Time complexity: O(n), where n is the size of the incidence list.

Lazy adjacency list for vertices

`igraph_lazy_adjlist_init` — Initializes a lazy adjacency list.

```
igraph_error_t igraph_lazy_adjlist_init(const igraph_t *graph,
                            igraph_lazy_adjlist_t *al,
                            igraph_neimode_t mode,
                            igraph_loops_t loops,
                            igraph_bool_t multiple);
```

Create a lazy adjacency list for vertices. This function only allocates some memory for storing the vectors of an adjacency list, but the neighbor vertices are not queried, only at the `igraph_lazy_adjlist_get()` calls. Neighbor lists will be returned in sorted order.

As of igraph 0.10, there is a small performance cost to setting *`loops`* to a different value than `IGRAPH_LOOPS_TWICE` or setting *`multiple`* to a different value from `IGRAPH_MULTIPLE`.

Arguments:

`graph`: The input graph.

`al`: Pointer to an uninitialized adjacency list object.

`mode`: Constant specifying whether to include only outgoing (`IGRAPH_OUT`), only incoming (`IGRAPH_IN`), or both (`IGRAPH_ALL`) types of neighbors in the adjacency list. It is ignored for undirected graphs.

`loops`: Specifies how to treat loop edges. `IGRAPH_NO_LOOPS` removes loop edges from the adjacency list. `IGRAPH_LOOPS_ONCE` makes each loop edge appear only once in the adjacency list of the corresponding vertex. `IGRAPH_LOOPS_TWICE` makes loop edges appear *twice* in the adjacency list of the corresponding vertex, but only if the graph is undirected or *`mode`* is set to `IGRAPH_ALL`.

`multiple`: Specifies how to treat multiple (parallel) edges. `IGRAPH_NO_MULTIPLE` collapses parallel edges into a single one; `IGRAPH_MULTIPLE` keeps the multiplicities of parallel edges so the same vertex will appear as many times in the adjacency list of another vertex as the number of parallel edges going between the two vertices.

Returns:

Error code.

See also:

`igraph_neighbors()` for getting the neighbor lists of individual vertices.

Time complexity: O(|V|), the number of vertices, possibly, but depends on the underlying memory management too.

`igraph_lazy_adjlist_destroy` — Deallocate a lazt adjacency list.

```
void igraph_lazy_adjlist_destroy(igraph_lazy_adjlist_t *al);
```

Free all allocated memory for a lazy adjacency list.

Arguments:

`al`: The adjacency list to deallocate.

Time complexity: depends on the memory management.

`igraph_lazy_adjlist_get` — Query neighbor vertices.

```
#define igraph_lazy_adjlist_get(al,no)
```

If the function is called for the first time for a vertex then the result is stored in the adjacency list and no further query operations are needed when the neighbors of the same vertex are queried again.

Arguments:

`al`: The lazy adjacency list.

`no`: The vertex ID to query.

Returns:

Pointer to a vector, or `NULL` upon error. Errors can only occur the first time this function is called for a given vertex. It is safe to modify this vector, modification does not affect the original graph.

See also:

`igraph_lazy_adjlist_has()` to check if this function has already been called for a vertex.

Time complexity: O(d), the number of neighbor vertices for the first time, O(1) for subsequent calls.

`igraph_lazy_adjlist_has` — Are adjacenct vertices already stored in a lazy adjacency list?

```
#define igraph_lazy_adjlist_has(al,no)
```

Arguments:

`al`: The lazy adjacency list.

`no`: The vertex ID to query.

Returns:

True if the adjacent vertices of this vertex are already computed and stored, false otherwise.

Time complexity: O(1).

`igraph_lazy_adjlist_size` — Returns the number of vertices in a lazy adjacency list.

```
igraph_int_t igraph_lazy_adjlist_size(const igraph_lazy_adjlist_t *al);
```

Arguments:

al: The lazy adjacency list.

Returns:

The number of vertices in the lazy adjacency list.

Time complexity: O(1).

`igraph_lazy_adjlist_clear` — Removes all edges from a lazy adjacency list.

```
void igraph_lazy_adjlist_clear(igraph_lazy_adjlist_t *al);
```

Arguments:

al: The lazy adjacency list. Time complexity: depends on memory management, typically O(n), where n is the total number of elements in the adjacency list.

Lazy incidence list for edges

`igraph_lazy_inclist_init` — Initializes a lazy incidence list of edges.

```
igraph_error_t igraph_lazy_inclist_init(const igraph_t *graph,
                                        igraph_lazy_inclist_t *il,
                                        igraph_neimode_t mode,
                                        igraph_loops_t loops);
```

Create a lazy incidence list for edges. This function only allocates some memory for storing the vectors of an incidence list, but the incident edges are not queried, only when `igraph_lazy_inclist_get()` is called.

When *mode* is `IGRAPH_IN` or `IGRAPH_OUT`, each edge ID will appear in the incidence list *once*. When *mode* is `IGRAPH_ALL`, each edge ID will appear in the incidence list *twice*, once for the source vertex and once for the target edge. It also means that the edge IDs of loop edges will appear *twice* for the *same* vertex.

As of igraph 0.10, there is a small performance cost to setting *loops* to a different value than `IGRAPH_LOOPS_TWICE`.

Arguments:

graph: The input graph.

il: Pointer to an uninitialized incidence list.

mode: Constant, it gives whether incoming edges (IGRAPH_IN), outgoing edges (IGRAPH_OUT) or both types of edges (IGRAPH_ALL) are considered. It is ignored for undirected graphs.

loops: Specifies how to treat loop edges. IGRAPH_NO_LOOPS removes loop edges from the incidence list. IGRAPH_LOOPS_ONCE makes each loop edge appear only once in the incidence list of the corresponding vertex. IGRAPH_LOOPS_TWICE makes loop edges appear *twice* in the incidence list of the corresponding vertex, but only if the graph is undirected or mode is set to IGRAPH_ALL.

Returns:

Error code.

Time complexity: O(|V|), the number of vertices, possibly. But it also depends on the underlying memory management.

igraph_lazy_inclist_destroy — Deallocates a lazy incidence list.

```
void igraph_lazy_inclist_destroy(igraph_lazy_inclist_t *il);
```

Frees all allocated memory for a lazy incidence list.

Arguments:

il: The incidence list to deallocate.

Time complexity: depends on memory management.

igraph_lazy_inclist_get — Query incident edges.

```
#define igraph_lazy_inclist_get(il,no)
```

If the function is called for the first time for a vertex, then the result is stored in the incidence list and no further query operations are needed when the incident edges of the same vertex are queried again.

Arguments:

il: The lazy incidence list object.

no: The vertex ID to query.

Returns:

Pointer to a vector, or NULL upon error. Errors can only occur the first time this function is called for a given vertex. It is safe to modify this vector, modification does not affect the original graph.

See also:

igraph_lazy_inclist_has() to check if this function has already been called for a vertex.

Time complexity: O(d), the number of incident edges for the first time, O(1) for subsequent calls with the same *no* argument.

igraph_lazy_inclist_has — Are incident edges already stored in a lazy inclist?

```
#define igraph_lazy_inclist_has(il,no)
```

Arguments:

il: The lazy incidence list.

no: The vertex ID to query.

Returns:

True if the incident edges of this vertex are already computed and stored, false otherwise.

Time complexity: O(1).

igraph_lazy_inclist_size — Returns the number of vertices in a lazy incidence list.

```
igraph_int_t igraph_lazy_inclist_size(const igraph_lazy_inclist_t *il);
```

Arguments:

il: The lazy incidence list.

Returns:

The number of vertices in the lazy incidence list.

Time complexity: O(1).

igraph_lazy_inclist_clear — Removes all edges from a lazy incidence list.

```
void igraph_lazy_inclist_clear(igraph_lazy_inclist_t *il);
```

Arguments:

il: The lazy incidence list.

Time complexity: depends on memory management, typically O(n), where n is the total number of elements in the incidence list.

Partial prefix sum trees

The igraph_psumtree_t data type represents a partial prefix sum tree. A partial prefix sum tree is a data structure that can be used to draw samples from a discrete probability distribution with dynamic

probabilities that are updated frequently. This is achieved by creating a binary tree where the leaves are the items. Each leaf contains the probability corresponding to the items. Intermediate nodes of the tree always contain the sum of its two children. When the value of a leaf node is updated, the values of its ancestors are also updated accordingly.

Samples can be drawn from the probability distribution represented by the tree by generating a uniform random number between 0 (inclusive) and the value of the root of the tree (exclusive), and then following the branches of the tree as follows. In each step, the value in the current node is compared with the generated number. If the value in the node is larger, the left branch of the tree is taken; otherwise the generated number is decreased by the value in the node and the right branch of the tree is taken, until a leaf node is reached.

Note that the sampling process works only if all the values in the tree are non-negative. This is enforced by the object; in particular, trying to set a negative value for an item will produce an igraph error.

`igraph_psumtree_init` — Initializes a partial prefix sum tree.

```
igraph_error_t igraph_psumtree_init(igraph_psumtree_t *t, igraph_int_t size);
```

The tree is initialized with a fixed number of elements. After initialization, the value corresponding to each element is zero.

Arguments:

t: The tree to initialize.

size: The number of elements in the tree. It must be at least one.

Returns:

Error code, typically `IGRAPH_ENOMEM` if there is not enough memory.

Time complexity: O(n) for a tree containing n elements

`igraph_psumtree_destroy` — Destroys a partial prefix sum tree.

```
void igraph_psumtree_destroy(igraph_psumtree_t *t);
```

All partial prefix sum trees initialized by `igraph_psumtree_init()` should be properly destroyed by this function. A destroyed tree needs to be reinitialized by `igraph_psumtree_init()` if you want to use it again.

Arguments:

t: Pointer to the (previously initialized) tree to destroy.

Time complexity: operating system dependent.

`igraph_psumtree_size` — Returns the size of the tree.

```
igraph_int_t igraph_psumtree_size(const igraph_psumtree_t *t);
```

Arguments:

t: The tree object

Returns:

The number of discrete items in the tree.

Time complexity: O(1).

`igraph_psumtree_get` — Retrieves the value corresponding to an item in the tree.

```
igraph_real_t igraph_psumtree_get(const igraph_psumtree_t *t, igraph_int_t idx)
```

Arguments:

t: The tree to query.

idx: The index of the item whose value is to be retrieved.

Returns:

The value corresponding to the item with the given index.

Time complexity: O(1)

`igraph_psumtree_sum` — Returns the sum of the values of the leaves in the tree.

```
igraph_real_t igraph_psumtree_sum(const igraph_psumtree_t *t);
```

Arguments:

t: The tree object

Returns:

The sum of the values of the leaves in the tree.

Time complexity: O(1).

`igraph_psumtree_search` — Finds an item in the tree, given a value.

```
igraph_error_t igraph_psumtree_search(const igraph_psumtree_t *t, igraph_int_t
                           igraph_real_t search);
```

This function finds the item with the lowest index where it holds that the sum of all the items with a *lower* index is less than or equal to the given value and that the sum of all the items with a lower index plus the item itself is larger than the given value.

If you think about the partial prefix sum tree as a tool to sample from a discrete probability distribution, then calling this function repeatedly with uniformly distributed random numbers in the range 0 (inclusive) to the sum of all values in the tree (exclusive) will sample the items in the tree with a probability that is proportional to their associated values.

Arguments:

t: The tree to query.

idx: The index of the item is returned here.

search: The value to use for the search. Must be in the interval `[0, sum)`, where `sum` is the sum of all elements (leaves) in the tree.

Returns:

Error code; currently the search always succeeds.

Time complexity: O(log n), where n is the number of items in the tree.

`igraph_psumtree_update` — Updates the value associated to an item in the tree.

```
igraph_error_t igraph_psumtree_update(igraph_psumtree_t *t, igraph_int_t idx,
                           igraph_real_t new_value);
```

Arguments:

t: The tree to query.

idx: The index of the item to update.

new_value: The new value of the item.

Returns:

Error code, `IGRAPH_EINVAL` if the new value is negative or NaN, `IGRAPH_SUCCESS` if the operation was successful.

Time complexity: O(log n), where n is the number of items in the tree.

`igraph_psumtree_reset` — Resets all the values in the tree to zero.

```
void igraph_psumtree_reset(igraph_psumtree_t *t);
```

Arguments:

t: The tree to reset.

Bitsets

About igraph_bitset_t objects

The igraph_bitset_t data type is a simple and efficient interface to arrays containing boolean values. It is similar to the bitset template in the C++ standard library, although the main difference being the C++ version's size is initialized at compile time.

The igraph_bitset_t type and use O(n/w) space to store n elements, where w is the bit width of igraph_int_t, the integer type used throughout the library (either 32 or 64). Sometimes they use more, this is because bitsets can shrink, but even if they shrink, the current implementation does not free a single bit of memory.

The elements in an igraph_bitset_t object and its variants are indexed from zero, we follow the usual C convention here. Bitsets are indexed from right to left, meaning index 0 is the least significant bit and index `n - 1` is the most significant bit.

The elements of a bitset always occupy a single block of memory, the starting address of this memory block can be queried with the `VECTOR()` macro. This way, bitset objects can be used with standard mathematical libraries, like the GNU Scientific Library.

Note that while the interface of bitset functions is similar to igraph's vector functions, there is one major difference: bitset functions such as `igraph_bitset_and()` do not verify that that sizes of input parameters are compatible, and do not automatically resize the output parameter. Doing so is the responsibility of the user.

Constructors and destructors

igraph_bitset_t objects have to be initialized before using them, this is analogous to calling a constructor on them. There are two igraph_bitset_t constructors, for your convenience. `igraph_bitset_init()` is the basic constructor, it creates a bitset of the given length, filled with zeros. `igraph_bitset_init_copy()` creates a new identical copy of an already existing and initialized bitset.

If a igraph_bitset_t object is not needed any more, it should be destroyed to free its allocated memory by calling the igraph_bitset_t destructor, `igraph_bitset_destroy()`.

`igraph_bitset_init` — Initializes a bitset object (constructor).

```
igraph_error_t igraph_bitset_init(igraph_bitset_t *bitset, igraph_int_t size);
```

Every bitset needs to be initialized before it can be used, and there are a number of initialization functions or otherwise called constructors. This function constructs a bitset of the given size and initializes each entry to 0.

Every bitset object initialized by this function should be destroyed (ie. the memory allocated for it should be freed) when it is not needed anymore, the `igraph_bitset_destroy()` function is responsible for this.

Arguments:

bitset: Pointer to a not yet initialized bitset object.

size: The size of the bitset.

Returns:

error code: `IGRAPH_ENOMEM` if there is not enough memory.

Time complexity: operating system dependent, the amount of "time" required to allocate O(n/w) elements, n is the number of elements. w is the word size of the machine (32 or 64).

`igraph_bitset_init_copy` — Initializes a bitset from another bitset object (constructor).

```
igraph_error_t igraph_bitset_init_copy(igraph_bitset_t *dest, const igraph_bits
```

The contents of the existing bitset object will be copied to the new one.

Arguments:

dest: Pointer to a not yet initialized bitset object.

src: The original bitset object to copy.

Returns:

Error code: `IGRAPH_ENOMEM` if there is not enough memory.

Time complexity: operating system dependent, usually O(n/w), n is the size of the bitset, w is the word size of the machine (32 or 64).

`igraph_bitset_destroy` — Destroys a bitset object.

```
void igraph_bitset_destroy(igraph_bitset_t *bitset);
```

All bitsets initialized by `igraph_bitset_init()` should be properly destroyed by this function. A destroyed bitset needs to be reinitialized by `igraph_bitset_init()` or another constructor.

Arguments:

bitset: Pointer to the (previously initialized) bitset object to destroy.

Time complexity: operating system dependent.

Accessing elements

The simplest way to access an element of a bitset is to use the `IGRAPH_BIT_TEST()`, `IGRAPH_BIT_SET()` and `IGRAPH_BIT_CLEAR()` macros.

There are a few other macros which allow manual manipulation of bitsets. Those are `VECTOR()`, `IGRAPH_BIT_SLOT()`, `IGRAPH_BIT_MASK()` and `IGRAPH_BIT_NSLOTS()`.

`IGRAPH_BIT_MASK` — Computes mask used to access a specific bit of an integer.

```
#define IGRAPH_BIT_MASK(i)
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

Used in combination with `IGRAPH_BIT_SLOT()` to access an element of a bitset. Usage:

```
IGRAPH_BIT_MASK(10)
```

to obtain an integer where only the 11th least significant bit is set. Note that passing negative values here results in undefined behaviour.

Arguments:

b: The only bit index that should have its bit set.

Time complexity: O(1).

`IGRAPH_BIT_SLOT` — Computes index used to access a specific slot of a bitset.

```
#define IGRAPH_BIT_SLOT(i)
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

Used in combination with `IGRAPH_BIT_MASK` to access an element of a bitset. Usage:

```
IGRAPH_BIT_SLOT(70)
```

will return 1 if using 64-bit words or 2 if using 32-bit words.

Arguments:

i: The bit index whose slot should be determined.

Time complexity: O(1).

`IGRAPH_BIT_SET` — Sets a specific bit in a bitset to 1 without altering other bits.

```
#define IGRAPH_BIT_SET(bitset, i)
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

Usage:

```
IGRAPH_BIT_SET(bitset, 3)
```

will set the fourth least significant bit in the bitset to 1.

Arguments:

bitset: The bitset

i: The bit index that should have its bit set to 1 after the operation.

Time complexity: O(1).

IGRAPH_BIT_CLEAR — Sets a specific bit in a bitset to 0 without altering other bits.

```
#define IGRAPH_BIT_CLEAR(bitset, i)
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

Usage:

```
IGRAPH_BIT_CLEAR(bitset, 4)
```

will set the fifth least significant bit in the bitset to 0.

Arguments:

bitset: The bitset

i: The bit index that should have its bit set to 0 after the operation.

Time complexity: O(1).

IGRAPH_BIT_TEST — Tests whether a bit is set in a bitset.

```
#define IGRAPH_BIT_TEST(bitset, i)
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

Returns 0 if the bit at the specified bit index is not set, otherwise returns a non-zero value. Usage:

```
IGRAPH_BIT_TEST(bitset, 7)
```

will test the eighth least significant bit in the bitset.

Arguments:

bitset: The bitset

i: The bit index that should have its bit tested.

Time complexity: O(1).

IGRAPH_BIT_NSLOTS — Computes the number of slots required to store a specified number of bits.

```
#define IGRAPH_BIT_NSLOTS(nbits)
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

Usage:

```
IGRAPH_BIT_NSLOTS(70)
```

will return 2 if using 64-bit words and 3 if using 32-bit words.

```
IGRAPH_BIT_NSLOTS(128)
```

will return 2 if using 64-bit words and 4 if using 32-bit words.

Arguments:

nbits: The specified number of bits.

Time complexity: O(1).

Bitset operations

igraph_bitset_fill — Fills a bitset with a constant value.

```
void igraph_bitset_fill(igraph_bitset_t *bitset, igraph_bool_t value);
```

Sets all bits of a bitset to the same value.

Arguments:

bitset: The bitset object to modify.

value: The value to set for all bits.

See also:

```
igraph_bitset_null()
```

Time complexity: O(n/w).

igraph_bitset_null — Clears all bits in a bitset.

```
void igraph_bitset_null(igraph_bitset_t *bitset);
```

Arguments:

bitset: The bitset object to clear all bits in.

See also:

`igraph_bitset_fill()`

Time complexity: O(n/w).

igraph_bitset_or — Bitwise OR of two bitsets.

```
void igraph_bitset_or(igraph_bitset_t *dest,
                      const igraph_bitset_t *src1, const igraph_bitset_t *src2)
```

Applies a bitwise or to the contents of two bitsets and stores it in an already initialized bitset. The destination bitset may be equal to one (or even both) of the sources. When working with bitsets, it is common that those created are of the same size fixed size. Therefore, this function does not check the sizes of the bitsets passed to it, the caller must do so if necessary.

Arguments:

dest: The bitset object where the result is stored

src1: A bitset. Must have have same size as *dest*.

src2: A bitset. Must have have same size as *dest*.

Time complexity: O(n/w).

igraph_bitset_and — Bitwise AND of two bitsets.

```
void igraph_bitset_and(igraph_bitset_t *dest, const igraph_bitset_t *src1, cons
```

Applies a bitwise and to the contents of two bitsets and stores it in an already initialized bitset. The destination bitset may be equal to one (or even both) of the sources. When working with bitsets, it is common that those created are of the same size fixed size. Therefore, this function does not check the sizes of the bitsets passed to it, the caller must do so if necessary.

Arguments:

dest: The bitset object where the result is stored

src1: A bitset. Must have have same size as *dest*.

src2: A bitset. Must have have same size as *dest*.

Time complexity: O(n/w).

igraph_bitset_xor — Bitwise XOR of two bitsets.

```
void igraph_bitset_xor(igraph_bitset_t *dest,
                       const igraph_bitset_t *src1, const igraph_bitset_t *src2
```

Applies a bitwise xor to the contents of two bitsets and stores it in an already initialized bitset. The destination bitset may be equal to one (or even both) of the sources. When working with bitsets, it is

common that those created are of the same size fixed size. Therefore, this function does not check the sizes of the bitsets passed to it, the caller must do so if necessary.

Arguments:

dest: The bitset object where the result is stored

src1: A bitset. Must have have same size as *dest*.

src2: A bitset. Must have have same size as *dest*.

Time complexity: O(n/w).

igraph_bitset_not — Bitwise negation of a bitset.

```
void igraph_bitset_not(igraph_bitset_t *dest, const igraph_bitset_t *src);
```

Applies a bitwise not to the contents of a bitset and stores it in an already initialized bitset. The destination bitset may be equal to the source. When working with bitsets, it is common that those created are of the same size fixed size. Therefore, this function does not check the sizes of the bitsets passed to it, the caller must do so if necessary.

Arguments:

dest: The bitset object where the result is stored

src: A bitset. Must have have same size as *dest*.

Time complexity: O(n/w).

igraph_bitset_popcount — The population count of the bitset.

```
igraph_int_t igraph_bitset_popcount(const igraph_bitset_t *bitset);
```

Returns the number of set bits, also called the population count, of the bitset.

Arguments:

bitset: The bitset object

Returns:

The population count of the bitset.

Time complexity: O(n/w).

igraph_bitset_countl_zero — The number of leading zeros in the bitset.

```
igraph_int_t igraph_bitset_countl_zero(const igraph_bitset_t *bitset);
```

Returns the number of leading (starting at the most significant bit) zeros in the bitset before the first one is encountered. If the bitset is all zeros, then its size is returned.

Arguments:

bitset: The bitset object

Returns:

The number of leading zeros in the bitset.

Time complexity: O(n/w).

`igraph_bitset_countl_one` — The number of leading ones in the bitset.

```
igraph_int_t igraph_bitset_countl_one(const igraph_bitset_t *bitset);
```

Returns the number of leading ones (starting at the most significant bit) in the bitset before the first zero is encountered. If the bitset is all ones, then its size is returned.

Arguments:

`bitset`: The bitset object

Returns:

The number of leading ones in the bitset.

Time complexity: O(n/w).

`igraph_bitset_countr_zero` — The number of trailing zeros in the bitset.

```
igraph_int_t igraph_bitset_countr_zero(const igraph_bitset_t *bitset);
```

Returns the number of trailing (starting at the least significant bit) zeros in the bitset before the first one is encountered. If the bitset is all zeros, then its size is returned.

Arguments:

`bitset`: The bitset object

Returns:

The number of trailing zeros in the bitset.

Time complexity: O(n/w).

`igraph_bitset_countr_one` — The number of trailing ones in the bitset.

```
igraph_int_t igraph_bitset_countr_one(const igraph_bitset_t *bitset);
```

Returns the number of trailing ones (starting at the least significant bit) in the bitset before the first zero is encountered. If the bitset is all ones, then its size is returned.

Arguments:

`bitset`: The bitset object

Returns:

The number of trailing ones in the bitset.

Time complexity: O(n/w).

igraph_bitset_is_all_zero — Are all bits zeros?

```
igraph_bool_t igraph_bitset_is_all_zero(const igraph_bitset_t *bitset);
```

Arguments:

bitset: The bitset object to test.

Returns:

True if none of the bits are set.

Time complexity: O(n/w).

igraph_bitset_is_all_one — Are all bits ones?

```
igraph_bool_t igraph_bitset_is_all_one(const igraph_bitset_t *bitset);
```

Arguments:

bitset: The bitset object to test.

Returns:

True if all of the bits are set.

Time complexity: O(n/w).

igraph_bitset_is_any_zero — Are any bits zeros?

```
igraph_bool_t igraph_bitset_is_any_zero(const igraph_bitset_t *bitset);
```

Arguments:

bitset: The bitset object to test.

Returns:

True if at least one bit is zero.

Time complexity: O(n/w).

igraph_bitset_is_any_one — Are any bits ones?

```
igraph_bool_t igraph_bitset_is_any_one(const igraph_bitset_t *bitset);
```

Arguments:

bitset: The bitset object to test.

Returns:

True if at least one bit is one.

Time complexity: O(n/w).

Bitset properties

igraph_bitset_size — Returns the length of the bitset.

```
igraph_int_t igraph_bitset_size(const igraph_bitset_t *bitset);
```

Arguments:

bitset: The bitset object

Returns:

The size of the bitset.

Time complexity: O(1).

igraph_bitset_capacity — Returns the allocated capacity of the bitset.

```
igraph_int_t igraph_bitset_capacity(const igraph_bitset_t *bitset);
```

Note that this might be different from the size of the bitset (as queried by `igraph_bitset_size()`), and specifies how many elements the bitset can hold, without reallocation.

Arguments:

bitset: Pointer to the (previously initialized) bitset object to query.

Returns:

The allocated capacity.

See also:

`igraph_bitset_size()`.

Time complexity: O(1).

Resizing operations

igraph_bitset_reserve — Reserves memory for a bitset.

```
igraph_error_t igraph_bitset_reserve(igraph_bitset_t *bitset, igraph_int_t capa
```

igraph bitsets are flexible, they can grow and shrink. Growing however occasionally needs the data in the bitset to be copied. In order to avoid this, you can call this function to reserve space for future growth of the bitset.

Note that this function does *not* change the size of the bitset. Let us see a small example to clarify things: if you reserve space for 100 elements and the size of your bitset was (and still is) 60, then you can surely add additional 40 elements to your bitset before it will be copied.

Arguments:

bitset: The bitset object.

capacity: The new *allocated* size of the bitset.

Returns:

Error code: IGRAPH_ENOMEM if there is not enough memory.

Time complexity: operating system dependent, should be around O(n/w), n is the new allocated size of the bitset, w is the word size of the machine (32 or 64).

igraph_bitset_resize — Resizes the bitset.

```
igraph_error_t igraph_bitset_resize(igraph_bitset_t *bitset, igraph_int_t new_s
```

Note that this function does not free any memory, just sets the size of the bitset to the given one. It may, on the other hand, allocate more memory if the new size is larger than the previous one. In this case the newly appeared elements in the bitset are set to zero.

Arguments:

bitset: The bitset object

new_size: The new size of the bitset.

Returns:

Error code, IGRAPH_ENOMEM if there is not enough memory. Note that this function *never* returns an error if the bitset is made smaller.

See also:

igraph_bitset_reserve() for allocating memory for future extensions of a bitset.

Time complexity: O(1) if the new size is smaller, operating system dependent if it is larger. In the latter case it is usually around O(n/w), n is the new size of the bitset, w is the word size of the machine (32 or 64).

Copying bitsets

igraph_bitset_update — Update a bitset from another one.

```
igraph_error_t igraph_bitset_update(igraph_bitset_t *dest, const igraph_bitset_
```

The size and contents of *dest* will be identical to that of *src*.

Arguments:

dest: Pointer to an initialized bitset object. This will be updated.

src: The bitset to update from.

Returns:

Error code: IGRAPH_ENOMEM if there is not enough memory.

Time complexity: operating system dependent, usually O(n/w), n is the size of the bitset, w is the word size of the machine (32 or 64).

Chapter 8. Random numbers

About random numbers in igraph

Some algorithms in igraph, such as sampling from random graph models, require random number generators (RNGs). igraph includes a flexible RNG framework that allows hooking up arbitrary random number generators, and comes with several ready-to-use generators. This framework is used in igraph's high-level interfaces to integrate with the host language's own RNG.

The default random number generator

`igraph_rng_default` — Query the default random number generator.

```
igraph_rng_t *igraph_rng_default(void);
```

Returns:

A pointer to the default random number generator.

See also:

`igraph_rng_set_default()`

`igraph_rng_set_default` — Set the default igraph random number generator.

```
igraph_rng_t *igraph_rng_set_default(igraph_rng_t *rng);
```

This function updates the default RNG used by igraph to be the one pointed to by *rng*, and returns a pointer to the previous default RNG. Future calls to `igraph_rng_default()` will return the same pointer as *rng*. The RNG pointed to by *rng* must not be destroyed for as long as it is used as the default.

Arguments:

rng: The random number generator to use as default from now on. Calling `igraph_rng_destroy()` on it, while it is still being used as the default will result in crashes and/or unpredictable results.

Returns:

Pointer the previous default RNG.

Time complexity: O(1).

Creating random number generators

`igraph_rng_init` — Initializes a random number generator.

```
igraph_error_t igraph_rng_init(igraph_rng_t *rng, const igraph_rng_type_t *type
```

This function allocates memory for a random number generator, with the given type, and sets its seed to the default.

Arguments:

rng: Pointer to an uninitialized RNG.

type: The type of the RNG, such as `igraph_rngtype_mt19937`, `igraph_rngtype_glibc2`, `igraph_rngtype_pcg32` or `igraph_rngtype_pcg64`.

Returns:

Error code.

`igraph_rng_destroy` — Deallocates memory associated with a random number generator.

```
void igraph_rng_destroy(igraph_rng_t *rng);
```

Arguments:

rng: The RNG to destroy. Do not destroy an RNG that is used as the default igraph RNG.

Time complexity: O(1).

`igraph_rng_seed` — Seeds a random number generator.

```
igraph_error_t igraph_rng_seed(igraph_rng_t *rng, igraph_uint_t seed);
```

Arguments:

rng: The RNG.

seed: The new seed.

Returns:

Error code.

Time complexity: usually O(1), but may depend on the type of the RNG.

`igraph_rng_bits` — The number of random bits that a random number generator can produces in a single round.

```
igraph_int_t igraph_rng_bits(const igraph_rng_t* rng);
```

Arguments:

rng: The RNG.

Returns:

The number of random bits that can be generated in a single round with the RNG.

Time complexity: O(1).

`igraph_rng_max` — The maximum possible integer for a random number generator.

```
igraph_uint_t igraph_rng_max(const igraph_rng_t *rng);
```

Note that this number is only for informational purposes; it returns the maximum possible integer that can be generated with the RNG with a single call to its internals. It is derived directly from the number of random *bits* that the RNG can generate in a single round. When this is smaller than what would be needed by other RNG functions like `igraph_rng_get_integer()`, igraph will call the RNG multiple times to generate more random bits.

Arguments:

rng: The RNG.

Returns:

The largest possible integer that can be generated in a single round with the RNG.

Time complexity: O(1).

`igraph_rng_name` — The type of a random number generator.

```
const char *igraph_rng_name(const igraph_rng_t *rng);
```

Arguments:

rng: The RNG.

Returns:

The name of the type of the generator. Do not deallocate or change the returned string.

Time complexity: O(1).

Generating random numbers

`igraph_rng_get_bool` — Generate a random boolean.

```
igraph_bool_t igraph_rng_get_bool(igraph_rng_t *rng);
```

Use this function only when a single random boolean, i.e. a single bit is needed at a time. It is not efficient for generating multiple bits.

Arguments:

rng: Pointer to the RNG to use for the generation. Use `igraph_rng_default()` here to use the default igraph RNG.

Returns:

The generated bit, as a truth value.

`igraph_rng_get_integer` — Generate an integer random number from an interval.

```
igraph_int_t igraph_rng_get_integer(
    igraph_rng_t *rng, igraph_int_t l, igraph_int_t h
);
```

Generate uniformly distributed integers from the interval `[l, h]`.

Arguments:

rng: Pointer to the RNG to use for the generation. Use `igraph_rng_default()` here to use the default igraph RNG.

l: Lower limit, inclusive, it can be negative as well.

h: Upper limit, inclusive, it can be negative as well, but it must be at least `l`.

Returns:

The generated random integer.

Time complexity: O(log2(h-l+1) / bits) where bits is the value of `igraph_rng_bits`(rng).

`igraph_rng_get_unif01` — Samples uniformly from the unit interval.

```
igraph_real_t igraph_rng_get_unif01(igraph_rng_t *rng);
```

Generates uniformly distributed real numbers from the `[0, 1)` half-open interval.

Arguments:

rng: Pointer to the RNG to use. Use `igraph_rng_default()` here to use the default igraph RNG.

Returns:

The generated uniformly distributed random number.

Time complexity: depends on the type of the RNG.

`igraph_rng_get_unif` — Samples real numbers from a given interval.

```
igraph_real_t igraph_rng_get_unif(igraph_rng_t *rng,
                                  igraph_real_t l, igraph_real_t h);
```

Generates uniformly distributed real numbers from the `[l, h)` half-open interval.

Arguments:

rng: Pointer to the RNG to use. Use `igraph_rng_default()` here to use the default igraph RNG.

l: The lower bound, it can be negative.

h: The upper bound, it can be negative, but it has to be larger than the lower bound.

Returns:

The generated uniformly distributed random number.

Time complexity: depends on the type of the RNG.

`igraph_rng_get_normal` — Samples from a normal distribution.

```
igraph_real_t igraph_rng_get_normal(igraph_rng_t *rng,
                                    igraph_real_t m, igraph_real_t s);
```

Generates random variates from a normal distribution with probability density

`exp( -(x - m)^2 / (2 s^2) )`.

Arguments:

rng: Pointer to the RNG to use. Use `igraph_rng_default()` here to use the default igraph RNG.

m: The mean.

s: The standard deviation.

Returns:

The generated normally distributed random number.

Time complexity: depends on the type of the RNG.

`igraph_rng_get_exp` — Samples from an exponential distribution.

```
igraph_real_t igraph_rng_get_exp(igraph_rng_t *rng, igraph_real_t rate);
```

Generates random variates from an exponential distribution with probability density proportional to

`exp(-rate x)`.

Arguments:

rng: Pointer to the RNG to use. Use `igraph_rng_default()` here to use the default igraph RNG.

rate: Rate parameter.

Returns:

The generated sample.

Time complexity: depends on the RNG.

`igraph_rng_get_gamma` — Samples from a gamma distribution.

```
igraph_real_t igraph_rng_get_gamma(igraph_rng_t *rng, igraph_real_t shape,
                                   igraph_real_t scale);
```

Generates random variates from a gamma distribution with probability density proportional to

`x^(shape-1) exp(-x / scale)`.

Arguments:

rng: Pointer to the RNG to use. Use `igraph_rng_default()` here to use the default igraph RNG.

shape: Shape parameter.

scale: Scale parameter.

Returns:

The generated sample.

Time complexity: depends on the RNG.

`igraph_rng_get_binom` — Samples from a binomial distribution.

```
igraph_real_t igraph_rng_get_binom(igraph_rng_t *rng, igraph_int_t n, igraph_rea
```

Generates random variates from a binomial distribution. The number `k` is generated with probability

`(n \choose k) p^k (1-p)^(n-k)`, `k = 0, 1, ..., n`.

Arguments:

rng: Pointer to the RNG to use. Use `igraph_rng_default()` here to use the default igraph RNG.

n: Number of observations.

p: Probability of an event.

Returns:

The generated binomially distributed random number.

Time complexity: depends on the RNG.

`igraph_rng_get_geom` — Samples from a geometric distribution.

```
igraph_real_t igraph_rng_get_geom(igraph_rng_t *rng, igraph_real_t p);
```

Generates random variates from a geometric distribution. The number `k` is generated with probability

`(1 - p)^k p, k = 0, 1, 2, ....`

Arguments:

`rng`: Pointer to the RNG to use. Use `igraph_rng_default()` here to use the default igraph RNG.

`p`: The probability of success in each trial. Must be larger than zero and smaller or equal to 1.

Returns:

The generated geometrically distributed random number.

Time complexity: depends on the RNG.

`igraph_rng_get_pois` — Samples from a Poisson distribution.

```
igraph_real_t igraph_rng_get_pois(igraph_rng_t *rng, igraph_real_t rate);
```

Generates random variates from a Poisson distribution. The number `k` is generated with probability

`rate^k * exp(-rate) / k!, k = 0, 1, 2, ....`

Arguments:

`rng`: Pointer to the RNG to use. Use `igraph_rng_default()` here to use the default igraph RNG.

`rate`: The rate parameter of the Poisson distribution. Must not be negative.

Returns:

The generated geometrically distributed random number.

Time complexity: depends on the RNG.

Supported random number generators

By default igraph uses the MT19937 generator. Prior to igraph version 0.6, the generator supplied by the standard C library was used. This means the GLIBC2 generator on GNU libc 2 systems, and

maybe the BSD RAND generator on others. The RAND generator was removed due to poor statistical properties in version 0.10. The PCG32 generator was added in version 0.10.

igraph_rngtype_mt19937 — The MT19937 random number generator.

```
const igraph_rng_type_t igraph_rngtype_mt19937 = {
    /* name= */      "MT19937",
    /* bits=  */     32,
    /* init= */      igraph_rng_mt19937_init,
    /* destroy= */   igraph_rng_mt19937_destroy,
    /* seed= */      igraph_rng_mt19937_seed,
    /* get= */       igraph_rng_mt19937_get,
    /* get_int= */   NULL,
    /* get_real= */  NULL,
    /* get_norm= */  NULL,
    /* get_geom= */  NULL,
    /* get_binom= */ NULL,
    /* get_exp= */   NULL,
    /* get_gamma= */ NULL,
    /* get_pois= */  NULL
};
```

The MT19937 generator of Makoto Matsumoto and Takuji Nishimura is a variant of the twisted generalized feedback shift-register algorithm, and is known as the “Mersenne Twister” generator. It has a Mersenne prime period of 2^19937 - 1 (about 10^6000) and is equi-distributed in 623 dimensions. It has passed the diehard statistical tests. It uses 624 words of state per generator and is comparable in speed to the other generators. The original generator used a default seed of 4357 and choosing `s` equal to zero in `igraph_rng_mt19937_seed`() reproduces this. Later versions switched to 5489 as the default seed, you can choose this explicitly via `igraph_rng_seed()` instead if you require it.

For more information see, Makoto Matsumoto and Takuji Nishimura, “Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator”. ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 1 (Jan. 1998), Pages 3–30

The generator `igraph_rngtype_mt19937` uses the second revision of the seeding procedure published by the two authors above in 2002. The original seeding procedures could cause spurious artifacts for some seed values.

This generator was ported from the GNU Scientific Library.

igraph_rngtype_glibc2 — The random number generator introduced in GNU libc 2.

```
const igraph_rng_type_t igraph_rngtype_glibc2 = {
    /* name= */      "LIBC",
    /* bits=  */     31,
    /* init= */      igraph_rng_glibc2_init,
    /* destroy= */   igraph_rng_glibc2_destroy,
    /* seed= */      igraph_rng_glibc2_seed,
    /* get= */       igraph_rng_glibc2_get,
    /* get_int= */   NULL,
    /* get_real= */  NULL,
    /* get_norm= */  NULL,
    /* get_geom= */  NULL,
```

```
    /* get_binom= */ NULL,
    /* get_exp= */   NULL,
    /* get_gamma= */ NULL,
    /* get_pois= */  NULL
};
```

This is a linear feedback shift register generator with a 128-byte buffer. This generator was the default prior to igraph version 0.6, at least on systems relying on GNU libc. This generator was ported from the GNU Scientific Library. It is a reimplementation and does not call the system glibc generator.

`igraph_rngtype_pcg32` — The PCG random number generator (32-bit version).

```
const igraph_rng_type_t igraph_rngtype_pcg32 = {
    /* name= */      "PCG32",
    /* bits=  */     32,
    /* init= */      igraph_rng_pcg32_init,
    /* destroy= */   igraph_rng_pcg32_destroy,
    /* seed= */      igraph_rng_pcg32_seed,
    /* get= */       igraph_rng_pcg32_get,
    /* get_int= */   NULL,
    /* get_real= */  NULL,
    /* get_norm= */  NULL,
    /* get_geom= */  NULL,
    /* get_binom= */ NULL,
    /* get_exp= */   NULL,
    /* get_gamma= */ NULL,
    /* get_pois= */  NULL
};
```

This is an implementation of the PCG random number generator; see https://www.pcg-random.org for more details. This implementation returns 32 random bits in a single iteration.

The generator was ported from the original source code published by the authors at https://github.com/imneme/pcg-c.

`igraph_rngtype_pcg64` — The PCG random number generator (64-bit version).

```
const igraph_rng_type_t igraph_rngtype_pcg64 = {
    /* name= */      "PCG64",
    /* bits=  */     64,
    /* init= */      igraph_rng_pcg64_init,
    /* destroy= */   igraph_rng_pcg64_destroy,
    /* seed= */      igraph_rng_pcg64_seed,
    /* get= */       igraph_rng_pcg64_get,
    /* get_int= */   NULL,
    /* get_real= */  NULL,
    /* get_norm= */  NULL,
    /* get_geom= */  NULL,
    /* get_binom= */ NULL,
    /* get_exp= */   NULL,
    /* get_gamma= */ NULL,
    /* get_pois= */  NULL
```

```
};
```

This is an implementation of the PCG random number generator; see https://www.pcg-random.org for more details. This implementation returns 64 random bits in a single iteration. It is only available on 64-bit plaforms with compilers that provide the __uint128_t type.

PCG64 typically provides better performance than PCG32 when sampling floating point numbers or very large integers, as it can provide twice as many random bits in a single generation round.

The generator was ported from the original source code published by the authors at https://github.com/imneme/pcg-c.

Use cases

Normal (default) use

If the user does not use any of the RNG functions explicitly, but calls some of the randomized igraph functions, then a default RNG is set up the first time an igraph function needs random numbers. The seed of this RNG is the output of the `time(0)` function call, using the `time` function from the standard C library. This ensures that igraph creates a different random graph, each time the C program is called.

The created default generator is stored internally and can be queried with the `igraph_rng_default()` function.

Reproducible simulations

If reproducible results are needed, then the user should set the seed of the default random number generator explicitly, using the `igraph_rng_seed()` function on the default generator, `igraph_rng_default()`. When setting the seed to the same number, igraph generates exactly the same random graph (or series of random graphs).

Changing the default generator

By default igraph uses the `igraph_rng_default()` random number generator. This can be changed any time by calling `igraph_rng_set_default()`, with an already initialized random number generator. Note that the old (replaced) generator is not destroyed, so no memory is deallocated.

Using multiple generators

igraph also provides functions to set up multiple random number generators, using the `igraph_rng_init()` function, and then generating random numbers from them, e.g. with `igraph_rng_get_integer()` and/or `igraph_rng_get_unif()` calls.

Note that initializing a new random number generator is independent of the generator that the igraph functions themselves use. If you want to replace that, then please use `igraph_rng_set_default()`.

Example

Example 8.1. File `examples/simple/random_seed.c`

Chapter 9. Vertex and edge selectors and sequences, iterators

About selectors, iterators

Everything about vertices and vertex selectors also applies to edges and edge selectors unless explicitly noted otherwise.

The vertex (and edge) selector notion was introduced in igraph 0.2. It is a way to reference a sequence of vertices or edges independently of the graph.

While this might sound quite mysterious, it is actually very simple. For example, all vertices of a graph can be selected by `igraph_vs_all()` and the graph independence means that `igraph_vs_all()` is not parametrized by a graph object. That is, `igraph_vs_all()` is the general *concept* of selecting all vertices of a graph. A vertex selector is then a way to specify the class of vertices to be visited. The selector might specify that all vertices of a graph or all the neighbours of a vertex are to be visited. A vertex selector is a way of saying that you want to visit a bunch of vertices, as opposed to a vertex iterator which is a concrete plan for visiting each of the chosen vertices of a specific graph.

To determine the actual vertex IDs implied by a vertex selector, you need to apply the concept of selecting vertices to a specific graph object. This can be accomplished by instantiating a vertex iterator using a specific vertex selection concept and a specific graph object. The notion of vertex iterators can be thought of in the following way. Given a specific graph object and the class of vertices to be visited, a vertex iterator is a road map, plan or route for how to visit the chosen vertices.

Some vertex selectors have *immediate* versions. These have the prefix `igraph_vss` instead of `igraph_vs`, e.g. `igraph_vss_all()` instead of `igraph_vs_all()`. The immediate versions are to be used in the parameter list of the igraph functions, such as `igraph_degree()`. These functions are not associated with any igraph_vs_t object, so they have no separate constructors and destructors (destroy functions).

Vertex selector constructors

Vertex selectors are created by vertex selector constructors, can be instantiated with `igraph_vit_create()`, and are destroyed with `igraph_vs_destroy()`.

`igraph_vs_all` — Vertex set, all vertices of a graph.

```
igraph_error_t igraph_vs_all(igraph_vs_t *vs);
```

Arguments:

vs: Pointer to an uninitialized igraph_vs_t object.

Returns:

Error code.

See also:

`igraph_vss_all()`, `igraph_vs_destroy()`

This selector includes all vertices of a given graph in increasing vertex ID order.

Time complexity: O(1).

igraph_vs_adj — Adjacent vertices of a vertex.

```
igraph_error_t igraph_vs_adj(
    igraph_vs_t *vs, igraph_int_t vid, igraph_neimode_t mode,
    igraph_loops_t loops, igraph_bool_t multiple
);
```

All neighboring vertices of a given vertex are selected by this selector. The `mode` argument controls the type of the neighboring vertices to be selected. The vertices are visited in increasing vertex ID order, as of igraph version 0.4.

Arguments:

vs: Pointer to an uninitialized vertex selector object.

vid: Vertex ID, the center of the neighborhood.

mode: Decides the type of the neighborhood for directed graphs. This parameter is ignored for undirected graphs. Possible values:

- `IGRAPH_OUT` All vertices to which there is a directed edge from `vid`. That is, all the out-neighbors of `vid`.
- `IGRAPH_IN` All vertices from which there is a directed edge to `vid`. In other words, all the in-neighbors of `vid`.
- `IGRAPH_ALL` All vertices to which or from which there is a directed edge from/to `vid`. That is, all the neighbors of `vid` considered as if the graph is undirected.

loops: Whether to include the vertex itself in the neighborhood if the vertex has a loop edge. If `IGRAPH_NO_LOOPS`, loop edges are excluded. If `IGRAPH_LOOPS_ONCE`, the vertex is included in its own neighborhood once for every loop edge that it has. If `IGRAPH_LOOPS_TWICE`, the vertex is included twice in its own neighborhood for every loop edge that it has, but only if the graph is undirected or *mode* is set to `IGRAPH_ALL`.

multiple: Whether to include multiple edges. If `IGRAPH_NO_MULTIPLE`, multiple edges are not included in the neighborhood. If `IGRAPH_MULTIPLE`, multiple edges are included in the neighborhood.

Returns:

Error code.

See also:

`igraph_vs_destroy()`

Time complexity: O(1).

igraph_vs_nonadj — Non-adjacent vertices of a vertex.

```
igraph_error_t igraph_vs_nonadj(igraph_vs_t *vs, igraph_int_t vid,
                    igraph_neimode_t mode);
```

All non-neighboring vertices of a given vertex. The *mode* argument controls the type of neighboring vertices *not* to select. Instead of selecting immediate neighbors of `vid` as is done by `igraph_vs_adj()`, the current function selects vertices that are *not* immediate neighbors of `vid`.

Arguments:

vs: Pointer to an uninitialized vertex selector object.

vid: Vertex ID, the “center” of the non-neighborhood.

mode: The type of neighborhood not to select in directed graphs. Possible values:

- `IGRAPH_OUT` All vertices will be selected except those to which there is a directed edge from `vid`. That is, we select all vertices excluding the out-neighbors of `vid`.
- `IGRAPH_IN` All vertices will be selected except those from which there is a directed edge to `vid`. In other words, we select all vertices but the in-neighbors of `vid`.
- `IGRAPH_ALL` All vertices will be selected except those from or to which there is a directed edge to or from `vid`. That is, we select all vertices of `vid` except for its immediate neighbors.

Returns:

Error code.

See also:

`igraph_vs_destroy()`

Time complexity: O(1).

Example 9.1. File `examples/simple/igraph_vs_nonadj.c`

`igraph_vs_none` — Empty vertex set.

```
igraph_error_t igraph_vs_none(igraph_vs_t *vs);
```

Creates an empty vertex selector.

Arguments:

vs: Pointer to an uninitialized vertex selector object.

Returns:

Error code.

See also:

`igraph_vss_none()`, `igraph_vs_destroy()`

Time complexity: O(1).

`igraph_vs_1` — Vertex set with a single vertex.

```
igraph_error_t igraph_vs_1(igraph_vs_t *vs, igraph_int_t vid);
```

This vertex selector selects a single vertex.

Arguments:

vs: Pointer to an uninitialized vertex selector object.

vid: The vertex ID to be selected.

Returns:

Error Code.

See also:

`igraph_vss_1()`, `igraph_vs_destroy()`

Time complexity: O(1).

`igraph_vs_vector` — Vertex set based on a vector.

```
igraph_error_t igraph_vs_vector(igraph_vs_t *vs,
                    const igraph_vector_int_t *v);
```

This function makes it possible to handle an igraph_vector_int_t temporarily as a vertex selector. The vertex selector should be thought of as a *view* into the vector. If you make changes to the vector that also affects the vertex selector. Destroying the vertex selector does not destroy the vector. Do not destroy the vector before destroying the vertex selector, or you might get strange behavior. Since selectors are not tied to any specific graph, this function does not check whether the vertex IDs in the vector are valid.

Arguments:

vs: Pointer to an uninitialized vertex selector.

v: Pointer to a igraph_vector_int_t object.

Returns:

Error code.

See also:

`igraph_vss_vector()`, `igraph_vs_destroy()`

Time complexity: O(1).

Example 9.2. File `examples/simple/igraph_vs_vector.c`

igraph_vs_vector_small — Create a vertex set by giving its elements.

```
igraph_error_t igraph_vs_vector_small(igraph_vs_t *vs, ...);
```

This function can be used to create a vertex selector with a few of vertices. Do not forget to include a `-1` after the last vertex ID. The behavior of the function is undefined if you don't use a `-1` properly.

Note that the vertex IDs supplied will be parsed as value of type int so you cannot supply arbitrarily large (too large for int) vertex IDs here.

Arguments:

vs: Pointer to an uninitialized vertex selector object.

...: Additional parameters, these will be the vertex IDs to be included in the vertex selector. Supply a `-1` after the last vertex ID.

Returns:

Error code.

See also:

`igraph_vs_destroy()`

Time complexity: O(n), the number of vertex IDs supplied.

igraph_vs_vector_copy — Vertex set based on a vector, with copying.

```
igraph_error_t igraph_vs_vector_copy(igraph_vs_t *vs, const igraph_vector_int_t
```

This function makes it possible to handle an igraph_vector_int_t permanently as a vertex selector. The vertex selector creates a copy of the original vector, so the vector can safely be destroyed after creating the vertex selector. Changing the original vector will not affect the vertex selector. The vertex selector is responsible for deleting the copy made by itself. Since selectors are not tied to any specific graph, this function does not check whether the vertex IDs in the vector are valid.

Arguments:

vs: Pointer to an uninitialized vertex selector.

v: Pointer to a igraph_vector_int_t object.

Returns:

Error code.

See also:

`igraph_vs_destroy()`

Time complexity: O(1).

igraph_vs_range — Vertex set, an interval of vertices.

```
igraph_error_t igraph_vs_range(igraph_vs_t *vs, igraph_int_t start, igraph_int_
```

Creates a vertex selector containing all vertices with vertex ID equal to or bigger than *from* and smaller than *to*. Note that the interval is closed from the left and open from the right, following C conventions.

Arguments:

vs: Pointer to an uninitialized vertex selector object.

start: The first vertex ID to be included in the vertex selector.

end: The first vertex ID *not* to be included in the vertex selector.

Returns:

Error code.

See also:

`igraph_vss_range()`, `igraph_vs_destroy()`

Time complexity: O(1).

Example 9.3. File `examples/simple/igraph_vs_range.c`

Generic vertex selector operations

igraph_vs_copy — Creates a copy of a vertex selector.

```
igraph_error_t igraph_vs_copy(igraph_vs_t* dest, const igraph_vs_t* src);
```

Arguments:

dest: An uninitialized selector that will contain the copy.

src: The selector being copied.

Returns:

Error code.

igraph_vs_destroy — Destroy a vertex set.

```
void igraph_vs_destroy(igraph_vs_t *vs);
```

This function should be called for all vertex selectors when they are not needed. The memory allocated for the vertex selector will be deallocated. Do not call this function on vertex selectors created with the immediate versions of the vertex selector constructors (starting with `igraph_vss`).

Arguments:

vs: Pointer to a vertex selector object.

Time complexity: operating system dependent, usually O(1).

`igraph_vs_is_all` — Check whether all vertices are included.

```
igraph_bool_t igraph_vs_is_all(const igraph_vs_t *vs);
```

This function checks whether the vertex selector object was created by `igraph_vs_all()` or `igraph_vss_all()`. Note that the vertex selector might contain all vertices in a given graph but if it wasn't created by the two constructors mentioned here the return value will be `false`.

Arguments:

vs: Pointer to a vertex selector object.

Returns:

`true` if the vertex selector contains all vertices and `false` otherwise.

Time complexity: O(1).

`igraph_vs_size` — Returns the size of the vertex selector.

```
igraph_error_t igraph_vs_size(const igraph_t *graph, const igraph_vs_t *vs,
                  igraph_int_t *result);
```

The size of the vertex selector is the number of vertices it will yield when it is iterated over.

Arguments:

graph: The graph over which we will iterate.

vs: the vertex selector.

result: The result will be returned here.

Returns:

Error code.

`igraph_vs_type` — Returns the type of the vertex selector.

```
igraph_vs_type_t igraph_vs_type(const igraph_vs_t *vs);
```

Immediate vertex selectors

`igraph_vss_all` — All vertices of a graph (immediate version).

```
igraph_vs_t igraph_vss_all(void);
```

Immediate vertex selector for all vertices in a graph. It can be used conveniently when some vertex property (e.g. betweenness, degree, etc.) should be calculated for all vertices.

Returns:

A vertex selector for all vertices in a graph.

See also:

`igraph_vs_all()`

Time complexity: O(1).

`igraph_vss_none` — Empty vertex set (immediate version).

```
igraph_vs_t igraph_vss_none(void);
```

The immediate version of the empty vertex selector.

Returns:

An empty vertex selector.

See also:

`igraph_vs_none()`

Time complexity: O(1).

`igraph_vss_1` — Vertex set with a single vertex (immediate version).

```
igraph_vs_t igraph_vss_1(igraph_int_t vid);
```

The immediate version of the single-vertex selector.

Arguments:

`vid`: The vertex to be selected.

Returns:

A vertex selector containing a single vertex.

See also:

`igraph_vs_1()`

Time complexity: O(1).

`igraph_vss_vector` — Vertex set based on a vector (immediate version).

```
igraph_vs_t igraph_vss_vector(const igraph_vector_int_t *v);
```

This is the immediate version of `igraph_vs_vector`.

Arguments:

v: Pointer to a igraph_vector_int_t object.

Returns:

A vertex selector object containing the vertices in the vector.

See also:

`igraph_vs_vector()`

Time complexity: O(1).

`igraph_vss_range` — An interval of vertices (immediate version).

```
igraph_vs_t igraph_vss_range(igraph_int_t start, igraph_int_t end);
```

The immediate version of `igraph_vs_range()`.

Arguments:

start: The first vertex ID to be included in the vertex selector.

end: The first vertex ID *not* to be included in the vertex selector.

Returns:

Error code.

See also:

`igraph_vs_range()`

Time complexity: O(1).

Vertex iterators

`igraph_vit_create` — Creates a vertex iterator from a vertex selector.

```
igraph_error_t igraph_vit_create(const igraph_t *graph, igraph_vs_t vs, igraph_v
```

This function instantiates a vertex selector object with a given graph. This is the step when the actual vertex IDs are created from the *logical* notion of the vertex selector based on the graph. E.g. a vertex selector created with `igraph_vs_all()` contains knowledge that *all* vertices are included in a (yet indefinite) graph. When instantiating it a vertex iterator object is created, this contains the actual vertex IDs in the graph supplied as a parameter.

The same vertex selector object can be used to instantiate any number vertex iterators.

Arguments:

`graph`: An igraph_t object, a graph.

`vs`: A vertex selector object.

`vit`: Pointer to an uninitialized vertex iterator object.

Returns:

Error code.

See also:

`igraph_vit_destroy()`.

Time complexity: it depends on the vertex selector type. O(1) for vertex selectors created with `igraph_vs_all()`, `igraph_vs_none()`, `igraph_vs_1`, `igraph_vs_vector`, `igraph_vs_range()`, `igraph_vs_vector()`, `igraph_vs_vector_small()`. O(d) for `igraph_vs_adj()`, d is the number of vertex IDs to be included in the iterator. O(|V|) for `igraph_vs_nonadj()`, |V| is the number of vertices in the graph.

`igraph_vit_destroy` — Destroys a vertex iterator.

```
void igraph_vit_destroy(const igraph_vit_t *vit);
```

Deallocates memory allocated for a vertex iterator.

Arguments:

`vit`: Pointer to an initialized vertex iterator object.

See also:

`igraph_vit_create()`

Time complexity: operating system dependent, usually O(1).

Stepping over the vertices

After creating an iterator with `igraph_vit_create()`, it points to the first vertex in the vertex determined by the vertex selector (if there is any). The `IGRAPH_VIT_NEXT()` macro steps to the next vertex, `IGRAPH_VIT_END()` checks whether there are more vertices to visit, `IGRAPH_VIT_SIZE()` gives the total size of the vertices visited so far and to be visited. `IGRAPH_VIT_RESET()` resets the iterator, it will point to the first vertex again. Finally `IGRAPH_VIT_GET()` gives the current vertex pointed to by the iterator (call this only if `IGRAPH_VIT_END()` is false).

Here is an example on how to step over the neighbors of vertex 0:

```
igraph_vs_t vs;
igraph_vit_t vit;
...
igraph_vs_adj(&vs, 0, IGRAPH_ALL);
igraph_vit_create(&graph, vs, &vit);
while (!IGRAPH_VIT_END(vit)) {
  printf(" %" IGRAPH_PRId, IGRAPH_VIT_GET(vit));
  IGRAPH_VIT_NEXT(vit);
}
printf("\n");
...
igraph_vit_destroy(&vit);
igraph_vs_destroy(&vs);
```

IGRAPH_VIT_NEXT — Next vertex.

```
#define IGRAPH_VIT_NEXT(vit)
```

Steps the iterator to the next vertex. Only call this function if `IGRAPH_VIT_END()` returns false.

Arguments:

vit: The vertex iterator to step.

Time complexity: O(1).

IGRAPH_VIT_END — Are we at the end?

```
#define IGRAPH_VIT_END(vit)
```

Checks whether there are more vertices to step to.

Arguments:

vit: The vertex iterator to check.

Returns:

Logical value, if true there are no more vertices to step to.

Time complexity: O(1).

IGRAPH_VIT_SIZE — Size of a vertex iterator.

```
#define IGRAPH_VIT_SIZE(vit)
```

Gives the number of vertices in a vertex iterator.

Arguments:

vit: The vertex iterator.

Returns:

The number of vertices.

Time complexity: O(1).

IGRAPH_VIT_RESET — Reset a vertex iterator.

```
#define IGRAPH_VIT_RESET(vit)
```

Resets a vertex iterator. After calling this macro the iterator will point to the first vertex.

Arguments:

vit: The vertex iterator.

Time complexity: O(1).

IGRAPH_VIT_GET — Query the current position.

```
#define IGRAPH_VIT_GET(vit)
```

Gives the vertex ID of the current vertex pointed to by the iterator.

Arguments:

vit: The vertex iterator.

Returns:

The vertex ID of the current vertex.

Time complexity: O(1).

Edge selector constructors

igraph_es_all — Edge set, all edges.

```
igraph_error_t igraph_es_all(igraph_es_t *es,
                  igraph_edgeorder_type_t order);
```

Arguments:

es: Pointer to an uninitialized edge selector object.

order: Constant giving the order in which the edges will be included in the selector. Possible values:

IGRAPH_EDGEORDER_ID	Edge ID order; currently performs the fastest.
IGRAPH_EDGEORDER_FROM	Vertex ID order, the id of the *source* vertex counts for directed graphs. The order of the incident edges of a given vertex is arbitrary.
IGRAPH_EDGEORDER_TO	Vertex ID order, the ID of the *target* vertex counts for directed graphs. The order of the incident edges of a given vertex is arbitrary.

For undirected graph the latter two is the same.

Returns:

Error code.

See also:

igraph_ess_all(), igraph_es_destroy()

Time complexity: O(1).

igraph_es_incident — Edges incident on a given vertex.

```
igraph_error_t igraph_es_incident(
    igraph_es_t *es, igraph_int_t vid, igraph_neimode_t mode,
    igraph_loops_t loops
);
```

Arguments:

es: Pointer to an uninitialized edge selector object.

vid: Vertex ID, of which the incident edges will be selected.

mode: Constant giving the type of the incident edges to select. This is ignored for undirected graphs. Possible values: IGRAPH_OUT, outgoing edges; IGRAPH_IN, incoming edges; IGRAPH_ALL, all edges.

loops: Whether to include loop edges in the result. If IGRAPH_NO_LOOPS, loop edges are excluded. If IGRAPH_LOOPS_ONCE, loop edges are included once. If IGRAPH_LOOPS_TWICE, loop edges are included twice, but only if the graph is undirected or *mode* is set to IGRAPH_ALL.

Returns:

Error code.

See also:

igraph_es_destroy()

Time complexity: O(1).

igraph_es_none — Empty edge selector.

```
igraph_error_t igraph_es_none(igraph_es_t *es);
```

Arguments:

es: Pointer to an uninitialized edge selector object to initialize.

Returns:

Error code.

See also:

`igraph_ess_none()`, `igraph_es_destroy()`

Time complexity: O(1).

igraph_es_1 — Edge selector containing a single edge.

```
igraph_error_t igraph_es_1(igraph_es_t *es, igraph_int_t eid);
```

Arguments:

es: Pointer to an uninitialized edge selector object.

eid: Edge ID of the edge to select.

Returns:

Error code.

See also:

`igraph_ess_1()`, `igraph_es_destroy()`

Time complexity: O(1).

igraph_es_all_between — Edge selector, all edge IDs between a pair of vertices.

```
igraph_error_t igraph_es_all_between(
    igraph_es_t *es, igraph_int_t from, igraph_int_t to,
    igraph_bool_t directed
);
```

This function takes a pair of vertices and creates a selector that matches all edges between those vertices.

Arguments:

es: Pointer to an uninitialized edge selector object.

from: The ID of the source vertex.

to: The ID of the target vertex.

directed: If edge directions should be taken into account. This will be ignored if the graph to select from is undirected.

Returns:

Error code.

See also:

`igraph_es_destroy()`

Time complexity: O(1).

`igraph_es_vector` — Handle a vector as an edge selector.

```
igraph_error_t igraph_es_vector(igraph_es_t *es, const igraph_vector_int_t *v);
```

Creates an edge selector which serves as a view into a vector containing edge IDs. Do not destroy the vector before destroying the edge selector. Since selectors are not tied to any specific graph, this function does not check whether the edge IDs in the vector are valid.

Arguments:

es: Pointer to an uninitialized edge selector.

v: Vector containing edge IDs.

Returns:

Error code.

See also:

`igraph_ess_vector()`, `igraph_es_destroy()`

Time complexity: O(1).

`igraph_es_range` — Edge selector, a sequence of edge IDs.

```
igraph_error_t igraph_es_range(igraph_es_t *es, igraph_int_t start, igraph_int_
```

Creates an edge selector containing all edges with edge ID equal to or bigger than *from* and smaller than *to*. Note that the interval is closed from the left and open from the right, following C conventions.

Arguments:

es: Pointer to an uninitialized edge selector object.

start: The first edge ID to be included in the edge selector.

end: The first edge ID *not* to be included in the edge selector.

Returns:

Error code.

See also:

`igraph_ess_range()`, `igraph_es_destroy()`

Time complexity: O(1).

`igraph_es_pairs` — Edge selector, multiple edges defined by their endpoints in a vector.

```
igraph_error_t igraph_es_pairs(igraph_es_t *es, const igraph_vector_int_t *v,
                    igraph_bool_t directed);
```

The edges between the given pairs of vertices will be included in the edge selection. The vertex pairs must be defined in the vector `v`, the first element of the vector is the first vertex of the first edge to be selected, the second element is the second vertex of the first edge, the third element is the first vertex of the second edge and so on.

Arguments:

es: Pointer to an uninitialized edge selector object.

v: The vector containing the endpoints of the edges.

directed: Whether the graph is directed or not.

Returns:

Error code.

See also:

`igraph_es_pairs_small()`, `igraph_es_destroy()`

Time complexity: O(n), the number of edges being selected.

Example 9.4. File `examples/simple/igraph_es_pairs.c`

`igraph_es_pairs_small` — Edge selector, multiple edges defined by their endpoints as arguments.

```
igraph_error_t igraph_es_pairs_small(igraph_es_t *es, igraph_bool_t directed, i
```

The edges between the given pairs of vertices will be included in the edge selection. The vertex pairs must be given as the arguments of the function call, the third argument is the first vertex of the first edge, the fourth argument is the second vertex of the first edge, the fifth is the first vertex of the second edge and so on. The last element of the argument list must be -1 to denote the end of the argument list.

Note that the vertex IDs supplied will be parsed as `int`'s so you cannot supply arbitrarily large (too large for int) vertex IDs here.

Arguments:

es: Pointer to an uninitialized edge selector object.

directed: Whether the graph is directed or not.

...: The additional arguments give the edges to be included in the selector, as pairs of vertex IDs. The last argument must be -1. The *first* parameter is present for technical reasons and represents the first variadic argument.

Returns:

Error code.

See also:

`igraph_es_pairs()`, `igraph_es_destroy()`

Time complexity: O(n), the number of edges being selected.

`igraph_es_path` — Edge selector, edge IDs on a path.

```
igraph_error_t igraph_es_path(igraph_es_t *es, const igraph_vector_int_t *v,
                  igraph_bool_t directed);
```

This function takes a vector of vertices and creates a selector of edges between those vertices. Vector {0, 3, 4, 7} will select edges (0 -> 3), (3 -> 4), (4 -> 7). If these edges don't exist then trying to create an iterator using this selector will fail.

Arguments:

es: Pointer to an uninitialized edge selector object.

v: Pointer to a vector of vertex IDs along the path.

directed: If edge directions should be taken into account. This will be ignored if the graph to select from is undirected.

Returns:

Error code.

See also:

`igraph_es_destroy()`

Time complexity: O(n), the number of vertices.

`igraph_es_vector_copy` — Edge set, based on a vector, with copying.

```
igraph_error_t igraph_es_vector_copy(igraph_es_t *es, const igraph_vector_int_t
```

This function makes it possible to handle an igraph_vector_int_t permanently as an edge selector. The edge selector creates a copy of the original vector, so the vector can safely be destroyed after creating the edge selector. Changing the original vector will not affect the edge selector. The edge selector is responsible for deleting the copy made by itself. Since selectors are not tied to any specific graph, this function does not check whether the edge IDs in the vector are valid.

Arguments:

es: Pointer to an uninitialized edge selector.

v: Pointer to a igraph_vector_int_t object.

Returns:

Error code.

See also:

`igraph_es_destroy()`

Time complexity: O(1).

Immediate edge selectors

`igraph_ess_all` — Edge set, all edges (immediate version).

```
igraph_es_t igraph_ess_all(igraph_edgeorder_type_t order);
```

The immediate version of the all-edges selector.

Arguments:

order: Constant giving the order of the edges in the edge selector. See `igraph_es_all()` for the possible values.

Returns:

The edge selector.

See also:

`igraph_es_all()`

Time complexity: O(1).

`igraph_ess_none` — Immediate empty edge selector.

```
igraph_es_t igraph_ess_none(void);
```

Immediate version of the empty edge selector.

Returns:

Initialized empty edge selector.

See also:

`igraph_es_none()`

Time complexity: O(1).

`igraph_ess_1` — Immediate version of the single edge edge selector.

```
igraph_es_t igraph_ess_1(igraph_int_t eid);
```

Arguments:

`eid`: The ID of the edge.

Returns:

The edge selector.

See also:

`igraph_es_1()`

Time complexity: O(1).

`igraph_ess_vector` — Immediate vector view edge selector.

```
igraph_es_t igraph_ess_vector(const igraph_vector_int_t *v);
```

This is the immediate version of the vector of edge IDs edge selector.

Arguments:

`v`: The vector of edge IDs.

Returns:

Edge selector, initialized.

See also:

`igraph_es_vector()`

Time complexity: O(1).

`igraph_ess_range` — Immediate version of the sequence edge selector.

```
igraph_es_t igraph_ess_range(igraph_int_t start, igraph_int_t end);
```

Arguments:

`start`: The first edge ID to be included in the edge selector.

`end`: The first edge ID *not* to be included in the edge selector.

Returns:

The initialized edge selector.

See also:

`igraph_es_range()`

Time complexity: O(1).

Generic edge selector operations

`igraph_es_as_vector` — Transform edge selector into vector.

```
igraph_error_t igraph_es_as_vector(const igraph_t *graph, igraph_es_t es,
                        igraph_vector_int_t *v);
```

Call this function on an edge selector to transform it into a vector. This is only implemented for sequence and vector selectors. If the edges do not exist in the graph, this will result in an error.

Arguments:

`graph`: Pointer to a graph to check if the edges in the selector exist.

`es`: An edge selector object.

`v`: Pointer to initialized vector. The result will be stored here.

Returns:

Error code.

Time complexity: O(n), the number of edges in the selector.

`igraph_es_copy` — Creates a copy of an edge selector.

```
igraph_error_t igraph_es_copy(igraph_es_t* dest, const igraph_es_t* src);
```

Arguments:

dest: An uninitialized selector that will contain the copy.

src: The selector being copied.

Returns:

Error code.

See also:

`igraph_es_destroy()`

`igraph_es_destroy` — Destroys an edge selector object.

```
void igraph_es_destroy(igraph_es_t *es);
```

Call this function on an edge selector when it is not needed any more. Do *not* call this function on edge selectors created by immediate constructors, those don't need to be destroyed.

Arguments:

es: Pointer to an edge selector object.

Time complexity: operating system dependent, usually O(1).

`igraph_es_is_all` — Check whether an edge selector includes all edges.

```
igraph_bool_t igraph_es_is_all(const igraph_es_t *es);
```

Arguments:

es: Pointer to an edge selector object.

Returns:

`true` if *es* was created with `igraph_es_all()` or `igraph_ess_all()`, and `false` otherwise.

Time complexity: O(1).

`igraph_es_size` — Returns the size of the edge selector.

```
igraph_error_t igraph_es_size(const igraph_t *graph, const igraph_es_t *es,
                  igraph_int_t *result);
```

The size of the edge selector is the number of edges it will yield when it is iterated over.

Arguments:

graph: The graph over which we will iterate.

es: The edge selector.

result: The result will be returned here.

Returns:

Error code.

igraph_es_type — Returns the type of the edge selector.

```
igraph_es_type_t igraph_es_type(const igraph_es_t *es);
```

Edge iterators

igraph_eit_create — Creates an edge iterator from an edge selector.

```
igraph_error_t igraph_eit_create(const igraph_t *graph, igraph_es_t es, igraph_
```

This function creates an edge iterator based on an edge selector and a graph.

The same edge selector can be used to create many edge iterators, also for different graphs.

Arguments:

graph: An igraph_t object for which the edge selector will be instantiated.

es: The edge selector to instantiate.

eit: Pointer to an uninitialized edge iterator.

Returns:

Error code.

See also:

`igraph_eit_destroy()`

Time complexity: depends on the type of the edge selector. For edge selectors created by `igraph_es_all()`, `igraph_es_none()`, `igraph_es_1()`, `igraph_es_vector()`,

`igraph_es_range()` it is O(1). For `igraph_es_incident()` it is O(d) where d is the number of incident edges of the vertex.

`igraph_eit_destroy` — Destroys an edge iterator.

```
void igraph_eit_destroy(const igraph_eit_t *eit);
```

Arguments:

`eit`: Pointer to an edge iterator to destroy.

See also:

`igraph_eit_create()`

Time complexity: operating system dependent, usually O(1).

Stepping over the edges

Just like for vertex iterators, macros are provided for stepping over a sequence of edges: `IGRAPH_EIT_NEXT()` goes to the next edge, `IGRAPH_EIT_END()` checks whether there are more edges to visit, `IGRAPH_EIT_SIZE()` gives the number of edges in the edge sequence, `IGRAPH_EIT_RESET()` resets the iterator to the first edge and `IGRAPH_EIT_GET()` returns the id of the current edge.

`IGRAPH_EIT_NEXT` — Next edge.

```
#define IGRAPH_EIT_NEXT(eit)
```

Steps the iterator to the next edge. Call this function only if `IGRAPH_EIT_END()` returns false.

Arguments:

`eit`: The edge iterator to step.

Time complexity: O(1).

`IGRAPH_EIT_END` — Are we at the end?

```
#define IGRAPH_EIT_END(eit)
```

Checks whether there are more edges to step to.

Arguments:

`wit`: The edge iterator to check.

Returns:

Logical value, if true there are no more edges to step to.

Time complexity: O(1).

`IGRAPH_EIT_SIZE` — Number of edges in the iterator.

```
#define IGRAPH_EIT_SIZE(eit)
```

Gives the number of edges in an edge iterator.

Arguments:

eit: The edge iterator.

Returns:

The number of edges.

Time complexity: O(1).

IGRAPH_EIT_RESET — Reset an edge iterator.

```
#define IGRAPH_EIT_RESET(eit)
```

Resets an edge iterator. After calling this macro the iterator will point to the first edge.

Arguments:

eit: The edge iterator.

Time complexity: O(1).

IGRAPH_EIT_GET — Query an edge iterator.

```
#define IGRAPH_EIT_GET(eit)
```

Gives the edge ID of the current edge pointed to by an iterator.

Arguments:

eit: The edge iterator.

Returns:

The id of the current edge.

Time complexity: O(1).

Chapter 10. Graph, vertex and edge attributes

Attributes are numbers, boolean values or strings associated with the vertices or edges of a graph, or with the graph itself. E.g. you may label vertices with symbolic names or attach numeric weights to the edges of a graph. In addition to these three basic types, a custom object type is supported as well.

igraph attributes are designed to be flexible and extensible. In igraph attributes are implemented via an interface abstraction: any type implementing the functions in the interface can be used for storing vertex, edge and graph attributes. This means that different attribute implementations can be used together with igraph. This is reasonable: if igraph is used from Python attributes can be of any Python type, from R all R types are allowed. There is also an experimental attribute implementation to be used when programming in C, but by default it is currently turned off.

First we briefly look over how attribute handlers can be implemented. This is not something a user does every day. It is rather typically the job of the high level interface writers. (But it is possible to write an interface without implementing attributes.) Then we show the experimental C attribute handler.

The attribute handler interface

It is possible to attach an attribute handling interface to **igraph**. This is simply a table of functions, of type `igraph_attribute_table_t`. These functions are invoked to notify the attribute handling code about the structural changes in a graph. See the documentation of this type for details.

By default there is no attribute interface attached to **igraph**. To attach one, call `igraph_set_attribute_table` with your new table. This is normally done on program startup, and is kept untouched for the program's lifetime. It must be done before any graph object is created, as graphs created with a given attribute handler cannot be manipulated while a different attribute handler is active.

`igraph_attribute_table_t` — Table of functions to perform operations on attributes.

```
typedef struct igraph_attribute_table_t {
    igraph_error_t (*init)(igraph_t *graph, const igraph_attribute_record_list_t
    void           (*destroy)(igraph_t *graph);
    igraph_error_t (*copy)(igraph_t *to, const igraph_t *from, igraph_bool_t ga
                           igraph_bool_t va, igraph_bool_t ea);
    igraph_error_t (*add_vertices)(
        igraph_t *graph, igraph_int_t nv,
        const igraph_attribute_record_list_t *attr
    );
    igraph_error_t (*permute_vertices)(const igraph_t *graph,
                                       igraph_t *newgraph,
                                       const igraph_vector_int_t *idx);
    igraph_error_t (*combine_vertices)(const igraph_t *graph,
                                       igraph_t *newgraph,
                                       const igraph_vector_int_list_t *merges,
                                       const igraph_attribute_combination_t *com
    igraph_error_t (*add_edges)(
        igraph_t *graph, const igraph_vector_int_t *edges,
        const igraph_attribute_record_list_t *attr
    );
```

```
    igraph_error_t (*permute_edges)(const igraph_t *graph,
                                    igraph_t *newgraph, const igraph_vector_int
    igraph_error_t (*combine_edges)(const igraph_t *graph,
                                    igraph_t *newgraph,
                                    const igraph_vector_int_list_t *merges,
                                    const igraph_attribute_combination_t *comb)
    igraph_error_t (*get_info)(const igraph_t *graph,
                               igraph_strvector_t *gnames, igraph_vector_int_t
                               igraph_strvector_t *vnames, igraph_vector_int_t
                               igraph_strvector_t *enames, igraph_vector_int_t
    igraph_bool_t (*has_attr)(const igraph_t *graph, igraph_attribute_elemtype_
                              const char *name);
    igraph_error_t (*get_type)(const igraph_t *graph, igraph_attribute_type_t *
                               igraph_attribute_elemtype_t elemtype, const char
    igraph_error_t (*get_numeric_graph_attr)(const igraph_t *graph, const char
                                             igraph_vector_t *value);
    igraph_error_t (*get_string_graph_attr)(const igraph_t *graph, const char *
                                            igraph_strvector_t *value);
    igraph_error_t (*get_bool_graph_attr)(const igraph_t *igraph, const char *na
                                          igraph_vector_bool_t *value);
    igraph_error_t (*get_numeric_vertex_attr)(const igraph_t *graph, const char
                                              igraph_vs_t vs,
                                              igraph_vector_t *value);
    igraph_error_t (*get_string_vertex_attr)(const igraph_t *graph, const char
                                             igraph_vs_t vs,
                                             igraph_strvector_t *value);
    igraph_error_t (*get_bool_vertex_attr)(const igraph_t *graph, const char *na
                                           igraph_vs_t vs,
                                           igraph_vector_bool_t *value);
    igraph_error_t (*get_numeric_edge_attr)(const igraph_t *graph, const char *
                                            igraph_es_t es,
                                            igraph_vector_t *value);
    igraph_error_t (*get_string_edge_attr)(const igraph_t *graph, const char *na
                                           igraph_es_t es,
                                           igraph_strvector_t *value);
    igraph_error_t (*get_bool_edge_attr)(const igraph_t *graph, const char *name
                                         igraph_es_t es,
                                         igraph_vector_bool_t *value);
} igraph_attribute_table_t;
```

This type collects the functions defining an attribute handler. It has the following members:

Values:

`init`: This function is called whenever a new graph object is created, right after it is created but before any vertices or edges are added. It is supposed to set the `attr` member of the `igraph_t` object, which is guaranteed to be set to a null pointer before this function is called. It is expected to set the `attr` member to a non-null value *or* return an error code. Leaving the `attr` member at a null value while returning success is invalid and will trigger an error in the C core of igraph itself.

`destroy`: This function is called whenever the graph object is destroyed, right before freeing the allocated memory. It is supposed to do any cleanup operations that are need to dispose of the `attr` member of the `igraph_t` object properly. The caller will set the `attr` member to a null pointer after this function returns.

`copy`: This function is called when the C core wants to populate the attributes of a graph from another graph. The structure of the target graph is already initialized by the time this function is called, and the `attr` member of the graph is set to a null pointer. The function is supposed to populate the `attr` member of the target `igraph_t` object to a non-null value *or* return an error code. Leaving the `attr` member at a null value while returning success is invalid and will trigger an error in the C core of igraph itself.

`add_vertices`: Called when vertices are added to a graph, after the base data structure was modified. The number of vertices that were added is supplied as an argument. The function is supposed to set up default values for each vertex attribute that is currently registered on the graph, for all the newly added vertices. Expected to return an error code.

`permute_vertices`: Called when a new graph is created based on an existing one such that there is a mapping from the vertices of the new graph back to the vertices of the old graph (e.g. if vertices are removed from a graph). The supplied index vector defines which old vertex a new vertex corresponds to. Its length is the same as the number of vertices in the new graph, and for each new vertex it provides the ID of the corresponding vertex in the old graph. The function is supposed to set up the values of the vertex attributes of the new graph based on the attributes of the old graph and the provided index vector. Note that the old and the new graph *may* be the same, in which case it is the responsibility of the function to ensure that the operation can safely be performed in-place. If the two graph instances are *not* the same, implementors may safely assume that the new graph has no vertex attributes yet (but it may already have graph or edge attributes by the time this function is called).

`combine_vertices`: This function is called when the creation of a new graph involves a merge (contraction, etc.) of vertices from another graph. The function is called after the new graph was created. An argument specifies how several vertices from the old graph map to a single vertex in the new graph. It is guaranteed that the old and the new graph instances are different when this callback is called. Implementors may safely assume that the new graph has no vertex attributes yet (but it may already have graph or edge attributes by the time this function is called).

`add_edges`: Called when new edges are added to a graph, after the base data structure was modified. A vector containing the endpoints of the new edges are supplied as an argument. The function is supposed to set up default values for each edge attribute that is currently registered on the graph, for all the newly added edges. Expected to return an error code.

`permute_edges`: Called when a new graph is created based on an existing one such that some of the edges in the new graph should copy the attributes of some edges from the old graph (this also includes the deletion of edges). The supplied index vector defines which old edge a new edge corresponds to. Its length is the same as the number of edges in the new graph, and for each edge it provides the ID of the corresponding edge in the old graph. The function is supposed to set up the values of the edge attributes of the new

graph based on the attributes of the old graph and the provided index vector. Note that the old and the new graph *may* be the same, in which case it is the responsibility of the function to ensure that the operation can safely be performed in-place. If the two graph instances are *not* the same, implementors may safely assume that the new graph has no edge attributes yet (but it may already have graph or vertex attributes by the time this function is called).

`combine_edges`: This function is called when the creation of a new graph involves a merge (contraction, etc.) of edges from another graph. The function is after the new graph was created. An argument specifies how several edges from the old graph map to a single edge in the new graph. It is guaranteed that the old and the new graph instances are different when this callback is called. Implementors may safely assume that the new graph has no edge attributes yet (but it may already have graph or vertex attributes by the time this function is called).

`get_info`: Query the attributes of a graph, the names and types should be returned.

`has_attr`: Check whether a graph has the named graph/vertex/edge attribute.

`get_type`: Query the type of a graph/vertex/edge attribute.

`get_numeric_graph_attr`: Query a numeric graph attribute. The value should be appended to the provided *`value`* vector. No assumptions should be made about the initial contents of the *`value`* vector and it is not guaranteed to be empty.

`get_string_graph_attr`: Query a string graph attribute. The value should be appended to the provided *`value`* vector. No assumptions should be made about the initial contents of the *`value`* vector and it is not guaranteed to be empty.

`get_bool_graph_attr`: Query a boolean graph attribute. The value should be appended to the provided *`value`* vector. No assumptions should be made about the initial contents of the *`value`* vector and it is not guaranteed to be empty.

`get_numeric_vertex_attr`: Query a numeric vertex attribute, for the vertices included in *`vs`*. The attribute values should be appended to the provided *`value`* vector. No assumptions should be made about the initial contents of the *`value`* vector and it is not guaranteed to be empty.

`get_string_vertex_attr`: Query a string vertex attribute, for the vertices included in *`vs`*. The attribute values should be appended to the provided *`value`* vector. No assumptions should be made about the initial contents of the *`value`* vector and it is not guaranteed to be empty.

`get_bool_vertex_attr`: Query a boolean vertex attribute, for the vertices included in *`vs`*. The attribute values should be appended to the provided *`value`* vector. No assumptions should be made about the initial contents of the *`value`* vector and it is not guaranteed to be empty.

`get_numeric_edge_attr`: Query a numeric edge attribute, for the edges included in *`es`*. The attribute values should be appended to the provided *`value`* vector. No assumptions should be made about the initial contents of the *`value`* vector and it is not guaranteed to be empty.

`get_string_edge_attr`: Query a string edge attribute, for the the edges included in *`es`*. The attribute values should be appended to the provided *`value`* vector. No assumptions should be made about the initial contents of the *`value`* vector and it is not guaranteed to be empty.

`get_bool_edge_attr`: Query a boolean edge attribute, for the the edges included in *`es`*. The attribute values should be appended to the provided *`value`* vector. No assumptions should be made about the initial contents of the *`value`* vector and it is not guaranteed to be empty.

`igraph_set_attribute_table` — Attach an attribute table.

```
igraph_attribute_table_t *
igraph_set_attribute_table(const igraph_attribute_table_t * table);
```

This function attaches attribute handling code to the igraph library. Note that the attribute handler table is *not* thread-local even if igraph is compiled in thread-local mode. In the vast majority of cases, this is not a significant restriction.

Attribute handlers are normally attached on program startup, and are left active for the program's lifetime. This is because a graph object created with a given attribute handler must not be manipulated while a different attribute handler is active.

Arguments:

`table`: Pointer to an `igraph_attribute_table_t` object containing the functions for attribute manipulation. Supply `NULL` here if you don't want attributes.

Returns:

Pointer to the old attribute handling table.

Time complexity: O(1).

`igraph_attribute_type_t` — The possible types of the attributes.

```
typedef enum {
    IGRAPH_ATTRIBUTE_UNSPECIFIED = 0,
    IGRAPH_ATTRIBUTE_NUMERIC = 1,
    IGRAPH_ATTRIBUTE_BOOLEAN = 2,
    IGRAPH_ATTRIBUTE_STRING = 3,
    IGRAPH_ATTRIBUTE_OBJECT = 127
} igraph_attribute_type_t;
```

Values of this enum are used by the attribute interface to communicate the type of an attribute to igraph's C core. When igraph is integrated in a high-level language, the attribute type reported by the interface may not necessarily have to match the exact data type in the high-level language as long as the attribute interface can provide a conversion from the native high-level attribute value to one of the data types listed here. When the high-level data type is complex and has no suitable conversion to one of the atomic igraph attribute types (numeric, string or Boolean), the attribute interface should report the attribute as having an "object" type, which is ignored by the C core. See also `igraph_attribute_table_t`.

Values:

`IGRAPH_ATTRIBUTE_UNSPECIFIED`:	Currently used internally as a "null value" or "placeholder value" in some algorithms. Attribute records with this type must not be passed to igraph functions.
`IGRAPH_ATTRIBUTE_NUMERIC`:	Numeric attribute.
`IGRAPH_ATTRIBUTE_BOOLEAN`:	Logical values, true or false.
`IGRAPH_ATTRIBUTE_STRING`:	String attribute.
`IGRAPH_ATTRIBUTE_OBJECT`:	Custom attribute type, to be used for special data types by client applications. The R and Python interfaces use this for attributes that hold R or Python objects. Usually ignored by igraph functions.

`igraph_attribute_elemtype_t` — Types of objects to which attributes can be attached.

```
typedef enum {
    IGRAPH_ATTRIBUTE_GRAPH = 0,
    IGRAPH_ATTRIBUTE_VERTEX,
    IGRAPH_ATTRIBUTE_EDGE
} igraph_attribute_elemtype_t;
```

Values:

`IGRAPH_ATTRIBUTE_GRAPH`:	Denotes that an attribute belongs to the entire graph.
`IGRAPH_ATTRIBUTE_VERTEX`:	Denotes that an attribute belongs to the vertices of a graph.
`IGRAPH_ATTRIBUTE_EDGE`:	Denotes that an attribute belongs to the edges of a graph.

Attribute records

Functions in the attribute handler interface may refer to *"attribute* records" or *"attribute* record lists". An attribute record is simply a triplet consisting of an attribute name, an attribute type and a vector containing the values of the attribute. Attribute record lists are typed containers that contain a sequence of attribute records. Attribute record lists own the attribute records that they contain, and similarly, attribute records own the vectors contained in them. Destroying an attribute record destroys the vector of values inside it, and destroying an attribute record list destroys all attribute records in the list.

`igraph_attribute_record_t` — An attribute record holding the name, type and values of an attribute.

```
typedef struct igraph_attribute_record_t {
    char *name;
```

This composite data type is used in the attribute interface to specify a name-type-value triplet where the name is the name of a graph, vertex or edge attribute, the type is the corresponding igraph type of the attribute and the value is a *vector* of attribute values. Note that for graph attributes we use a vector of length 1. The type of the vector depends on the attribute type: it is `igraph_vector_t` for numeric attributes, `igraph_strvector_t` for string attributes and `igraph_vector_bool_t` for Boolean attributes.

The record also stores default values for the attribute. The default values are used when the value vector of the record is resized with `igraph_attribute_record_resize()`. It is important that the record stores *one* default value only, corresponding to the type of the attribute record. The default value is *cleared* when the type of the record is changed.

`igraph_attribute_record_init` — Initializes an attribute record with a given name and type.

```
igraph_error_t igraph_attribute_record_init(
    igraph_attribute_record_t *attr, const char* name, igraph_attribute_type_t
);
```

Arguments:

`attr`: the attribute record to initialize

`name`: name of the attribute

`type`: type of the attribute

Returns:

Error code: `IGRAPH_ENOMEM` if there is not enough memory.

Time complexity: O(1).

`igraph_attribute_record_init_copy` — Initializes an attribute record by copying another record.

```
igraph_error_t igraph_attribute_record_init_copy(
    igraph_attribute_record_t *to, const igraph_attribute_record_t *from
);
```

Copies made by this function are deep copies: a full copy of the value vector contained in the record is placed in the new record so they become independent of each other.

Arguments:

`to`: the attribute record to initialize

from: the attribute record to copy data from

Returns:

Error code: `IGRAPH_ENOMEM` if there is not enough memory.

Time complexity: operating system dependent, usually O(n), where n is the size of the value vector in the attribute record.

`igraph_attribute_record_size` — Returns the size of the value vector in an attribute record.

```
igraph_int_t igraph_attribute_record_size(const igraph_attribute_record_t *attr
```

Arguments:

attr: the attribute record to query

Returns:

the number of elements in the value vector of the attribute record

`igraph_attribute_record_resize` — Resizes the value vector in an attribute record.

```
igraph_error_t igraph_attribute_record_resize(
    igraph_attribute_record_t *attr, igraph_int_t new_size
);
```

When the value vector is shorter than the desired length, it will be expanded with `IGRAPH_NAN` for numeric vectors, `false` for Boolean vectors and empty strings for string vectors.

Arguments:

attr: the attribute record to update

new_size: the new size of the value vector

Returns:

Error code: `IGRAPH_ENOMEM` if there is not enough memory. `IGRAPH_EINVAL` if the type of the attribute record is not specified yet.

`igraph_attribute_record_set_name` — Sets the attribute name in an attribute record.

```
igraph_error_t igraph_attribute_record_set_name(
    igraph_attribute_record_t *attr, const char* name
);
```

Arguments:

attr: the attribute record to update

name: the new name

Returns:

Error code: IGRAPH_ENOMEM if there is not enough memory.

igraph_attribute_record_set_type — Sets the type of an attribute record.

```
igraph_error_t igraph_attribute_record_set_type(
    igraph_attribute_record_t *attr, igraph_attribute_type_t type
);
```

When the new type being set is different from the old type, any values already stored in the attribute record will be destroyed and a new, empty attribute value vector will be allocated. When the new type is the same as the old type, this function is a no-op.

Arguments:

attr: the attribute record to update

type: the new type

Returns:

Error code: IGRAPH_ENOMEM if there is not enough memory.

igraph_attribute_record_set_default_numeric — Sets the default value of the attribute to the given number.

```
igraph_error_t igraph_attribute_record_set_default_numeric(
    igraph_attribute_record_t *attr, igraph_real_t value
);
```

This function must be called for numeric attribute records only. When not specified, the default value of numeric attributes is NaN.

Arguments:

attr: the attribute record to update

value: the new default value

Returns:

Error code: `IGRAPH_EINVAL` if the attribute record has a non-numeric type

`igraph_attribute_record_set_default_string` — Sets the default value of the attribute to the given string.

```
igraph_error_t igraph_attribute_record_set_default_string(
    igraph_attribute_record_t *attr, const char* value
);
```

This function must be called for string attribute records only. When not specified, the default value of string attributes is an empty string.

Arguments:

`attr`: the attribute record to update

`value`: the new default value. `NULL` means an empty string.

Returns:

Error code: `IGRAPH_ENOMEM` if there is not enough memory `IGRAPH_EINVAL` if the attribute record is not of string type

`igraph_attribute_record_set_default_boolean` — Sets the default value of the attribute to the given logical value.

```
igraph_error_t igraph_attribute_record_set_default_boolean(
    igraph_attribute_record_t *attr, igraph_bool_t value
);
```

This function must be called for Boolean attribute records only. When not specified, the default value of Boolean attributes is `false`.

Arguments:

`attr`: the attribute record to update

`value`: the new default value

Returns:

Error code: `IGRAPH_EINVAL` if the attribute record is not of Boolean type

`igraph_attribute_record_destroy` — Destroys an attribute record.

```
void igraph_attribute_record_destroy(igraph_attribute_record_t *attr);
```

Arguments:

attr: the previously initialized attribute record to destroy.

Time complexity: operating system dependent.

Handling attribute combination lists

Several graph operations may collapse multiple vertices or edges into a single one. Attribute combination lists are used to indicate to the attribute handler how to combine the attributes of the original vertices or edges and how to derive the final attribute value that is to be assigned to the collapsed vertex or edge. For example, `igraph_simplify()` removes loops and combines multiple edges into a single one; in case of a graph with an edge attribute named `weight` the attribute combination list can tell the attribute handler whether the weight of a collapsed edge should be the sum, the mean or some other function of the weights of the original edges that were collapsed into one.

One attribute combination list may contain several attribute combination records, one for each vertex or edge attribute that is to be handled during the operation.

`igraph_attribute_combination_init` — Initialize attribute combination list.

```
igraph_error_t igraph_attribute_combination_init(igraph_attribute_combination_t
```

Arguments:

comb: The uninitialized attribute combination list.

Returns:

Error code.

Time complexity: O(1)

`igraph_attribute_combination_add` — Add combination record to attribute combination list.

```
igraph_error_t igraph_attribute_combination_add(igraph_attribute_combination_t
                                        const char *name,
                                        igraph_attribute_combination_type_t type,
                                        igraph_function_pointer_t func);
```

Arguments:

comb: The attribute combination list.

name: The name of the attribute. If the name already exists the attribute combination record will be replaced. Use NULL to add a default combination record for all atributes not in the list.

type: The type of the attribute combination. See `igraph_attribute_combination_type_t` for the options.

func: Function to be used if *type* is `IGRAPH_ATTRIBUTE_COMBINE_FUNCTION`. This function is called by the concrete attribute handler attached to igraph, and its calling signature depends completely on the attribute handler. For instance, if you are using attributes from C and you have attached the C attribute handler, you need to follow the documentation of the C attribute handler for more details.

Returns:

Error code.

Time complexity: O(n), where n is the number of current attribute combinations.

`igraph_attribute_combination_remove` — Remove a record from an attribute combination list.

```
igraph_error_t igraph_attribute_combination_remove(igraph_attribute_combination
                                        const char *name);
```

Arguments:

comb: The attribute combination list.

name: The attribute name of the attribute combination record to remove. It will be ignored if the named attribute does not exist. It can be NULL to remove the default combination record.

Returns:

Error code. This currently always returns IGRAPH_SUCCESS.

Time complexity: O(n), where n is the number of records in the attribute combination list.

`igraph_attribute_combination_destroy` — Destroy attribute combination list.

```
void igraph_attribute_combination_destroy(igraph_attribute_combination_t *comb)
```

Arguments:

comb: The attribute combination list.

Time complexity: O(n), where n is the number of records in the attribute combination list.

`igraph_attribute_combination_type_t` — The possible types of attribute combinations.

```
typedef enum {
    IGRAPH_ATTRIBUTE_COMBINE_IGNORE = 0,
```

```
    IGRAPH_ATTRIBUTE_COMBINE_DEFAULT = 1,
    IGRAPH_ATTRIBUTE_COMBINE_FUNCTION = 2,
    IGRAPH_ATTRIBUTE_COMBINE_SUM = 3,
    IGRAPH_ATTRIBUTE_COMBINE_PROD = 4,
    IGRAPH_ATTRIBUTE_COMBINE_MIN = 5,
    IGRAPH_ATTRIBUTE_COMBINE_MAX = 6,
    IGRAPH_ATTRIBUTE_COMBINE_RANDOM = 7,
    IGRAPH_ATTRIBUTE_COMBINE_FIRST = 8,
    IGRAPH_ATTRIBUTE_COMBINE_LAST = 9,
    IGRAPH_ATTRIBUTE_COMBINE_MEAN = 10,
    IGRAPH_ATTRIBUTE_COMBINE_MEDIAN = 11,
    IGRAPH_ATTRIBUTE_COMBINE_CONCAT = 12
} igraph_attribute_combination_type_t;
```

Values:

`IGRAPH_ATTRIBUTE_COMBINE_IGNORE`: Ignore old attributes, use an empty value.

`IGRAPH_ATTRIBUTE_COMBINE_DEFAULT`: Use the default way to combine attributes (decided by the attribute handler implementation).

`IGRAPH_ATTRIBUTE_COMBINE_FUNCTION`: Supply your own function to combine attributes.

`IGRAPH_ATTRIBUTE_COMBINE_SUM`: Take the sum of the attributes.

`IGRAPH_ATTRIBUTE_COMBINE_PROD`: Take the product of the attributes.

`IGRAPH_ATTRIBUTE_COMBINE_MIN`: Take the minimum attribute.

`IGRAPH_ATTRIBUTE_COMBINE_MAX`: Take the maximum attribute.

`IGRAPH_ATTRIBUTE_COMBINE_RANDOM`: Take a random attribute.

`IGRAPH_ATTRIBUTE_COMBINE_FIRST`: Take the first attribute.

`IGRAPH_ATTRIBUTE_COMBINE_LAST`: Take the last attribute.

`IGRAPH_ATTRIBUTE_COMBINE_MEAN`: Take the mean of the attributes.

`IGRAPH_ATTRIBUTE_COMBINE_MEDIAN`: Take the median of the attributes.

`IGRAPH_ATTRIBUTE_COMBINE_CONCAT`: Concatenate the attributes.

`igraph_attribute_combination` — Initialize attribute combination list and add records.

```
igraph_error_t igraph_attribute_combination(
        igraph_attribute_combination_t *comb, ...);
```

Arguments:

comb: The uninitialized attribute combination list.

...: A list of 'name, type[, func]', where:

name: The name of the attribute. If the name already exists the attribute combination record will be replaced. Use NULL to add a default combination record for all atributes not in the list.

type: The type of the attribute combination. See `igraph_attribute_combination_type_t` for the options.

func: Function to be used if *type* is `IGRAPH_ATTRIBUTE_COMBINE_FUNCTION`. The list is closed by setting the name to `IGRAPH_NO_MORE_ATTRIBUTES`.

Returns:

Error code.

Time complexity: O(n^2), where n is the number attribute combinations records to add.

Example 10.1. File `examples/simple/igraph_attribute_combination.c`

Accessing attributes from C

There is an experimental attribute handler that can be used from C code. In this section we show how this works. This attribute handler is by default not attached (the default is no attribute handler), so we first need to attach it:

```
igraph_set_attribute_table(&igraph_cattribute_table);
```

Now the attribute functions are available. Please note that the attribute handler must be attached before you call any other igraph functions, otherwise you might end up with graphs without attributes and an active attribute handler, which might cause unexpected program behaviour. The rule is that you attach the attribute handler in the beginning of your `main()` and never touch it again. Detaching the attribute handler might lead to memory leaks.

It is not currently possible to have attribute handlers on a per-graph basis. All graphs in an application must be managed with the same attribute handler. This also applies to the default case when there is no attribute handler at all.

The C attribute handler supports attaching real numbers, boolean values and character strings as attributes. No vector values are allowed. For example, vertices have a `name` attribute holding a single string value for each vertex, but it is not possible to have a `coords` attribute which is a vector of numbers per vertex.

The functions documented in this section are specific to the C attribute handler. Code using these functions will not function when a different attribute handler is attached.

Example 10.2. File `examples/simple/cattributes.c`

Example 10.3. File `examples/simple/cattributes2.c`

Example 10.4. File `examples/simple/cattributes3.c`

Example 10.5. File `examples/simple/cattributes4.c`

Query attributes

`igraph_cattribute_list` — List all attributes.

```
igraph_error_t igraph_cattribute_list(const igraph_t *graph,
                            igraph_strvector_t *gnames, igraph_vector_int_t *gty
                            igraph_strvector_t *vnames, igraph_vector_int_t *vty
                            igraph_strvector_t *enames, igraph_vector_int_t *ety
```

See `igraph_attribute_type_t` for the various attribute types.

Arguments:

`graph`: The input graph.

`gnames`: String vector, the names of the graph attributes.

`gtypes`: Numeric vector, the types of the graph attributes.

`vnames`: String vector, the names of the vertex attributes.

`vtypes`: Numeric vector, the types of the vertex attributes.

`enames`: String vector, the names of the edge attributes.

`etypes`: Numeric vector, the types of the edge attributes.

Returns:

Error code.

Naturally, the string vector with the attribute names and the numeric vector with the attribute types are in the right order, i.e. the first name corresponds to the first type, etc. Time complexity: O(Ag+Av +Ae), the number of all attributes.

`igraph_cattribute_has_attr` — Checks whether a (graph, vertex or edge) attribute exists.

```
igraph_bool_t igraph_cattribute_has_attr(const igraph_t *graph,
        igraph_attribute_elemtype_t type,
        const char *name);
```

Arguments:

`graph`: The graph.

`type`: The type of the attribute, `IGRAPH_ATTRIBUTE_GRAPH`, `IGRAPH_ATTRIBUTE_VERTEX` or `IGRAPH_ATTRIBUTE_EDGE`.

name: Character constant, the name of the attribute.

Returns:

Boolean value, `true` if the attribute exists, `false` otherwise.

Time complexity: O(A), the number of (graph, vertex or edge) attributes, assuming attribute names are not too long.

`igraph_cattribute_GAN` — Query a numeric graph attribute.

```
igraph_real_t igraph_cattribute_GAN(const igraph_t *graph, const char *name);
```

Returns the value of the given numeric graph attribute. If the attribute does not exist, a warning is issued and NaN is returned.

Arguments:

graph: The input graph.

name: The name of the attribute to query.

Returns:

The value of the attribute.

See also:

`GAN` for a simpler interface.

Time complexity: O(Ag), the number of graph attributes.

`GAN` — Query a numeric graph attribute.

```
#define GAN(graph,n)
```

This is shorthand for `igraph_cattribute_GAN()`.

Arguments:

graph: The graph.

n: The name of the attribute.

Returns:

The value of the attribute.

`igraph_cattribute_GAB` — Query a boolean graph attribute.

```
igraph_bool_t igraph_cattribute_GAB(const igraph_t *graph, const char *name);
```

Returns the value of the given boolean graph attribute. If the attribute does not exist, a warning is issued and false is returned.

Arguments:

graph: The input graph.

name: The name of the attribute to query.

Returns:

The value of the attribute.

See also:

GAB for a simpler interface.

Time complexity: O(Ag), the number of graph attributes.

GAB — Query a boolean graph attribute.

```
#define GAB(graph,n)
```

This is shorthand for `igraph_cattribute_GAB()`.

Arguments:

graph: The graph.

n: The name of the attribute.

Returns:

The value of the attribute.

igraph_cattribute_GAS — Query a string graph attribute.

```
const char *igraph_cattribute_GAS(const igraph_t *graph, const char *name);
```

Returns a const pointer to the string graph attribute specified in *name*. The value must not be modified.
If the attribute does not exist, a warning is issued and an empty string is returned.

Arguments:

graph: The input graph.

name: The name of the attribute to query.

Returns:

The value of the attribute.

See also:

GAS for a simpler interface.

Time complexity: O(Ag), the number of graph attributes.

GAS — Query a string graph attribute.

```
#define GAS(graph,n)
```

This is shorthand for `igraph_cattribute_GAS()`.

Arguments:

graph: The graph.

n: The name of the attribute.

Returns:

The value of the attribute.

igraph_cattribute_VAN — Query a numeric vertex attribute.

```
igraph_real_t igraph_cattribute_VAN(const igraph_t *graph, const char *name,
                                    igraph_int_t vid);
```

If the attribute does not exist, a warning is issued and NaN is returned. See `igraph_cattribute_VANV()` for an error-checked version.

Arguments:

graph: The input graph.

name: The name of the attribute.

vid: The id of the queried vertex.

Returns:

The value of the attribute.

See also:

`VAN` macro for a simpler interface.

Time complexity: O(Av), the number of vertex attributes.

VAN — Query a numeric vertex attribute.

```
#define VAN(graph,n,v)
```

This is shorthand for `igraph_cattribute_VAN()`.

Arguments:

graph: The graph.

n: The name of the attribute.

v: The id of the vertex.

Returns:

The value of the attribute.

`igraph_cattribute_VANV` — Query a numeric vertex attribute for many vertices.

```
igraph_error_t igraph_cattribute_VANV(const igraph_t *graph, const char *name,
                            igraph_vs_t vids, igraph_vector_t *result);
```

Arguments:

`graph`: The input graph.

`name`: The name of the attribute.

`vids`: The vertices to query.

`result`: Pointer to an initialized vector, the result is stored here. It will be resized, if needed.

Returns:

Error code.

Time complexity: O(v), where v is the number of vertices in 'vids'.

`VANV` — Query a numeric vertex attribute for all vertices.

```
#define VANV(graph,n,vec)
```

This is a shorthand for `igraph_cattribute_VANV()`.

Arguments:

`graph`: The graph.

`n`: The name of the attribute.

`vec`: Pointer to an initialized vector, the result is stored here. It will be resized, if needed.

Returns:

Error code.

`igraph_cattribute_VAB` — Query a boolean vertex attribute.

```
igraph_bool_t igraph_cattribute_VAB(const igraph_t *graph, const char *name,
                                    igraph_int_t vid);
```

If the vertex attribute does not exist, a warning is issued and false is returned. See `igraph_cattribute_VABV()` for an error-checked version.

Arguments:

graph: The input graph.

name: The name of the attribute.

vid: The id of the queried vertex.

Returns:

The value of the attribute.

See also:

VAB macro for a simpler interface.

Time complexity: O(Av), the number of vertex attributes.

VAB — Query a boolean vertex attribute.

```
#define VAB(graph,n,v)
```

This is shorthand for `igraph_cattribute_VAB()`.

Arguments:

graph: The graph.

n: The name of the attribute.

v: The id of the vertex.

Returns:

The value of the attribute.

igraph_cattribute_VABV — Query a boolean vertex attribute for many vertices.

```
igraph_error_t igraph_cattribute_VABV(const igraph_t *graph, const char *name,
                            igraph_vs_t vids, igraph_vector_bool_t *result);
```

Arguments:

graph: The input graph.

name: The name of the attribute.

vids: The vertices to query.

result: Pointer to an initialized boolean vector, the result is stored here. It will be resized, if needed.

Returns:

Error code.

Time complexity: O(v), where v is the number of vertices in 'vids'.

VABV — Query a boolean vertex attribute for all vertices.

```
#define VABV(graph,n,vec)
```

This is a shorthand for `igraph_cattribute_VABV()`.

Arguments:

graph: The graph.

n: The name of the attribute.

vec: Pointer to an initialized boolean vector, the result is stored here. It will be resized, if needed.

Returns:

Error code.

igraph_cattribute_VAS — Query a string vertex attribute.

```
const char *igraph_cattribute_VAS(const igraph_t *graph, const char *name,
                                  igraph_int_t vid);
```

Returns a const pointer to the string vertex attribute specified in *name*. The value must not be modified. If the vertex attribute does not exist, a warning is issued and an empty string is returned. See `igraph_cattribute_VASV()` for an error-checked version.

Arguments:

graph: The input graph.

name: The name of the attribute.

vid: The id of the queried vertex.

Returns:

The value of the attribute.

See also:

The macro VAS for a simpler interface.

Time complexity: O(Av), the number of vertex attributes.

VAS — Query a string vertex attribute.

```
#define VAS(graph,n,v)
```

This is shorthand for `igraph_cattribute_VAS()`.

Arguments:

graph: The graph.

n: The name of the attribute.

v: The id of the vertex.

Returns:

The value of the attribute.

igraph_cattribute_VASV — Query a string vertex attribute for many vertices.

```
igraph_error_t igraph_cattribute_VASV(const igraph_t *graph, const char *name,
                            igraph_vs_t vids, igraph_strvector_t *result);
```

Arguments:

graph: The input graph.

name: The name of the attribute.

vids: The vertices to query.

result: Pointer to an initialized string vector, the result is stored here. It will be resized, if needed.

Returns:

Error code.

Time complexity: O(v), where v is the number of vertices in 'vids'. (We assume that the string attributes have a bounded length.)

VASV — Query a string vertex attribute for all vertices.

```
#define VASV(graph,n,vec)
```

This is a shorthand for `igraph_cattribute_VASV()`.

Arguments:

graph: The graph.

n: The name of the attribute.

vec: Pointer to an initialized string vector, the result is stored here. It will be resized, if needed.

Returns:

Error code.

igraph_cattribute_EAN — Query a numeric edge attribute.

```
igraph_real_t igraph_cattribute_EAN(const igraph_t *graph, const char *name,
                                    igraph_int_t eid);
```

If the attribute does not exist, a warning is issued and NaN is returned. See `igraph_cattribute_EANV()` for an error-checked version.

Arguments:

graph: The input graph.

name: The name of the attribute.

eid: The id of the queried edge.

Returns:

The value of the attribute.

See also:

`EAN` for an easier interface.

Time complexity: O(Ae), the number of edge attributes.

EAN — Query a numeric edge attribute.

```
#define EAN(graph,n,e)
```

This is shorthand for `igraph_cattribute_EAN()`.

Arguments:

graph: The graph.

n: The name of the attribute.

e: The id of the edge.

Returns:

The value of the attribute.

igraph_cattribute_EANV — Query a numeric edge attribute for many edges.

```
igraph_error_t igraph_cattribute_EANV(const igraph_t *graph, const char *name,
                            igraph_es_t eids, igraph_vector_t *result);
```

Arguments:

graph: The input graph.

name: The name of the attribute.

eids: The edges to query.

result: Pointer to an initialized vector, the result is stored here. It will be resized, if needed.

Returns:

Error code.

Time complexity: O(e), where e is the number of edges in 'eids'.

EANV — Query a numeric edge attribute for all edges.

```
#define EANV(graph,n,vec)
```

This is a shorthand for `igraph_cattribute_EANV()`.

Arguments:

graph: The graph.

n: The name of the attribute.

vec: Pointer to an initialized vector, the result is stored here. It will be resized, if needed.

Returns:

Error code.

igraph_cattribute_EAB — Query a boolean edge attribute.

```
igraph_bool_t igraph_cattribute_EAB(const igraph_t *graph, const char *name,
                                    igraph_int_t eid);
```

If the edge attribute does not exist, a warning is issued and false is returned. See `igraph_cattribute_EABV()` for an error-checked version.

Arguments:

graph: The input graph.

name: The name of the attribute.

eid: The id of the queried edge.

Returns:

The value of the attribute.

See also:

EAB for an easier interface.

Time complexity: O(Ae), the number of edge attributes.

EAB — Query a boolean edge attribute.

```
#define EAB(graph,n,e)
```

This is shorthand for `igraph_cattribute_EAB()`.

Arguments:

graph: The graph.

n: The name of the attribute.

e: The id of the edge.

Returns:

The value of the attribute.

`igraph_cattribute_EABV` — Query a boolean edge attribute for many edges.

```
igraph_error_t igraph_cattribute_EABV(const igraph_t *graph, const char *name,
                            igraph_es_t eids, igraph_vector_bool_t *result);
```

Arguments:

graph: The input graph.

name: The name of the attribute.

eids: The edges to query.

result: Pointer to an initialized boolean vector, the result is stored here. It will be resized, if needed.

Returns:

Error code.

Time complexity: O(e), where e is the number of edges in 'eids'.

EABV — Query a boolean edge attribute for all edges.

```
#define EABV(graph,n,vec)
```

This is a shorthand for `igraph_cattribute_EABV()`.

Arguments:

graph: The graph.

n: The name of the attribute.

vec: Pointer to an initialized vector, the result is stored here. It will be resized, if needed.

Returns:

Error code.

`igraph_cattribute_EAS` — Query a string edge attribute.

```
const char *igraph_cattribute_EAS(const igraph_t *graph, const char *name,
                                  igraph_int_t eid);
```

Returns a const pointer to the string edge attribute specified in *`name`*. The value must not be modified. If the edge attribute does not exist, a warning is issued and an empty string is returned. See `igraph_cattribute_EASV()` for an error-checked version.

Arguments:

`graph`: The input graph.

`name`: The name of the attribute.

`eid`: The id of the queried edge.

Returns:

The value of the attribute.

\se EAS if you want to type less. Time complexity: O(Ae), the number of edge attributes.

EAS — Query a string edge attribute.

```
#define EAS(graph,n,e)
```

This is shorthand for `igraph_cattribute_EAS()`.

Arguments:

`graph`: The graph.

`n`: The name of the attribute.

`e`: The id of the edge.

Returns:

The value of the attribute.

`igraph_cattribute_EASV` — Query a string edge attribute for many edges.

```
igraph_error_t igraph_cattribute_EASV(const igraph_t *graph, const char *name,
                            igraph_es_t eids, igraph_strvector_t *result);
```

Arguments:

`graph`: The input graph.

`name`: The name of the attribute.

`eids`: The edges to query.

result: Pointer to an initialized string vector, the result is stored here. It will be resized, if needed.

Returns:

Error code.

Time complexity: O(e), where e is the number of edges in 'eids'. (We assume that the string attributes have a bounded length.)

EASV — Query a string edge attribute for all edges.

```
#define EASV(graph,n,vec)
```

This is a shorthand for `igraph_cattribute_EASV()`.

Arguments:

graph: The graph.

n: The name of the attribute.

vec: Pointer to an initialized string vector, the result is stored here. It will be resized, if needed.

Returns:

Error code.

Set attributes

igraph_cattribute_GAN_set — Set a numeric graph attribute.

```
igraph_error_t igraph_cattribute_GAN_set(igraph_t *graph, const char *name,
                                         igraph_real_t value);
```

Arguments:

graph: The graph.

name: Name of the graph attribute. If there is no such attribute yet, then it will be added.

value: The (new) value of the graph attribute.

Returns:

Error code.

\se `SETGAN` if you want to type less. Time complexity: O(1).

SETGAN — Set a numeric graph attribute

```
#define SETGAN(graph,n,value)
```

This is a shorthand for `igraph_cattribute_GAN_set()`.

Arguments:

graph: The graph.

n: The name of the attribute.

value: The new value of the attribute.

Returns:

Error code.

igraph_cattribute_GAB_set — Set a boolean graph attribute.

```
igraph_error_t igraph_cattribute_GAB_set(igraph_t *graph, const char *name,
                                         igraph_bool_t value);
```

Arguments:

graph: The graph.

name: Name of the graph attribute. If there is no such attribute yet, then it will be added.

value: The (new) value of the graph attribute.

Returns:

Error code.

\se SETGAN if you want to type less. Time complexity: O(1).

SETGAB — Set a boolean graph attribute

```
#define SETGAB(graph,n,value)
```

This is a shorthand for `igraph_cattribute_GAB_set()`.

Arguments:

graph: The graph.

n: The name of the attribute.

value: The new value of the attribute.

Returns:

Error code.

igraph_cattribute_GAS_set — Set a string graph attribute.

```
igraph_error_t igraph_cattribute_GAS_set(igraph_t *graph, const char *name,
                                         const char *value);
```

Arguments:

graph: The graph.

name: Name of the graph attribute. If there is no such attribute yet, then it will be added.

value: The (new) value of the graph attribute. It will be copied.

Returns:

Error code.

\se SETGAS if you want to type less. Time complexity: O(1).

SETGAS — Set a string graph attribute

```
#define SETGAS(graph,n,value)
```

This is a shorthand for `igraph_cattribute_GAS_set()`.

Arguments:

graph: The graph.

n: The name of the attribute.

value: The new value of the attribute.

Returns:

Error code.

igraph_cattribute_VAN_set — Set a numeric vertex attribute.

```
igraph_error_t igraph_cattribute_VAN_set(igraph_t *graph, const char *name,
                                         igraph_int_t vid, igraph_real_t value);
```

The attribute will be added if not present already. If present it will be overwritten. The same *value* is set for all vertices included in *vid*.

Arguments:

graph: The graph.

name: Name of the attribute.

vid: Vertices for which to set the attribute.

value: The (new) value of the attribute.

Returns:

Error code.

See also:

SETVAN for a simpler way.

Time complexity: O(n), the number of vertices if the attribute is new, O(|vid|) otherwise.

SETVAN — Set a numeric vertex attribute

```
#define SETVAN(graph,n,vid,value)
```

This is a shorthand for `igraph_cattribute_VAN_set()`.

Arguments:

graph: The graph.

n: The name of the attribute.

vid: Ids of the vertices to set.

value: The new value of the attribute.

Returns:

Error code.

igraph_cattribute_VAB_set — Set a boolean vertex attribute.

```
igraph_error_t igraph_cattribute_VAB_set(igraph_t *graph, const char *name,
                                         igraph_int_t vid, igraph_bool_t value);
```

The attribute will be added if not present already. If present it will be overwritten. The same *value* is set for all vertices included in *vid*.

Arguments:

graph: The graph.

name: Name of the attribute.

vid: Vertices for which to set the attribute.

value: The (new) value of the attribute.

Returns:

Error code.

See also:

SETVAB for a simpler way.

Time complexity: O(n), the number of vertices if the attribute is new, O(|vid|) otherwise.

SETVAB — Set a boolean vertex attribute

```
#define SETVAB(graph,n,vid,value)
```

This is a shorthand for `igraph_cattribute_VAB_set()`.

Arguments:

graph: The graph.

n: The name of the attribute.

vid: Ids of the vertices to set.

value: The new value of the attribute.

Returns:

Error code.

`igraph_cattribute_VAS_set` — Set a string vertex attribute.

```
igraph_error_t igraph_cattribute_VAS_set(igraph_t *graph, const char *name,
                                         igraph_int_t vid, const char *value);
```

The attribute will be added if not present already. If present it will be overwritten. The same *value* is set for all vertices included in *vid*.

Arguments:

graph: The graph.

name: Name of the attribute.

vid: Vertices for which to set the attribute.

value: The (new) value of the attribute.

Returns:

Error code.

See also:

SETVAS for a simpler way.

Time complexity: O(n*l), n is the number of vertices, l is the length of the string to set. If the attribute if not new then only O(|vid|*l).

SETVAS — Set a string vertex attribute

```
#define SETVAS(graph,n,vid,value)
```

This is a shorthand for `igraph_cattribute_VAS_set()`.

Arguments:

graph: The graph.

n: The name of the attribute.

vid: Ids of the vertices to set.

value: The new value of the attribute.

Returns:

Error code.

igraph_cattribute_EAN_set — Set a numeric edge attribute.

```
igraph_error_t igraph_cattribute_EAN_set(igraph_t *graph, const char *name,
                                         igraph_int_t eid, igraph_real_t value);
```

The attribute will be added if not present already. If present it will be overwritten. The same *value* is set for all edges included in *vid*.

Arguments:

graph: The graph.

name: Name of the attribute.

eid: Edges for which to set the attribute.

value: The (new) value of the attribute.

Returns:

Error code.

See also:

SETEAN for a simpler way.

Time complexity: O(e), the number of edges if the attribute is new, O(|eid|) otherwise.

SETEAN — Set a numeric edge attribute

```
#define SETEAN(graph,n,eid,value)
```

This is a shorthand for `igraph_cattribute_EAN_set()`.

Arguments:

graph: The graph.

n: The name of the attribute.

eid: Ids of the edges to set.

value: The new value of the attribute.

Returns:

Error code.

`igraph_cattribute_EAB_set` — Set a boolean edge attribute.

```
igraph_error_t igraph_cattribute_EAB_set(igraph_t *graph, const char *name,
                                         igraph_int_t eid, igraph_bool_t value);
```

The attribute will be added if not present already. If present it will be overwritten. The same *value* is set for all edges included in *vid*.

Arguments:

graph: The graph.

name: Name of the attribute.

eid: Edges for which to set the attribute.

value: The (new) value of the attribute.

Returns:

Error code.

See also:

SETEAB for a simpler way.

Time complexity: O(e), the number of edges if the attribute is new, O(|eid|) otherwise.

SETEAB — Set a boolean edge attribute

```
#define SETEAB(graph,n,eid,value)
```

This is a shorthand for `igraph_cattribute_EAB_set()`.

Arguments:

graph: The graph.

n: The name of the attribute.

eid: Ids of the edges to set.

value: The new value of the attribute.

Returns:

Error code.

`igraph_cattribute_EAS_set` — Set a string edge attribute.

```
igraph_error_t igraph_cattribute_EAS_set(igraph_t *graph, const char *name,
                                         igraph_int_t eid, const char *value);
```

The attribute will be added if not present already. If present it will be overwritten. The same *value* is set for all edges included in *vid*.

Arguments:

graph: The graph.

name: Name of the attribute.

eid: Edges for which to set the attribute.

value: The (new) value of the attribute.

Returns:

Error code.

See also:

SETEAS for a simpler way.

Time complexity: O(e*l), n is the number of edges, l is the length of the string to set. If the attribute if not new then only O(|eid|*l).

SETEAS — Set a string edge attribute

```
#define SETEAS(graph,n,eid,value)
```

This is a shorthand for `igraph_cattribute_EAS_set()`.

Arguments:

graph: The graph.

n: The name of the attribute.

eid: Ids of the edges to set.

value: The new value of the attribute.

Returns:

Error code.

`igraph_cattribute_VAN_setv` — Set a numeric vertex attribute for all vertices.

```
igraph_error_t igraph_cattribute_VAN_setv(igraph_t *graph, const char *name,
                                          const igraph_vector_t *v);
```

The attribute will be added if not present yet.

Arguments:

graph: The graph.

name: Name of the attribute.

v: The new attribute values. The length of this vector must match the number of vertices.

Returns:

Error code.

See also:

SETVANV for a simpler way.

Time complexity: O(n), the number of vertices.

SETVANV — Set a numeric vertex attribute for all vertices

```
#define SETVANV(graph,n,v)
```

This is a shorthand for `igraph_cattribute_VAN_setv()`.

Arguments:

graph: The graph.

n: The name of the attribute.

v: Vector containing the new values of the attributes.

Returns:

Error code.

igraph_cattribute_VAB_setv — Set a boolean vertex attribute for all vertices.

```
igraph_error_t igraph_cattribute_VAB_setv(igraph_t *graph, const char *name,
                                          const igraph_vector_bool_t *v);
```

The attribute will be added if not present yet.

Arguments:

graph: The graph.

name: Name of the attribute.

v: The new attribute values. The length of this boolean vector must match the number of vertices.

Returns:

Error code.

See also:

SETVANV for a simpler way.

Time complexity: O(n), the number of vertices.

SETVABV — Set a boolean vertex attribute for all vertices

```
#define SETVABV(graph,n,v)
```

This is a shorthand for `igraph_cattribute_VAB_setv()`.

Arguments:

graph: The graph.

n: The name of the attribute.

v: Vector containing the new values of the attributes.

Returns:

Error code.

igraph_cattribute_VAS_setv — Set a string vertex attribute for all vertices.

```
igraph_error_t igraph_cattribute_VAS_setv(igraph_t *graph, const char *name,
                                          const igraph_strvector_t *sv);
```

The attribute will be added if not present yet.

Arguments:

graph: The graph.

name: Name of the attribute.

sv: String vector, the new attribute values. The length of this vector must match the number of vertices.

Returns:

Error code.

See also:

SETVASV for a simpler way.

Time complexity: O(n+l), n is the number of vertices, l is the total length of the strings.

SETVASV — Set a string vertex attribute for all vertices

```
#define SETVASV(graph,n,v)
```

This is a shorthand for `igraph_cattribute_VAS_setv()`.

Arguments:

graph: The graph.

n: The name of the attribute.

v: Vector containing the new values of the attributes.

Returns:

Error code.

`igraph_cattribute_EAN_setv` — Set a numeric edge attribute for all edges.

```
igraph_error_t igraph_cattribute_EAN_setv(igraph_t *graph, const char *name,
                                          const igraph_vector_t *v);
```

The attribute will be added if not present yet.

Arguments:

graph: The graph.

name: Name of the attribute.

v: The new attribute values. The length of this vector must match the number of edges.

Returns:

Error code.

See also:

`SETEANV` for a simpler way.

Time complexity: O(e), the number of edges.

SETEANV — Set a numeric edge attribute for all edges

```
#define SETEANV(graph,n,v)
```

This is a shorthand for `igraph_cattribute_EAN_setv()`.

Arguments:

graph: The graph.

n: The name of the attribute.

v: Vector containing the new values of the attributes.

`igraph_cattribute_EAB_setv` — Set a boolean edge attribute for all edges.

```
igraph_error_t igraph_cattribute_EAB_setv(igraph_t *graph, const char *name,
```

```
                                    const igraph_vector_bool_t *v);
```

The attribute will be added if not present yet.

Arguments:

graph: The graph.

name: Name of the attribute.

v: The new attribute values. The length of this vector must match the number of edges.

Returns:

Error code.

See also:

SETEABV for a simpler way.

Time complexity: O(e), the number of edges.

SETEABV — Set a boolean edge attribute for all edges

```
#define SETEABV(graph,n,v)
```

This is a shorthand for `igraph_cattribute_EAB_setv()`.

Arguments:

graph: The graph.

n: The name of the attribute.

v: Vector containing the new values of the attributes.

`igraph_cattribute_EAS_setv` — Set a string edge attribute for all edges.

```
igraph_error_t igraph_cattribute_EAS_setv(igraph_t *graph, const char *name,
                                    const igraph_strvector_t *sv);
```

The attribute will be added if not present yet.

Arguments:

graph: The graph.

name: Name of the attribute.

sv: String vector, the new attribute values. The length of this vector must match the number of edges.

Returns:

Error code.

See also:

SETEASV for a simpler way.

Time complexity: O(e+l), e is the number of edges, l is the total length of the strings.

SETEASV — Set a string edge attribute for all edges

```
#define SETEASV(graph,n,v)
```

This is a shorthand for `igraph_cattribute_EAS_setv()`.

Arguments:

graph: The graph.

n: The name of the attribute.

v: Vector containing the new values of the attributes.

Remove attributes

igraph_cattribute_remove_g — Remove a graph attribute.

```
void igraph_cattribute_remove_g(igraph_t *graph, const char *name);
```

Arguments:

graph: The graph object.

name: Name of the graph attribute to remove.

See also:

DELGA for a simpler way.

DELGA — Remove a graph attribute.

```
#define DELGA(graph,n)
```

A shorthand for `igraph_cattribute_remove_g()`.

Arguments:

graph: The graph.

n: The name of the attribute to remove.

igraph_cattribute_remove_v — Remove a vertex attribute.

```
void igraph_cattribute_remove_v(igraph_t *graph, const char *name);
```

Arguments:

graph: The graph object.

name: Name of the vertex attribute to remove.

See also:

DELVA for a simpler way.

DELVA — Remove a vertex attribute.

```
#define DELVA(graph,n)
```

A shorthand for `igraph_cattribute_remove_v()`.

Arguments:

graph: The graph.

n: The name of the attribute to remove.

igraph_cattribute_remove_e — Remove an edge attribute.

```
void igraph_cattribute_remove_e(igraph_t *graph, const char *name);
```

Arguments:

graph: The graph object.

name: Name of the edge attribute to remove.

See also:

DELEA for a simpler way.

DELEA — Remove an edge attribute.

```
#define DELEA(graph,n)
```

A shorthand for `igraph_cattribute_remove_e()`.

Arguments:

graph: The graph.

n: The name of the attribute to remove.

igraph_cattribute_remove_all — Remove all graph/vertex/edge attributes.

```
void igraph_cattribute_remove_all(igraph_t *graph, igraph_bool_t g,
                                  igraph_bool_t v, igraph_bool_t e);
```

Arguments:

graph: The graph object.

g: Boolean, whether to remove graph attributes.

v: Boolean, whether to remove vertex attributes.

e: Boolean, whether to remove edge attributes.

See also:

`DELGAS`, `DELVAS`, `DELEAS`, `DELALL` for simpler ways.

`DELGAS` — Remove all graph attributes.

```
#define DELGAS(graph)
```

Calls `igraph_cattribute_remove_all()`.

Arguments:

graph: The graph.

`DELVAS` — Remove all vertex attributes.

```
#define DELVAS(graph)
```

Calls `igraph_cattribute_remove_all()`.

Arguments:

graph: The graph.

`DELEAS` — Remove all edge attributes.

```
#define DELEAS(graph)
```

Calls `igraph_cattribute_remove_all()`.

Arguments:

graph: The graph.

`DELALL` — Remove all attributes.

```
#define DELALL(graph)
```

All graph, vertex and edges attributes will be removed. Calls `igraph_cattribute_remove_all()`.

Arguments:

graph: The graph.

Custom attribute combination functions

The C attribute handler supports combining the attributes of multiple vertices of edges into a single attribute during a vertex or edge contraction operation via a user-defined function. This is achieved by setting the type of the attribute combination to `IGRAPH_ATTRIBUTE_COMBINE_FUNCTION` and passing in a pointer to the custom combination function when specifying attribute combinations in `igraph_attribute_combination()` or `igraph_attribute_combination_add()`. For the C attribute handler, the signature of the function depends on the type of the underlying attribute. For numeric attributes, use:

```
igraph_error_t function(const igraph_vector_t *input, igraph_real_t *output);
```

where *input* will receive a vector containing the value of the attribute for all the vertices or edges being combined, and *output* must be filled by the function to the combined value. Similarly, for Boolean attributes, the function takes a boolean vector in *input* and must return the combined Boolean value in *output*:

```
igraph_error_t function(const igraph_vector_bool_t *input, igraph_bool_t *outpu
```

For string attributes, the signature is slightly different:

```
igraph_error_t function(const igraph_strvector_t *input, char **output);
```

In case of strings, all strings in the input vector are *owned* by igraph and must not be modified or freed in the combination handler. The string returned to the caller in *output* remains owned by the caller; igraph will make a copy it and store the copy in the appropriate part of the data structure holding the vertex or edge attributes.

Chapter 11. Deterministic graph generators

About generators

Most functions that create graphs in a deterministic manner are documented here. See also stochastic generators, spatial graph generators, bipartite graph generators, and operators that transform graphs.

Basic graph creation

`igraph_create` — Creates a graph with the specified edges.

```
igraph_error_t igraph_create(igraph_t *graph, const igraph_vector_int_t *edges,
                             igraph_int_t n, igraph_bool_t directed);
```

Arguments:

graph: An uninitialized graph object.

edges: The edges to add, the first two elements are the first edge, etc.

n: The number of vertices in the graph, if smaller or equal to the highest vertex ID in the *edges* vector it will be increased automatically. So it is safe to give 0 here.

directed: Boolean, whether to create a directed graph or not. If yes, then the first edge points from the first vertex ID in *edges* to the second, etc.

Returns:

Error code: `IGRAPH_EINVAL`: invalid edges vector (odd number of vertices). `IGRAPH_EINVVID`: invalid (negative) vertex ID.

Time complexity: O(|V|+|E|), |V| is the number of vertices, |E| the number of edges in the graph.

Example 11.1. File `examples/simple/igraph_create.c`

`igraph_small` — Shorthand to create a small graph, giving the edges as arguments.

```
igraph_error_t igraph_small(igraph_t *graph, igraph_int_t n, igraph_bool_t direc
                            int first, ...);
```

This function is handy when a relatively small graph needs to be created. Instead of giving the edges as a vector, they are given simply as arguments and a `-1` needs to be given after the last meaningful edge argument.

This function is intended to be used with vertex IDs that are entered as literal integers. If you use a variable instead of a literal, make sure that it is of type int, as this is the type that this function assumes

for all variadic arguments. Using a different integer type is undefined behaviour and likely to cause platform-specific issues.

Arguments:

graph: Pointer to an uninitialized graph object. The result will be stored here.

n: The number of vertices in the graph; a non-negative integer.

directed: Boolean constant; gives whether the graph should be directed. Supported values are:

`IGRAPH_DIRECTED` The graph to be created will be *directed.*

`IGRAPH_UNDIRECTED` The graph to be created will be *undirected.*

...: The additional arguments giving the edges of the graph, and *must* be of type int. Don't forget to supply an additional `-1` after the last (meaningful) argument. The *first* parameter is present for technical reasons and represents the first variadic argument.

Returns:

Error code.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges in the graph to create.

Example 11.2. File `examples/simple/igraph_small.c`

Graphs from adjacency matrices and adjacency lists

These functions create graphs from weighted or unweighted adjacency matrices, or an adjacency list.

`igraph_adjacency` — Creates a graph from an adjacency matrix.

```
igraph_error_t igraph_adjacency(
    igraph_t *graph, const igraph_matrix_t *adjmatrix, igraph_adjacency_t mode,
    igraph_loops_t loops
);
```

The order of the vertices in the matrix is preserved, i.e. the vertex corresponding to the first row/column will be vertex with id 0, the next row is for vertex 1, etc. No guarantees are given about the ordering of edges.

Arguments:

graph: Pointer to an uninitialized graph object.

adjmatrix: The adjacency matrix. How it is interpreted depends on the *mode* argument.

mode: Constant to specify how the given matrix is interpreted as an adjacency matrix. Possible values (A(i,j) is the element in row i and column j in the adjacency matrix *adjmatrix*):

`IGRAPH_ADJ_DIRECTED` The graph will be directed and an element gives the number of edges between two vertices.

`IGRAPH_ADJ_UNDIRECTED` The graph will be undirected and an element gives the number of edges between two vertices. If the input matrix is not symmetric, an error is thrown.

`IGRAPH_ADJ_MAX` An undirected graph will be created and the number of edges between vertices i and j is max(A(i,j), A(j,i)).

`IGRAPH_ADJ_MIN` An undirected graph will be created with min(A(i,j), A(j,i)) edges between vertices i and j.

`IGRAPH_ADJ_PLUS` An undirected graph will be created with A(i,j)+A(j,i) edges between vertices i and j.

`IGRAPH_ADJ_UPPER` An undirected graph will be created. Only the upper right triangle (including the diagonal) is used for the number of edges.

`IGRAPH_ADJ_LOWER` An undirected graph will be created. Only the lower left triangle (including the diagonal) is used for the number of edges.

`loops`: Constant of type `igraph_loops_t` to specify how the diagonal of the matrix should be treated when creating loop edges. Ignored for modes `IGRAPH_ADJ_DIRECTED`, `IGRAPH_ADJ_UPPER` and `IGRAPH_ADJ_LOWER`.

`IGRAPH_NO_LOOPS` Ignore the diagonal of the input matrix and do not create loops.

`IGRAPH_LOOPS_ONCE` Treat the diagonal entries as the number of loop edges incident on the corresponding vertex.

`IGRAPH_LOOPS_TWICE` Treat the diagonal entries as *twice* the number of loop edges incident on the corresponding vertex. Odd numbers in the diagonal will return an error code.

Returns:

Error code, `IGRAPH_EINVAL`: Non-square adjacency matrix, negative entry in adjacency matrix, or an odd number was found in the diagonal with `IGRAPH_LOOPS_TWICE`

Time complexity: O(|V||V|), |V| is the number of vertices in the graph.

`igraph_weighted_adjacency` — Creates a graph from a weighted adjacency matrix.

```
igraph_error_t igraph_weighted_adjacency(
    igraph_t *graph, const igraph_matrix_t *adjmatrix, igraph_adjacency_t mode,
    igraph_vector_t *weights, igraph_loops_t loops
);
```

The order of the vertices in the matrix is preserved, i.e. the vertex corresponding to the first row/column will be vertex with id 0, the next row is for vertex 1, etc. No guarantees are given for the ordering of edges.

Arguments:

graph: Pointer to an uninitialized graph object.

adjmatrix: The weighted adjacency matrix. How it is interpreted depends on the *mode* argument. The common feature is that edges with zero weights are considered nonexistent (however, negative weights are permitted).

mode: Constant to specify how the given matrix is interpreted as an adjacency matrix. Possible values (A(i,j) is the element in row i and column j in the adjacency matrix *adjmatrix*):

IGRAPH_ADJ_DIRECTED	The graph will be directed and an element specifies the weight of the edge between two vertices.
IGRAPH_ADJ_UNDIRECTED	This is the same as IGRAPH_ADJ_MAX, for convenience.
IGRAPH_ADJ_MAX	An undirected graph will be created and the weight of the edge between vertices i and j is max(A(i,j), A(j,i)).
IGRAPH_ADJ_MIN	An undirected graph will be created and the weight of the edge between vertices i and j is min(A(i,j), A(j,i)).
IGRAPH_ADJ_PLUS	An undirected graph will be created and the weight of the edge between vertices i and j is A(i,j)+A(j,i).
IGRAPH_ADJ_UPPER	An undirected graph will be created. Only the upper right triangle (including the diagonal) is used for the edge weights.
IGRAPH_ADJ_LOWER	An undirected graph will be created. Only the lower left triangle (including the diagonal) is used for the edge weights.

weights: Pointer to an initialized vector, the weights will be stored here.

loops: Constant to specify how the diagonal of the matrix should be treated when creating loop edges. Ignored for modes IGRAPH_ADJ_DIRECTED, IGRAPH_ADJ_UPPER and IGRAPH_ADJ_LOWER.

IGRAPH_NO_LOOPS	Ignore the diagonal of the input matrix and do not create loops.
IGRAPH_LOOPS_ONCE	Treat the diagonal entries as the weight of the loop edge incident on the corresponding vertex.
IGRAPH_LOOPS_TWICE	Treat the diagonal entries as *twice* the weight of the loop edge incident on the corresponding vertex.

Returns:

Error code, IGRAPH_EINVAL: non-square matrix.

Time complexity: O(|V||V|), |V| is the number of vertices in the graph.

Example 11.3. File examples/simple/igraph_weighted_adjacency.c

igraph_sparse_adjacency — Creates a graph from a sparse adjacency matrix.

```
igraph_error_t igraph_sparse_adjacency(igraph_t *graph, igraph_sparsemat_t *adj
        igraph_adjacency_t mode, igraph_loops_t loops);
```

This has the same functionality as `igraph_adjacency()`, but uses a column-compressed adjacency matrix.

Time complexity: O(|E|), where |E| is the number of edges in the graph.

igraph_sparse_weighted_adjacency — Creates a graph from a weighted sparse adjacency matrix.

```
igraph_error_t igraph_sparse_weighted_adjacency(
    igraph_t *graph, igraph_sparsemat_t *adjmatrix, igraph_adjacency_t mode,
    igraph_vector_t *weights, igraph_loops_t loops
);
```

This has the same functionality as `igraph_weighted_adjacency()`, but uses a column-compressed adjacency matrix.

Time complexity: O(|E|), where |E| is the number of edges in the graph.

igraph_adjlist — Creates a graph from an adjacency list.

```
igraph_error_t igraph_adjlist(igraph_t *graph, const igraph_adjlist_t *adjlist,
                    igraph_neimode_t mode, igraph_bool_t duplicate);
```

An adjacency list is a list of vectors, containing the neighbors of all vertices. For operations that involve many changes to the graph structure, it is recommended that you convert the graph into an adjacency list via `igraph_adjlist_init()`, perform the modifications (these are cheap for an adjacency list) and then recreate the igraph graph via this function.

Arguments:

graph: Pointer to an uninitialized graph object.

adjlist: The adjacency list.

mode: Whether or not to create a directed graph. `IGRAPH_ALL` means an undirected graph, `IGRAPH_OUT` means a directed graph from an out-adjacency list (i.e. each list contains the successors of the corresponding vertices), `IGRAPH_IN` means a directed graph from an in-adjacency list

duplicate: Boolean constant. For undirected graphs this specifies whether each edge is included twice, in the vectors of both adjacent vertices. If this is `false`, then it is assumed that every edge is included only once. This argument is ignored for directed graphs.

Returns:

Error code.

See also:

`igraph_adjlist_init()` for the opposite operation.

Time complexity: O(|V|+|E|).

Regular structures

These functions produce various basic regular graph structures, such as paths, cycles or lattices.

`igraph_star` — Creates a *star* graph, every vertex connects only to the center.

```
igraph_error_t igraph_star(igraph_t *graph, igraph_int_t n, igraph_star_mode_t m
               igraph_int_t center);
```

Arguments:

graph: Pointer to an uninitialized graph object, this will be the result.

n: Integer constant, the number of vertices in the graph.

mode: Constant, gives the type of the star graph to create. Possible values:

`IGRAPH_STAR_OUT`	directed star graph, edges point *from* the center to the other vertices.
`IGRAPH_STAR_IN`	directed star graph, edges point *to* the center from the other vertices.
`IGRAPH_STAR_MUTUAL`	directed star graph with mutual edges.
`IGRAPH_STAR_UNDIRECTED`	an undirected star graph is created.

center: Id of the vertex which will be the center of the graph.

Returns:

Error code:

`IGRAPH_EINVVID`	invalid number of vertices.
`IGRAPH_EINVAL`	invalid center vertex.
`IGRAPH_EINVMODE`	invalid mode argument.

Time complexity: O(|V|), the number of vertices in the graph.

See also:

`igraph_wheel()`, `igraph_square_lattice()`, `igraph_ring()`, `igraph_kary_tree()` for creating other regular structures.

Example 11.4. File `examples/simple/igraph_star.c`

`igraph_wheel` — Creates a *wheel* graph, a union of a star and a cycle graph.

```
igraph_error_t igraph_wheel(igraph_t *graph, igraph_int_t n, igraph_wheel_mode_
                igraph_int_t center);
```

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).

Arguments:

graph: Pointer to an uninitialized graph object, this will be the result.

n: Integer constant, the number of vertices in the graph.

mode: Constant, gives the type of the star graph to create. Possible values:

`IGRAPH_WHEEL_OUT`	directed wheel graph, edges point *from* the center to the other vertices.
`IGRAPH_WHEEL_IN`	directed wheel graph, edges point *to* the center from the other vertices.
`IGRAPH_WHEEL_MUTUAL`	directed wheel graph with mutual edges.
`IGRAPH_WHEEL_UNDIRECTED`	an undirected wheel graph is created.

center: Id of the vertex which will be the center of the graph.

Returns:

Error code:

`IGRAPH_EINVVID`	invalid number of vertices.
`IGRAPH_EINVAL`	invalid center vertex.
`IGRAPH_EINVMODE`	invalid mode argument.

Time complexity: O(|V|), the number of vertices in the graph.

See also:

`igraph_square_lattice()`, `igraph_ring()`, `igraph_star()`, `igraph_kary_tree()` for creating other regular structures.

`igraph_hypercube` — The n-dimensional hypercube graph.

```
igraph_error_t igraph_hypercube(igraph_t *graph,
```

```
                                    igraph_int_t n, igraph_bool_t directed);
```

The hypercube graph `Q_n` has `2^n` vertices and `2^(n-1) n` edges. Two vertices are connected when the binary representations of their zero-based vertex IDs differs in precisely one bit.

Arguments:

graph: An uninitialized graph object.

n: The dimension of the hypercube graph.

directed: Whether the graph should be directed. Edges will point from lower index vertices towards higher index ones.

Returns:

Error code.

See also:

`igraph_square_lattice()`

Time complexity: O(2^n)

`igraph_square_lattice` — Arbitrary dimensional square lattices.

```
igraph_error_t igraph_square_lattice(
    igraph_t *graph, const igraph_vector_int_t *dimvector, igraph_int_t nei,
    igraph_bool_t directed, igraph_bool_t mutual, const igraph_vector_bool_t *pe
);
```

Creates d-dimensional square lattices of the given size. Optionally, the lattice can be made periodic, and the neighbors within a given graph distance can be connected.

In the zero-dimensional case, the singleton graph is returned.

The vertices of the resulting graph are ordered such that the index of the vertex at position `(i_1, i_2, i_3, ..., i_d)` in a lattice of size `(n_1, n_2, ..., n_d)` will be `i_1 + n_1 * i_2 + n_1 * n_2 * i_3 + ...`.

Arguments:

graph: An uninitialized graph object.

dimvector: Vector giving the sizes of the lattice in each of its dimensions. The dimension of the lattice will be the same as the length of this vector.

nei: Integer value giving the distance (number of steps) within which two vertices will be connected.

directed: Boolean, whether to create a directed graph. If the `mutual` and `circular` arguments are not set to true, edges will be directed from lower-index vertices towards higher-index ones.

mutual: Boolean, if the graph is directed this gives whether to create all connections as mutual.

periodic: Boolean vector, defines whether the generated lattice is periodic along each dimension. The length of this vector must match the length of *dimvector*. This parameter may also be NULL, which implies that the lattice will not be periodic.

Returns:

Error code: IGRAPH_EINVAL: invalid (negative) dimension vector or mismatch between the length of the dimension vector and the periodicity vector.

See also:

igraph_hypercube() to create a hypercube graph; igraph_ring() to create a cycle graph or path graph; igraph_triangular_lattice() and igraph_hexagonal_lattice() to create other types of lattices; igraph_regular_tree() to create a Bethe lattice.

Time complexity: If *nei* is less than two then it is O(|V|+|E|) (as far as I remember), |V| and |E| are the number of vertices and edges in the generated graph. Otherwise it is O(|V|*d^k+|E|), d is the average degree of the graph, k is the *nei* argument.

igraph_triangular_lattice — A triangular lattice with the given shape.

```
igraph_error_t igraph_triangular_lattice(
        igraph_t *graph, const igraph_vector_int_t *dims,
        igraph_bool_t directed, igraph_bool_t mutual);
```

Creates a triangular lattice whose vertices have the form (i, j) for non-negative integers i and j and (i, j) is generally connected with (i + 1, j), (i, j + 1), and (i - 1, j + 1). The function constructs a planar dual of the graph constructed by igraph_hexagonal_lattice(). In particular, there a one-to-one correspondence between the vertices in the constructed graph and the cycles of length 6 in the graph constructed by igraph_hexagonal_lattice() with the same *dims* parameter.

The vertices of the resulting graph are ordered lexicographically with the 2nd coordinate being more significant, e.g., (i, j) < (i + 1, j) and (i + 1, j) < (i, j + 1)

Arguments:

graph: An uninitialized graph object.

dims: Integer vector, defines the shape of the lattice. If *dims* is of length 1, the resulting lattice has a triangular shape where each side of the triangle contains dims[0] vertices. If *dims* is of length 2, the resulting lattice has a "quasi rectangular" shape with the sides containing dims[0] and dims[1] vertices, respectively. If *dims* is of length 3, the resulting lattice has a hexagonal shape where the sides of the hexagon contain dims[0], dims[1] and dims[2] vertices. All dimensions must be non-negative.

directed: Boolean, whether to create a directed graph. If the mutual argument is not set to true, edges will be directed from lower-index vertices towards higher-index ones.

mutual: Boolean, if the graph is directed this gives whether to create all connections as mutual.

Returns:

Error code: IGRAPH_EINVAL: The size of *dims* must be either 1, 2, or 3 with all the components at least 1.

See also:

`igraph_hexagonal_lattice()` and `igraph_square_lattice()` for creating other types of lattices; `igraph_regular_tree()` to create a Bethe lattice.

Time complexity: O(|V|), where |V| is the number of vertices in the generated graph.

`igraph_hexagonal_lattice` — A hexagonal lattice with the given shape.

```
igraph_error_t igraph_hexagonal_lattice(
        igraph_t *graph, const igraph_vector_int_t *dims,
        igraph_bool_t directed, igraph_bool_t mutual);
```

Creates a hexagonal lattice whose vertices have the form (i, j) for non-negative integers i and j and (i, j) is generally connected with (i + 1, j), and if i is odd also with (i - 1, j + 1). The function constructs a planar dual of the graph constructed by `igraph_triangular_lattice()`. In particular, there a one-to-one correspondence between the cycles of length 6 in the constructed graph and the vertices of the graph constructed by `igraph_triangular_lattice()` function with the same *`dims`* parameter.

The vertices of the resulting graph are ordered lexicographically with the 2nd coordinate being more significant, e.g., (i, j) < (i + 1, j) and (i + 1, j) < (i, j + 1)

Arguments:

`graph`: An uninitialized graph object.

`dims`: Integer vector, defines the shape of the lattice. If *`dims`* is of length 1, the resulting lattice has a triangular shape where each side of the triangle contains `dims[0]` vertices. If *`dims`* is of length 2, the resulting lattice has a "quasi rectangular" shape with the sides containing `dims[0]` and `dims[1]` vertices, respectively. If *`dims`* is of length 3, the resulting lattice has a hexagonal shape where the sides of the hexagon contain `dims[0]`, `dims[1]` and `dims[2]` vertices. All coordinates must be non-negative.

`directed`: Boolean, whether to create a directed graph. If the `mutual` argument is not set to true, edges will be directed from lower-index vertices towards higher-index ones.

`mutual`: Boolean, if the graph is directed this gives whether to create all connections as mutual.

Returns:

Error code: `IGRAPH_EINVAL`: The size of *`dims`* must be either 1, 2, or 3 with all the components at least 1.

See also:

`igraph_triangular_lattice()` and `igraph_square_lattice()` for creating other types of lattices; ; `igraph_regular_tree()` to create a Bethe lattice.

Time complexity: O(|V|), where |V| is the number of vertices in the generated graph.

`igraph_ring` — Creates a *cycle* graph or a *path* graph.

```
igraph_error_t igraph_ring(igraph_t *graph, igraph_int_t n, igraph_bool_t direc
                igraph_bool_t mutual, igraph_bool_t circular);
```

A circular ring on `n` vertices is commonly known in graph theory as the cycle graph, and often denoted by `C_n`. Removing a single edge from the cycle graph `C_n` results in the path graph `P_n`. This function can generate both.

When *`n`* is 1 or 2, the result may not be a simple graph: the one-cycle contains a self-loop and the undirected or reciprocally connected directed two-cycle contains parallel edges.

Arguments:

`graph`: Pointer to an uninitialized graph object.

`n`: The number of vertices in the graph.

`directed`: Whether to create a directed graph. All edges will be oriented in the same direction along the cycle or path.

`mutual`: Whether to create mutual edges in directed graphs. It is ignored for undirected graphs.

`circular`: Whether to create a closed ring (a cycle) or an open path.

Returns:

Error code: `IGRAPH_EINVAL`: invalid number of vertices.

Time complexity: O(|V|), the number of vertices in the graph.

See also:

`igraph_square_lattice()` for generating more general (periodic or non-periodic) lattices.

Example 11.5. File `examples/simple/igraph_ring.c`

`igraph_path_graph` — A path graph `P_n`.

```
igraph_error_t igraph_path_graph(
        igraph_t *graph, igraph_int_t n,
        igraph_bool_t directed, igraph_bool_t mutual);
```

Creates the path graph `P_n` on *`n`* vertices.

This is a convenience wrapper to `igraph_ring()`.

Arguments:

`graph`: Pointer to an uninitialized graph object.

`n`: The number of vertices in the graph.

`directed`: Whether to create a directed graph.

`mutual`: Whether to create mutual edges in directed graphs. It is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(|V|), the number of vertices in the graph.

igraph_cycle_graph — A cycle graph C_n.

```
igraph_error_t igraph_cycle_graph(
        igraph_t *graph, igraph_int_t n,
        igraph_bool_t directed, igraph_bool_t mutual);
```

Creates the cycle graph C_n on *n* vertices.

When *n* is 1 or 2, the result may not be a simple graph: the one-cycle contains a self-loop and the undirected or reciprocally connected directed two-cycle contains parallel edges.

This is a convenience wrapper to `igraph_ring()`.

Arguments:

graph: Pointer to an uninitialized graph object.

n: The number of vertices in the graph.

directed: Whether to create a directed graph.

mutual: Whether to create mutual edges in directed graphs. It is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(|V|), the number of vertices in the graph.

igraph_lcf — Creates a graph from LCF notation.

```
igraph_error_t igraph_lcf(igraph_t *graph, igraph_int_t n,
                          const igraph_vector_int_t *shifts,
                          igraph_int_t repeats);
```

LCF notation (named after Lederberg, Coxeter and Frucht) is a concise notation for 3-regular Hamiltonian graphs. It consists 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.

Arguments:

graph: Pointer to an uninitialized graph object.

n: Integer constant giving the number of vertices. This is normally set to the number of shifts multiplied by the number of repeats.

shifts: An integer vector giving the shifts.

repeats: The number of repeats for the shifts.

Returns:

Error code.

See also:

`igraph_lcf_small()`, `igraph_extended_chordal_ring()`

Time complexity: O(|V|+|E|), linear in the number of vertices plus the number of edges.

`igraph_lcf_small` — Shorthand to create a graph from LCF notation, giving shifts as the arguments.

```
igraph_error_t igraph_lcf_small(igraph_t *graph, igraph_int_t n, ...);
```

This function provides a shorthand to give the shifts of the LCF notation directly as function arguments. See `igraph_lcf()` for an explanation of LCF notation.

Arguments:

`graph`: Pointer to an uninitialized graph object.

`n`: Integer, the number of vertices in the graph.

`...`: The shifts and the number of repeats for the shifts, plus an additional 0 to mark the end of the arguments.

Returns:

Error code.

See also:

See `igraph_lcf()` for a similar function using an `igraph_vector_t` instead of the variable length argument list; `igraph_circulant()` to create circulant graphs.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

Example 11.6. File `examples/simple/igraph_lcf.c`

`igraph_circulant` — Creates a circulant graph.

```
igraph_error_t igraph_circulant(igraph_t *graph, igraph_int_t n, const igraph_v
```

A circulant graph `G(n, shifts)` consists of *`n`* vertices `v_0`, ..., `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.

The function can generate either directed or undirected graphs. It does not generate multi-edges or self-loops.

Arguments:

`graph`: Pointer to an uninitialized graph object, the result will be stored here.

`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.

Returns:

Error code.

See also:

`igraph_ring()`, `igraph_generalized_petersen()`, `igraph_extended_chordal_ring()`, `igraph_lcf()`

Time complexity: O(|V| |shifts|), the number of vertices in the graph times the number of shifts.

`igraph_extended_chordal_ring` — Create an extended chordal ring.

```
igraph_error_t igraph_extended_chordal_ring(
    igraph_t *graph, igraph_int_t nodes, const igraph_matrix_int_t *W,
    igraph_bool_t directed);
```

An extended chordal ring is a cycle graph with additional chords connecting its vertices. Each row `L` of the matrix *`W`* specifies a set of chords to be inserted, in the following way: vertex `i` will connect to a vertex `L[(i mod p)]` steps ahead of it along the cycle, where `p` is the length of `L`. In other words, vertex `i` will be connected to vertex `(i + L[(i mod p)]) mod nodes`. If multiple edges are defined in this way, this will output a non-simple graph. The result can be simplified using `igraph_simplify()`.

See also Kotsis, G: Interconnection Topologies for Parallel Processing Systems, PARS Mitteilungen 11, 1-6, 1993. The igraph extended chordal rings are not identical to the ones in the paper. In igraph the matrix specifies which edges to add. In the paper, a condition is specified which should simultaneously hold between two endpoints and the reverse endpoints.

Arguments:

`graph`: Pointer to an uninitialized graph object, the result will be stored here.

`nodes`: Integer constant, the number of vertices in the graph. It must be at least 3.

`W`: The matrix specifying the extra edges. The number of columns should divide the number of total vertices. The elements are allowed to be negative.

`directed`: Whether the graph should be directed.

Returns:

Error code.

See also:

`igraph_ring()`, `igraph_lcf()`, `igraph_circulant()`

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

Tree generators

These functions generate tree graphs.

igraph_kary_tree — Creates a k-ary tree in which almost all vertices have k children.

```
igraph_error_t igraph_kary_tree(igraph_t *graph, igraph_int_t n, igraph_int_t ch
               igraph_tree_mode_t type);
```

To obtain a completely symmetric tree with `l` layers, where each vertex has precisely *`children`* descendants, use `n = (children^(l+1) - 1) / (children - 1)`. Such trees are often called `k`-ary trees, where `k` refers to the number of children.

Note that for `n=0`, the null graph is returned, which is not considered to be a tree by `igraph_is_tree()`.

Arguments:

`graph`: Pointer to an uninitialized graph object.

`n`: Integer, the number of vertices in the graph.

`children`: Integer, the number of children of a vertex in the tree.

`type`: Constant, gives whether to create a directed tree, and if this is the case, also its orientation. Possible values:

`IGRAPH_TREE_OUT`	directed tree, the edges point from the parents to their children.
`IGRAPH_TREE_IN`	directed tree, the edges point from the children to their parents.
`IGRAPH_TREE_UNDIRECTED`	undirected tree.

Returns:

Error code: `IGRAPH_EINVAL`: invalid number of vertices. `IGRAPH_INVMODE`: invalid mode argument.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges in the graph.

See also:

`igraph_regular_tree()`, `igraph_symmetric_tree()` and `igraph_star()` for creating other regular structures; `igraph_from_prufer()` and `igraph_tree_from_parent_vector()` for creating arbitrary trees; `igraph_tree_game()` for uniform random sampling of trees; `igraph_realize_degree_sequence()` with `IGRAPH_REALIZE_DEGSEQ_SMALLEST` to create a tree with given degrees.

Example 11.7. File `examples/simple/igraph_kary_tree.c`

igraph_symmetric_tree — Creates a symmetric tree with the specified number of branches at each level.

```
igraph_error_t igraph_symmetric_tree(igraph_t *graph, const igraph_vector_int_t
               igraph_tree_mode_t type);
```

This function creates a tree in which all vertices at distance `d` from the root have *`branching_counts`*[d] children.

Arguments:

`graph`: Pointer to an uninitialized graph object.

`branches`: Vector detailing the number of branches at each level.

`type`: Constant, gives whether to create a directed tree, and if this is the case, also its orientation. Possible values:

`IGRAPH_TREE_OUT`	directed tree, the edges point from the parents to their children.
`IGRAPH_TREE_IN`	directed tree, the edges point from the children to their parents.
`IGRAPH_TREE_UNDIRECTED`	undirected tree.

Returns:

Error code: `IGRAPH_INVMODE`: invalid mode argument. `IGRAPH_EINVAL`: invalid number of children.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges in the graph.

See also:

`igraph_kary_tree()`, `igraph_regular_tree()` and `igraph_star()` for creating other regular tree structures; `igraph_from_prufer()` for creating arbitrary trees; `igraph_tree_game()` for uniform random sampling of trees.

Example 11.8. File `examples/simple/igraph_symmetric_tree.c`

`igraph_regular_tree` — Creates a regular tree.

```
igraph_error_t igraph_regular_tree(igraph_t *graph, igraph_int_t h, igraph_int_
```

All vertices of a regular tree, except its leaves, have the same total degree *`k`*. This is different from a k-ary tree (`igraph_kary_tree()`), where all vertices have the same number of children, thus the degre of the root is one less than the degree of the other internal vertices. Regular trees are also referred to as Bethe lattices.

Arguments:

`graph`: Pointer to an uninitialized graph object.

`h`: The height of the tree, i.e. the distance between the root and the leaves.

`k`: The degree of the regular tree.

`type`: Constant, gives whether to create a directed tree, and if this is the case, also its orientation. Possible values:

`IGRAPH_TREE_OUT`	directed tree, the edges point from the parents to their children.

`IGRAPH_TREE_IN`	directed tree, the edges point from the children to their parents.
`IGRAPH_TREE_UNDIRECTED`	undirected tree.

Returns:

Error code.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges in the graph.

See also:

`igraph_kary_tree()` to create k-ary tree where each vertex has the same number of children, i.e. out-degree, instead of the same total degree. `igraph_symmetric_tree()` to use a different number of children at each level.

Example 11.9. File `examples/simple/igraph_regular_tree.c`

`igraph_tree_from_parent_vector` — Constructs a tree or forest from a vector encoding the parent of each vertex.

```
igraph_error_t igraph_tree_from_parent_vector(
        igraph_t *graph,
        const igraph_vector_int_t *parents,
        igraph_tree_mode_t type);
```

Rooted trees and forests are conveniently represented using a *`parents`* vector where the ID of the parent of vertex `v` is stored in `parents[v]`. This function serves to construct an igraph graph from a parent vector representation. The result is guaranteed to be a forest or a tree. If the *`parents`* vector is found to encode a cycle or a self-loop, an error is raised.

Several igraph functions produce such vectors, such as graph traversal functions (`igraph_bfs()` and `igraph_dfs()`), shortest path functions that construct a shortest path tree, as well as some other specialized functions like `igraph_dominator_tree()` or `igraph_cohesive_blocks()`. Vertices which do not have parents (i.e. roots) get a negative entry in the *`parents`* vector.

Use `igraph_bfs()` or `igraph_dfs()` to convert a forest into a parent vector representation. For trees, i.e. forests with a single root, it is more convenient to use `igraph_bfs_simple()`.

Arguments:

`graph`: Pointer to an uninitialized graph object.

`parents`: The parent vector. `parents[v]` is the ID of the parent vertex of `v`. `parents[v] < 0` indicates that `v` does not have a parent.

`type`: Constant, gives whether to create a directed tree, and if this is the case, also its orientation. Possible values:

`IGRAPH_TREE_OUT`	directed tree, the edges point from the parents to their children.
`IGRAPH_TREE_IN`	directed tree, the edges point from the children to their parents.

```
IGRAPH_TREE_UNDIRECTED
undirected tree.
```

Returns:

Error code.

See also:

`igraph_bfs()`, `igraph_bfs_simple()` for back-conversion; `igraph_from_prufer()` for creating trees from Prüfer sequences; `igraph_is_tree()` and `igraph_is_forest()` to check if a graph is a tree or forest.

Time complexity: O(n) where n is the length of *`parents`*.

`igraph_from_prufer` — Generates a tree from a Prüfer sequence.

```
igraph_error_t igraph_from_prufer(igraph_t *graph, const igraph_vector_int_t *p
```

A Prüfer sequence is a unique sequence of integers associated with a labelled tree. A tree on `n` vertices can be represented by a sequence of `n-2` integers, each between `0` and `n-1` (inclusive). The algorithm used by this function is based on Paulius Micikevi#ius, Saverio Caminiti, Narsingh Deo: Linear-time Algorithms for Encoding Trees as Sequences of Node Labels

Arguments:

`graph`: Pointer to an uninitialized graph object.

`prufer`: The Prüfer sequence

Returns:

Error code:

`IGRAPH_ENOMEM` there is not enough memory to perform the operation.

`IGRAPH_EINVAL` invalid Prüfer sequence given

See also:

`igraph_to_prufer()`, `igraph_kary_tree()`, `igraph_tree_game()`

Time complexity: O(|V|), where |V| is the number of vertices in the tree.

Graphs with given degrees

These functions generate graphs with the specified degrees.

`igraph_realize_degree_sequence` — Generates a graph with the given degree sequence.

```
igraph_error_t igraph_realize_degree_sequence(
        igraph_t *graph,
        const igraph_vector_int_t *outdeg, const igraph_vector_int_t *indeg,
        igraph_edge_type_sw_t allowed_edge_types,
        igraph_realize_degseq_t method);
```

This function generates an undirected graph that realizes a given degree sequence, or a directed graph that realizes a given pair of out- and in-degree sequences.

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 of highest degree. In the directed case, the "highest" (in, out) degree pairs are determined based on lexicographic ordering. This step is repeated until all degrees have been connected up.

Loopless multigraphs are generated using an analogous algorithm: an arbitrary vertex is chosen, and it is connected with a single connection to a highest remaining degee vertex. If self-loops are also allowed, the same algorithm is used, but if a non-zero vertex remains at the end of the procedure, the graph is completed by adding self-loops to it. Thus, the result will contain at most one vertex with self-loops.

The `method` parameter controls the order in which the vertices to be connected are chosen. In the undirected case, `IGRAPH_REALIZE_DEGSEQ_SMALLEST` produces a connected graph when one exists. This makes this method suitable for constructing trees with a given degree sequence.

For a undirected simple graph, the time complexity is O(V + alpha(V) * E). For an undirected multi graph, the time complexity is O(V * E + V log V). For a directed graph, the time complexity is O(E + V^2 log V).

References:

V. Havel: Poznámka o existenci kone#ných graf# (A remark on the existence of finite graphs), #asopis pro p#stování matematiky 80, 477-480 (1955). http://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). https://www.jstor.org/stable/2098770

D. J. Kleitman and D. L. Wang: Algorithms for Constructing Graphs and Digraphs with Given Valences and Factors, Discrete Mathematics 6, 1 (1973). https://doi.org/10.1016/0012-365X%2873%2990037-X P. L. Erd#s, I. Miklós, Z. Toroczkai: A simple Havel-Hakimi type algorithm to realize graphical degree sequences of directed graphs, The Electronic Journal of Combinatorics 17.1 (2010). http://eudml.org/doc/227072

Sz. Horvát and C. D. Modes: Connectedness matters: construction and exact random sampling of connected networks (2021). https://doi.org/10.1088/2632-072X/abced5

Arguments:

graph:	Pointer to an uninitialized graph object.
outdeg:	The degree sequence of an undirected graph (if *indeg* is NULL), or the out-degree sequence of a directed graph (if *indeg* is given).
indeg:	The in-degree sequence of a directed graph. Pass `NULL` to generate an undirected graph.
allowed_edge_types:	The types of edges to allow in the graph. See `igraph_edge_type_sw_t`. For directed graphs, only `IGRAPH_SIMPLE_SW` is implemented at this moment. For undirected graphs, the following values are valid:

`IGRAPH_SIMPLE_SW`	simple graphs (i.e. no self-loops or multi-edges allowed).
`IGRAPH_LOOPS_SW`	single self-loops are allowed, but not multi-edges; currently not implemented.
`IGRAPH_MULTI_SW`	multi-edges are allowed, but not self-loops.
`IGRAPH_LOOPS_SW | IGRAPH_MULTI_SW`	both self-loops and multi-edges are allowed.

method: The method to generate the graph. Possible values:

`IGRAPH_REALIZE_DEGSEQ_SMALLEST`	The vertex with smallest remaining degree is selected first. 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, as well as http://szhorvat.net/pelican/hh-connected-graphs.html). This method can be used to construct a tree from its degrees. In the directed case it tends to generate weakly connected graphs, but this is not guaranteed.
`IGRAPH_REALIZE_DEGSEQ_LARGEST`	The vertex with the largest remaining degree is selected first. The result is usually a graph with high positive degree assortativity, and is often disconnected.
`IGRAPH_REALIZE_DEGSEQ_INDEX`	The vertices are selected in order of their index (i.e. their position in the degree vector). Note that sorting the degree vector and using the `INDEX` method is not equivalent to the `SMALLEST` method above, as `SMALLEST` uses the smallest *remaining* degree for selecting vertices, not the smallest *initial* degree.

Returns:

Error code:

IGRAPH_UNIMPLEMENTED	The requested method is not implemented.
IGRAPH_ENOMEM	There is not enough memory to perform the operation.
IGRAPH_EINVAL	Invalid method parameter, or invalid in- and/or out-degree vectors. The degree vectors should be non-negative, the length and sum of *outdeg* and *indeg* should match for directed graphs.

See also:

igraph_is_graphical() to test graphicality without generating a graph; igraph_realize_bipartite_degree_sequence() to create bipartite graphs from two degree sequence; igraph_degree_sequence_game() to generate random graphs with a given degree sequence; igraph_k_regular_game() to generate random regular graphs; igraph_rewire() to randomly rewire the edges of a graph while preserving its degree sequence.

Example 11.10. File examples/simple/igraph_realize_degree_sequence.c

igraph_realize_bipartite_degree_sequence — Generates a bipartite graph with the given bidegree sequence.

```
igraph_error_t igraph_realize_bipartite_degree_sequence(
    igraph_t *graph,
    const igraph_vector_int_t *degrees1, const igraph_vector_int_t *degrees2,
    const igraph_edge_type_sw_t allowed_edge_types, const igraph_realize_degseq
);
```

This function generates a bipartite graph with the given bidegree sequence, using a Havel-Hakimi-like construction algorithm. The order in which vertices are connected up is controlled by the *method* parameter. When using the IGRAPH_REALIZE_DEGSEQ_SMALLEST method, it is ensured that the graph will be connected if and only if the given bidegree sequence is potentially connected.

The vertices of the graph will be ordered so that those having *degrees1* come first, followed by *degrees2*.

Arguments:

graph:	Pointer to an uninitialized graph object.
degrees1:	The degree sequence of the first partition.
degrees2:	The degree sequence of the second partition.
allowed_edge_types:	The types of edges to allow in the graph.
	IGRAPH_SIMPLE_SW simple graph (i.e. no multi-edges allowed).
	IGRAPH_MULTI_SW multi-edges are allowed
method:	Controls the order in which vertices are selected for connection. Possible values:

`IGRAPH_REALIZE_DEGSEQ_SMALLEST`	The vertex with smallest remaining degree is selected first, from either partition. The result is usually a graph with high negative degree assortativity. This method is guaranteed to generate a connected graph, if one exists.
`IGRAPH_REALIZE_DEGSEQ_LARGEST`	The vertex with the largest remaining degree is selected first, from either parition. The result is usually a graph with high positive degree assortativity, and is often disconnected.
`IGRAPH_REALIZE_DEGSEQ_INDEX`	The vertices are selected in order of their index.

Returns:

Error code.

See also:

`igraph_is_bigraphical()` to test bigraphicality without generating a graph.

Complete graphs

These functions produce single and multipartite complete graphs, as well as related graphs.

`igraph_full` — Creates a full graph (complete graph).

```
igraph_error_t igraph_full(igraph_t *graph, igraph_int_t n, igraph_bool_t direc
                igraph_bool_t loops);
```

In a full graph every possible edge is present: every vertex is connected to every other vertex. **igraph** generalizes the usual concept of complete graphs in graph theory to graphs with self-loops as well as to directed graphs.

Arguments:

`graph`: Pointer to an uninitialized graph object.

`n`: Integer, the number of vertices in the graph.

`directed`: Whether to create a directed graph.

`loops`: Whether to include self-loops.

Returns:

Error code: `IGRAPH_EINVAL`: invalid number of vertices.

Time complexity: O(|V|^2) = O(|E|), where |V| is the number of vertices and |E| is the number of edges.

See also:

`igraph_square_lattice()`, `igraph_star()`, `igraph_kary_tree()` for creating other regular structures.

Example 11.11. File `examples/simple/igraph_full.c`

`igraph_full_citation` — Creates a full citation graph (a complete directed acyclic graph).

```
igraph_error_t igraph_full_citation(igraph_t *graph, igraph_int_t n,
                        igraph_bool_t directed);
```

This is a directed graph, where every `i->j` edge is present if and only if `j<i`. If the *`directed`* argument is false then an undirected graph is created, and it is just a complete graph.

Arguments:

`graph`: Pointer to an uninitialized graph object, the result is stored here.

`n`: The number of vertices.

`directed`: Whether to created a directed graph. If false an undirected graph is created.

Returns:

Error code.

See also:

`igraph_full()`

Time complexity: O(|V|^2) = O(|E|), where |V| is the number of vertices and |E| is the number of edges.

`igraph_full_multipartite` — Creates a full multipartite graph.

```
igraph_error_t igraph_full_multipartite(igraph_t *graph,
                         igraph_vector_int_t *types,
                         const igraph_vector_int_t *n,
                         igraph_bool_t directed,
                         igraph_neimode_t mode);
```

A multipartite graph contains two or more types of vertices and connections are only possible between two vertices of different types. This function creates a complete multipartite graph.

Arguments:

`graph`: Pointer to an uninitialized graph object, the graph will be created here.

`types`: Pointer to an integer vector. If not a null pointer, the type of each vertex will be stored here.

n: Pointer to an integer vector, the number of vertices of each type.

directed: Boolean, whether to create a directed graph.

mode: A constant that gives the type of connections for directed graphs. If `IGRAPH_OUT`, then edges point from vertices of low-index vertices to high-index vertices; if `IGRAPH_IN`, then the opposite direction is realized; `IGRAPH_ALL`, then mutual edges will be created.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

See also:

`igraph_full_bipartite()` for complete bipartite graphs, `igraph_turan()` for Turán graphs.

`igraph_turan` — Creates a Turán graph.

```
igraph_error_t igraph_turan(igraph_t *graph,
                            igraph_vector_int_t *types,
                            igraph_int_t n,
                            igraph_int_t r);
```

Turán graphs are complete multipartite graphs with the property that the sizes of the partitions are as close to equal as possible.

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.

Arguments:

graph: Pointer to an igraph_t object, the graph will be created here.

types: Pointer to an integer vector. If not a null pointer, the type (partition index) of each vertex will be stored here.

n: Integer, the number of vertices in the graph.

r: Integer, the number of partitions of the graph, must be positive.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

See also:

`igraph_full_multipartite()` for full multipartite graphs.

Pre-defined graphs

These functions return graphs from various graph collections.

`igraph_famous` — Create a famous graph by simply providing its name.

```
igraph_error_t igraph_famous(igraph_t *graph, const char *name);
```

The name of the graph can be simply supplied as a string. Note that this function creates graphs which don't take any parameters, there are separate functions for graphs with parameters, e.g. `igraph_full()` for creating a full graph.

The following graphs are supported:

`Bull`	The bull graph, 5 vertices, 5 edges, resembles 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 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 on 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-incident 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 non-line 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 hypotracable 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).

Arguments:

`graph`: Pointer to an uninitialized graph object.

`name`: Character constant, the name of the graph to be created, it is case insensitive.

Returns:

Error code, `IGRAPH_EINVAL` if there is no graph with the given name.

See also:

Other functions for creating graph structures: `igraph_ring()`, `igraph_kary_tree()`, `igraph_square_lattice()`, `igraph_full()`.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges in the graph.

`igraph_atlas` — Create a small graph from the “Graph Atlas”.

```
igraph_error_t igraph_atlas(igraph_t *graph, igraph_int_t number);
```

The graph atlas contains all simple undirected unlabeled graphs on between 0 and 7 vertices. The number of the graph is given as a parameter. The graphs are listed:

1. in increasing order of number of vertices;
2. for a fixed number of vertices, in increasing order of the number of edges;
3. for fixed numbers of vertices and edges, in lexicographically increasing order of the degree sequence, for example 111223 < 112222;
4. for fixed degree sequence, in increasing number of automorphisms.

The data was converted from the NetworkX software package, see https://networkx.org/.

See *An Atlas of Graphs* by Ronald C. Read and Robin J. Wilson, Oxford University Press, 1998.

Arguments:

`graph`: Pointer to an uninitialized graph object.

`number`: The number of the graph to generate. Must be between 0 and 1252 (inclusive). Graphs on 0-7 vertices start at numbers 0, 1, 2, 4, 8, 19, 53, and 209, respectively.

Returns:

Error code.

Added in version 0.2.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

Example 11.12. File `examples/simple/igraph_atlas.c`

Other well-known graphs from graph theory

`igraph_de_bruijn` — Generate a de Bruijn graph.

```
igraph_error_t igraph_de_bruijn(igraph_t *graph, igraph_int_t m, igraph_int_t n
```

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.

Arguments:

`graph`: Pointer to an uninitialized graph object, the result will be stored here.

`m`: Integer, the number of letters in the alphabet.

`n`: Integer, the length of the strings.

Returns:

Error code.

See also:

`igraph_kautz()`.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

`igraph_kautz` — Generate a Kautz graph.

```
igraph_error_t igraph_kautz(igraph_t *graph, igraph_int_t m, igraph_int_t n);
```

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. For string length 1 the new letter cannot equal the old letter, so there are no loops.

Kautz graphs have some interesting properties, see e.g. Wikipedia for details.

Vincent Matossian wrote the first version of this function in R, thanks.

Arguments:

graph: Pointer to an uninitialized graph object, the result will be stored here.

m: Integer, m+1 is the number of letters in the alphabet.

n: Integer, n+1 is the length of the strings.

Returns:

Error code.

See also:

igraph_de_bruijn().

Time complexity: O(|V|* [(m+1)/m]^n +|E|), in practice it is more like O(|V|+|E|). |V| is the number of vertices, |E| is the number of edges and m and n are the corresponding arguments.

igraph_generalized_petersen — Creates a Generalized Petersen graph.

```
igraph_error_t igraph_generalized_petersen(igraph_t *graph, igraph_int_t n, igra
```

The generalized Petersen graph G(n, k) consists of *n* vertices v_0, ..., v_n forming an "outer" cycle graph, and *n* additional vertices u_0, ..., u_n forming an "inner" circulant graph where u_i is connected to u_(i + k mod n). Additionally, all v_i are connected to u_i.

G(n, k) has 2n vertices and 3n edges. The Petersen graph itself is G(5, 2).

Reference:

M. E. Watkins, A Theorem on Tait Colorings with an Application to the Generalized Petersen Graphs, Journal of Combinatorial Theory 6, 152-164 (1969). https://doi.org/10.1016%2FS0021-9800%2869%2980116-X

Arguments:

graph: Pointer to an uninitialized graph object, the result will be stored here.

n: Integer, n is the number of vertices in the inner and outer cycle/circulant graphs. It must be at least 3.

k: Integer, k is the shift of the circulant graph. It must be positive and less than n/2.

Returns:

Error code.

See also:

igraph_famous() for the original Petersen graph.

Time complexity: O(|V|), the number of vertices in the graph.

`igraph_mycielski_graph` — The Mycielski graph of order *k*.

```
igraph_error_t igraph_mycielski_graph(igraph_t *graph, igraph_int_t k);
```

The Mycielski graph of order *k*, denoted `M_k`, is a triangle-free graph on *k* vertices with chromatic number *k*. It is defined through the Mycielski construction described in the documentation of `igraph_mycielskian()`.

Some authors define Mycielski graphs only for `k > 1`. igraph extends this to all `k >= 0`. The first few Mycielski graphs are:

1. M_0: Null graph
2. M_1: Single vertex
3. M_2: Path graph with 2 vertices
4. M_3: Cycle graph with 5 vertices
5. M_4: Grötzsch graph (a triangle-free graph with chromatic number 4)

The vertex count of `M_k` is `n_k = 3 * 2^(k-2) - 1` for `k > 1` and `k` otherwise. The edge count is `m_k = (7 * 3^(k-2) + 1) / 2 - 3 * 2^(k - 2)` for `k > 1` and 0 otherwise.

Arguments:

graph: Pointer to an uninitialized graph object. The generated Mycielski graph will be stored here.

k: Integer, the order of the Mycielski graph (must be non-negative).

Returns:

Error code.

See also:

`igraph_mycielskian()`.

Time complexity: O(3^k), i.e. exponential in *k*.

Chapter 12. Stochastic graph generators ("games")

"Games" are random graph generators, i.e. they generate a different graph every time they are called. igraph includes many such generators. Some implement stochastic graph construction processes inspired by real-world mechanics, such as preferential attachment, while others are designed to produce graphs with certain used properties (e.g. fixed number of edges, fixed degrees, etc.)

The Erd#s-Rényi and related models

There are two classic random graph models referred to as the Erd#s-Rényi random graph, or sometimes simply *the* random graph. Both fix the vertex count n, but while the G(n,m) model prescribes precisely m edges, the G(n,p) model connects all vertex pairs independently with probability p. While these models look superficially different, when n is large they behave in a similar manner. G(n,m) graphs have a density of exactly `p = m / m_max`, while G(n,p) graphs have `m = p m_max` edges on *average*, where `m_max` is the number of vertex pairs. Indeed, these two models turns out to be two sides of the same coin: both can be understood as maximum entropy models with a constraint on the number of edges. The G(n,m) is obtained from a sharp constraint, while G(n,p) from an average constraint (soft constraint).

The maximum entropy framework allows for rigorous generalizations of these models to various scenarios, of which igraph supports many, such as models defined over directed graphs, bipartite graphs, multigraphs, or even over edge-labelled graphs. Constraining edge counts between various subsets of vertices yields further families of related models, such as `igraph_sbm_game()` (given connection probabilities between categories) or `igraph_degree_sequence_game()` (given incident edge counts, i.e. degrees, for each vertex).

`igraph_erdos_renyi_game_gnm` — Generates a random (Erd#s-Rényi) graph with a fixed number of edges.

```
igraph_error_t igraph_erdos_renyi_game_gnm(
        igraph_t *graph,
        igraph_int_t n, igraph_int_t m,
        igraph_bool_t directed,
        igraph_edge_type_sw_t allowed_edge_types,
        igraph_bool_t edge_labeled);
```

In the `G(n, m)` Erd#s-Rényi model, a graph with *`n`* vertices and *`m`* edges is generated uniformly at random.

Arguments:

`graph`:	Pointer to an uninitialized graph object.
`n`:	The number of vertices in the graph.
`m`:	The number of edges in the graph.
`directed`:	Whether to generate a directed graph.
`allowed_edge_types`:	Controls whether multi-edges and self-loops are generated. See `igraph_edge_type_sw_t`.

edge_labeled: If true, the sampling is done uniformly from the set of ordered edge lists. See `igraph_iea_game()` for more information. Set this to `false` to select the classic Erd#s-Rényi model. The constants `IGRAPH_EDGE_UNLABELED` and `IGRAPH_EDGE_LABELED` may be used instead of `false` and `true` for better readability.

Returns:

Error code: `IGRAPH_EINVAL`: invalid *n* or *m* parameter. `IGRAPH_ENOMEM`: there is not enough memory for the operation.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges in the graph.

See also:

`igraph_erdos_renyi_game_gnp()` to sample from the related `G(n, p)` model, which constrains the *expected* edge count; `igraph_iea_game()` to generate multigraph by assigning edges to vertex pairs uniformly and independently; `igraph_degree_sequence_game()` to constrain the degree sequence; `igraph_bipartite_game_gnm()` for the bipartite version of this model; `igraph_barabasi_game()` and `igraph_growing_random_game()` for other commonly used random graph models.

Example 12.1. File `examples/simple/igraph_erdos_renyi_game_gnm.c`

`igraph_erdos_renyi_game_gnp` — Generates a random (Erd#s-Rényi) graph with fixed edge probabilities.

```
igraph_error_t igraph_erdos_renyi_game_gnp(
        igraph_t *graph,
        igraph_int_t n, igraph_real_t p,
        igraph_bool_t directed,
        igraph_edge_type_sw_t allowed_edge_types,
        igraph_bool_t edge_labeled);
```

In the `G(n, p)` Erd#s-Rényi model, also known as the Gilbert model, or Bernoulli random graph, a graph with *n* vertices is generated such that every possible edge is included in the graph independently with probability *p*. This is equivalent to a maximum entropy random graph model model with a constraint on the *expected* edge count. The maximum entropy view allows for extending the model to multigraphs, as discussed by Park and Newman (2004), section III.D. In this case, *p* is interpreted as the expected number of edges between any vertex pair.

Setting `p = 1/2` and `multiple = false` generates all graphs without multi-edges on *n* vertices with the same probability.

For both simple and multigraphs, the expected mean degree of the graph is approximately `p n`; set `p = k/n` when a mean degree of approximately `k` is desired. More precisely, the expected mean degree is `p(n-1)` in (undirected or directed) graphs without self-loops, `p(n+1)` in undirected graphs with self-loops, and `p n` in directed graphs with self-loops.

When generating multigraphs, the distribution of the edge multiplicities is geometric, i.e. the probability of finding `m` edges between two vertices is `q (1-q)^m`, where `q = 1 / (1+p)`.

This function uses the sequential geometric sampling technique described in Batagelj and Brandes (2005), with a modification to handle multigraphs.

References:

J. Park and M. E. J. Newman: "Statistical Mechanics of Networks". Phys. Rev. E 70, 066117 (2004). https://doi.org/10.1103/PhysRevE.70.066117

V. Batagelj and U. Brandes: "Efficient Generation of Large Random Networks". Phys. Rev. E 71, 036113 (2005). https://doi.org/10.1103/PhysRevE.71.036113

Arguments:

graph:	Pointer to an uninitialized graph object.
n:	The number of vertices in the graph.
p:	The expected number of edges between any vertex pair. When multi-edges are disallowed, this is equivalent to the probability of having a connection between any two vertices.
directed:	Whether to generate a directed graph.
allowed_edge_types:	Controls whether multi-edges and self-loops are generated. See `igraph_edge_type_sw_t`.
edge_labeled:	If true, the model is defined over the set of ordered edge lists, i.e. over the set of edge-labeled graphs. Set it to `false` to select the classic Erd#s-Rényi model. The constants `IGRAPH_EDGE_UNLABELED` and `IGRAPH_EDGE_LABELED` may be used instead of `false` and `true` for better readability.

Returns:

Error code: `IGRAPH_EINVAL`: invalid *n* or *p* parameter. `IGRAPH_ENOMEM`: there is not enough memory for the operation.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges in the graph.

See also:

`igraph_erdos_renyi_game_gnm()` to generate random graphs with a sharply fixed edge count; `igraph_chung_lu_game()` and `igraph_static_fitness_game()` to generate random graphs with a fixed expected degree sequence; `igraph_bipartite_game_gnm()` for the bipartite version of this model; `igraph_barabasi_game()` and `igraph_growing_random_game()` for other commonly used random graph models.

Example 12.2. File `examples/simple/igraph_erdos_renyi_game_gnp.c`

`igraph_iea_game` — Generates a random multigraph through independent edge assignment.

```
igraph_error_t igraph_iea_game(
        igraph_t *graph,
        igraph_int_t n, igraph_int_t m,
        igraph_bool_t directed, igraph_bool_t loops);
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

This model generates random multigraphs on *n* vertices with *m* edges through independent edge assignment (IEA). Each of the *m* edges is assigned uniformly at random to an *ordered* vertex pair, independently of each other.

This model does not sample multigraphs uniformly. 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.

An alternative way to think of this model is that it performs uniform sampling of *edge-labeled* graphs, i.e. graphs in which not only vertices, but also edges carry unique identities and are distinguishable.

Arguments:

graph: Pointer to an uninitialized graph object.

n: The number of vertices in the graph.

m: The number of edges in the graph.

directed: Whether to generate a directed graph.

loops: Whether to generate self-loops.

Returns:

Error code: `IGRAPH_EINVAL`: invalid *n* or *m* parameter. `IGRAPH_ENOMEM`: there is not enough memory for the operation.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges in the graph.

See also:

`igraph_erdos_renyi_game_gnm()` to uniformly sample graphs with a given number of vertices and edges.

`igraph_sbm_game` — Sample from a stochastic block model.

```
igraph_error_t igraph_sbm_game(
        igraph_t *graph,
        const igraph_matrix_t *pref_matrix,
```

```
        const igraph_vector_int_t *block_sizes,
        igraph_bool_t directed,
        igraph_edge_type_sw_t allowed_edge_types);
```

This function samples graphs from a stochastic block model, a generalization of the G(n,p) model where the connection probability p (or expected number of edges for multigraphs) is specified separately between and within a given group of vertices.

The order of the vertex IDs in the generated graph corresponds to the *`block_sizes`* argument.

Reference:

Faust, K., & Wasserman, S. (1992a). Blockmodels: Interpretation and evaluation. Social Networks, 14, 5–-61. https://doi.org/10.1016/0378-8733(92)90013-W

Arguments:

`graph`: The output graph. This should be a pointer to an uninitialized graph.

`pref_matrix`: The matrix giving the connection probabilities (or expected edge multiplicities for multigraphs) between groups. This is a k-by-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).

`block_sizes`: An integer vector giving the number of vertices in each group.

`directed`: Boolean, whether to create a directed graph. If this argument is `false`, then *`pref_matrix`* must be symmetric.

`allowed_edge_types`: Controls whether multi-edges and self-loops are generated. See `igraph_edge_type_sw_t`.

Returns:

Error code.

Time complexity: O(|V|+|E|+k^2), where |V| is the number of vertices, |E| is the number of edges, and k is the number of groups.

See also:

`igraph_erdos_renyi_game_gnp()` for a simple Bernoulli graph.

`igraph_hsbm_game` — Hierarchical stochastic block model.

```
igraph_error_t igraph_hsbm_game(igraph_t *graph, igraph_int_t n,
                    igraph_int_t m, const igraph_vector_t *rho,
                    const igraph_matrix_t *C, igraph_real_t p);
```

The function generates a random graph according to the hierarchical stochastic block model.

Arguments:

`graph`: The generated graph is stored here.

`n`: The number of vertices in the graph.

m: The number of vertices per block. n/m must be integer.

rho: 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.

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.

p: The Bernoulli rate of connections between vertices in different blocks.

Returns:

Error code.

See also:

`igraph_sbm_game()` for the classic stochastic block model, `igraph_hsbm_list_game()` for a more general version.

`igraph_hsbm_list_game` — Hierarchical stochastic block model, more general version.

```
igraph_error_t igraph_hsbm_list_game(igraph_t *graph, igraph_int_t n,
                           const igraph_vector_int_t *mlist,
                           const igraph_vector_list_t *rholist,
                           const igraph_matrix_list_t *Clist,
                           igraph_real_t p);
```

The function generates a random graph according to the hierarchical stochastic block model.

Arguments:

graph: The generated graph is stored here.

n: The number of vertices in the graph.

mlist: An integer vector of block sizes.

rholist: A list of rho vectors (`igraph_vector_t` objects), one for each block.

Clist: A list of square matrices (`igraph_matrix_t` objects), one for each block, specifying the Bernoulli rates of connections within the block.

p: The Bernoulli rate of connections between vertices in different blocks.

Returns:

Error code.

See also:

`igraph_sbm_game()` for the classic stochastic block model, `igraph_hsbm_game()` for a simpler general version.

igraph_preference_game — Generates a graph with vertex types and connection preferences.

```
igraph_error_t igraph_preference_game(igraph_t *graph, igraph_int_t nodes,
                            igraph_int_t types,
                            const igraph_vector_t *type_dist,
                            igraph_bool_t fixed_sizes,
                            const igraph_matrix_t *pref_matrix,
                            igraph_vector_int_t *node_type_vec,
                            igraph_bool_t directed,
                            igraph_bool_t loops);
```

This is practically the nongrowing variant of `igraph_establishment_game()`. A given number of vertices are generated. Every vertex is assigned to a vertex type according to the given type probabilities. Finally, every vertex pair is evaluated and an edge is created between them with a probability depending on the types of the vertices involved.

In other words, this function generates a graph according to a block-model. Vertices are divided into groups (or blocks), and the probability the two vertices are connected depends on their groups only.

Arguments:

graph: Pointer to an uninitialized graph.

nodes: The number of vertices in the graph.

types: The number of vertex types.

type_dist: Vector giving the distribution of vertex types. If `NULL`, all vertex types will have equal probability. See also the *fixed_sizes* argument.

fixed_sizes: Boolean. If true, then the number of vertices with a given vertex type is fixed and the *type_dist* argument gives these numbers for each vertex type. If true, and *type_dist* is `NULL`, then the function tries to make vertex groups of the same size. If this is not possible, then some groups will have an extra vertex.

pref_matrix: Matrix giving the connection probabilities for different vertex types. This should be symmetric if the requested graph is undirected.

node_type_vec: A vector where the individual generated vertex types will be stored. If `NULL`, the vertex types won't be saved.

directed: Whether to generate a directed graph. If undirected graphs are requested, only the lower left triangle of the preference matrix is considered.

loops: Whether loop edges are allowed.

Returns:

Error code.

Added in version 0.3.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges in the graph.

See also:

`igraph_asymmetric_preference_game()`, `igraph_establishment_game()`, `igraph_callaway_traits_game()`

`igraph_asymmetric_preference_game` — Generates a graph with asymmetric vertex types and connection preferences.

```
igraph_error_t igraph_asymmetric_preference_game(igraph_t *graph, igraph_int_t
                                        igraph_int_t no_out_types,
                                        igraph_int_t no_in_types,
                                        const igraph_matrix_t *type_dist_matrix,
                                        const igraph_matrix_t *pref_matrix,
                                        igraph_vector_int_t *node_type_out_vec,
                                        igraph_vector_int_t *node_type_in_vec,
                                        igraph_bool_t loops);
```

This is the asymmetric variant of `igraph_preference_game()`. A given number of vertices are generated. Every vertex is assigned to an "outgoing" and an "incoming " vertex type according to the given joint type probabilities. Finally, every vertex pair is evaluated and a directed edge is created between them with a probability depending on the "outgoing" type of the source vertex and the "incoming" type of the target vertex.

Arguments:

graph: Pointer to an uninitialized graph.

nodes: The number of vertices in the graph.

no_out_types: The number of vertex out-types.

no_in_types: The number of vertex in-types.

type_dist_matrix: Matrix of size `out_types * in_types`, giving the joint distribution of vertex types. If `NULL`, incoming and outgoing vertex types are independent and uniformly distributed.

pref_matrix: Matrix of size `out_types * in_types`, giving the connection probabilities for different vertex types.

node_type_out_vec: A vector where the individual generated "outgoing" vertex types will be stored. If `NULL`, the vertex types won't be saved.

node_type_in_vec: A vector where the individual generated "incoming" vertex types will be stored. If `NULL`, the vertex types won't be saved.

loops: Whether loop edges are allowed.

Returns:

Error code.

Added in version 0.3.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges in the graph.

See also:

`igraph_preference_game()`

`igraph_correlated_game` — Generates a random graph correlated to an existing graph.

```
igraph_error_t igraph_correlated_game(igraph_t *new_graph, const igraph_t *old_
                           igraph_real_t corr, igraph_real_t p,
                           const igraph_vector_int_t *permutation);
```

Sample a new graph by perturbing the adjacency matrix of a given simple graph and shuffling its vertices.

Arguments:

`new_graph`: The new graph to initialize based on an existing graph.

`old_graph`: The original graph, which must be a simple graph.

`corr`: A value in the unit interval [0,1], the target Pearson correlation between the adjacency matrices of the original and the generated graph (the adjacency matrix being used as a vector).

`p`: The probability of an edge between two vertices. It must in the open (0,1) interval. Typically, the density of *`old_graph`*.

`permutation`: A permutation to apply to the vertices of the generated graph. The i-th element of the vector specifies the index of the vertex in the *original* graph that will become vertex i in the new graph. It can also be a null pointer, in which case the vertices will not be permuted.

Returns:

Error code

See also:

`igraph_correlated_pair_game()` for generating a pair of correlated random graphs in one go.

`igraph_correlated_pair_game` — Generates pairs of correlated random graphs.

```
igraph_error_t igraph_correlated_pair_game(igraph_t *graph1, igraph_t *graph2,
                                igraph_int_t n, igraph_real_t corr, igraph_real_
                                igraph_bool_t directed,
                                const igraph_vector_int_t *permutation);
```

Sample two random graphs, with given correlation.

Arguments:

graph1: The first graph will be stored here.

graph2: The second graph will be stored here.

n: The number of vertices in both 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: Whether to generate directed graphs.

permutation: A permutation to apply to the vertices of the generated graph. The i-th element of the vector specifies the index of the vertex in the *first* graph that will become vertex i in the second graph. It can also be a null pointer, in which case the vertices will not be permuted.

Returns:

Error code

See also:

`igraph_correlated_game()` for generating a correlated pair to a given graph.

Preferential attachment and related models

Preferential attachment models are growing random graphs where vertices are added iteratively, and connected to previously added vertices based on dynamically changing vertex properties, such as degree or time since the vertex was added.

`igraph_barabasi_game` — Generates a graph based on the Barabási-Albert model.

```
igraph_error_t igraph_barabasi_game(igraph_t *graph, igraph_int_t n,
                          igraph_real_t power,
                          igraph_int_t m,
                          const igraph_vector_int_t *outseq,
                          igraph_bool_t outpref,
                          igraph_real_t A,
                          igraph_bool_t directed,
                          igraph_barabasi_algorithm_t algo,
                          const igraph_t *start_from);
```

This function implements several variants of the preferential attachment process, including linear and non-linear varieties of the Barabási-Albert and Price models. The graph construction starts with a single vertex, or an existing graph given by the *start_from* parameter. Then new vertices are added one at a time. Each new vertex connects to *m* existing vertices, choosing them with probabilities proportional to

`d^power + A`,

where d is the in- or total degree of the existing vertex (controlled by the *outpref* argument), while *power* and *A* are given by parameters. The *constant attractiveness A* is used to ensure that vertices with zero in-degree can also be connected to with non-zero probability.

Barabási, A.-L. and Albert R. 1999. Emergence of scaling in random networks, Science, 286 509--512. https://doi.org/10.1126/science.286.5439.509

de Solla Price, D. J. 1965. Networks of Scientific Papers, Science, 149 510--515. https://doi.org/10.1126/science.149.3683.510

Arguments:

graph: An uninitialized graph object.

n: The number of vertices in the graph.

power: Power of the preferential attachment. In the classic preferential attachment model `power=1`. Other values allow for sampling from a non-linear preferential attachment model. Negative values are only allowed when no zero-degree vertices are present during the construction process, i.e. when the starting graph has no isolated vertices and *outpref* is set to `true`.

m: The number of outgoing edges generated for each vertex. Only used when *outseq* is `NULL`.

outseq: Gives the (out-)degrees of the vertices. If this is constant, this can be a `NULL` pointer. In this case *m* contains the constant out-degree. The very first vertex has by definition no outgoing edges, so the first number in this vector is ignored.

outpref: Boolean, if true not only the in- but also the out-degree of a vertex increases its citation probability. I.e., the citation probability is determined by the total degree of the vertices. Ignored and assumed to be true if the graph being generated is undirected.

A: The constant attractiveness of vertices. When *outpref* is set to `false`, it should be positive to ensure that zero in-degree vertices can be connected to as well.

directed: Boolean, whether to generate a directed graph. When set to `false`, outpref is assumed to be `true`.

algo: The algorithm to use to generate the network. Possible values:

`IGRAPH_BARABASI_BAG`	This is the algorithm that was previously (before version 0.6) solely implemented in igraph. 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=1 and A=1.
`IGRAPH_BARABASI_PSUMTREE`	This algorithm uses a partial prefix-sum tree to generate the graph. It does not generate multiple edges and works for any power and A values.
`IGRAPH_BARABASI_PSUMTREE_MULTIPLE`	This algorithm also uses a partial prefix-sum tree to generate the graph. The

difference is, that now multiple edges are allowed. This method was implemented under the name `igraph_nonlinear_barabasi_game` before version 0.6.

`start_from`: Either a `NULL` pointer, or a graph. In the former case, the starting configuration is a clique of size *`m`*. In the latter case, the graph is a starting configuration. The graph must be non-empty, i.e. it must have at least one vertex. If a graph is supplied here and the *`outseq`* argument is also given, then *`outseq`* should only contain information on the vertices that are not in the *`start_from`* graph.

Returns:

Error code: `IGRAPH_EINVAL`: invalid *`n`*, *`m`*, *`A`* or *`outseq`* parameter.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

Example 12.3. File `examples/simple/igraph_barabasi_game.c`

Example 12.4. File `examples/simple/igraph_barabasi_game2.c`

`igraph_barabasi_aging_game` — Preferential attachment with aging of vertices.

```
igraph_error_t igraph_barabasi_aging_game(igraph_t *graph,
                                          igraph_int_t nodes,
                                          igraph_int_t m,
                                          const igraph_vector_int_t *outseq,
                                          igraph_bool_t outpref,
                                          igraph_real_t pa_exp,
                                          igraph_real_t aging_exp,
                                          igraph_int_t aging_bins,
                                          igraph_real_t zero_deg_appeal,
                                          igraph_real_t zero_age_appeal,
                                          igraph_real_t deg_coef,
                                          igraph_real_t age_coef,
                                          igraph_bool_t directed);
```

This game starts with one vertex (if *`nodes`* > 0). In each step a new node is added, and it is connected to *`m`* existing nodes. Existing nodes to connect to are chosen with probability dependent on their (in-)degree (`k`) and age (`l`). The degree-dependent part is `deg_coef * k^pa_exp + zero_deg_appeal`, while the age-dependent part is `age_coef * l^aging_exp + zero_age_appeal`, which are multiplied to obtain the final weight.

The age `l` is based on the number of vertices in the network and the *`aging_bins`* argument: the age of a node is incremented by 1 after each `floor(nodes / aging_bins) + 1` time steps.

Arguments:

`graph`: Pointer to an uninitialized graph object.

`nodes`: The number of vertices in the graph.

m: The number of edges to add in each time step. Ignored if *outseq* is a non-zero length vector.

outseq: The number of edges to add in each time step. If it is NULL or a zero-length vector then it is ignored and the *m* argument is used instead.

outpref: Boolean constant, whether the edges initiated by a vertex contribute to the probability to gain a new edge.

pa_exp: The exponent of the preferential attachment, a small positive number usually, the value 1 yields the classic linear preferential attachment.

aging_exp: The exponent of the aging, this is a negative number usually.

aging_bins: Integer constant, the number of age bins to use.

zero_deg_appeal: The degree dependent part of the attractiveness of the zero degree vertices.

zero_age_appeal: The age dependent part of the attractiveness of the vertices of age zero. This parameter is usually zero.

deg_coef: The coefficient for the degree.

age_coef: The coefficient for the age.

directed: Boolean constant, whether to generate a directed graph.

Returns:

Error code.

Time complexity: O((|V|+|V|/aging_bins)*log(|V|)+|E|). |V| is the number of vertices, |E| the number of edges.

igraph_recent_degree_game — Stochastic graph generator based on the number of incident edges a node has gained recently.

```
igraph_error_t igraph_recent_degree_game(igraph_t *graph, igraph_int_t nodes,
                                         igraph_real_t power,
                                         igraph_int_t time_window,
                                         igraph_int_t m,
                                         const igraph_vector_int_t *outseq,
                                         igraph_bool_t outpref,
                                         igraph_real_t zero_appeal,
                                         igraph_bool_t directed);
```

Arguments:

graph: Pointer to an uninitialized graph object.

nodes: The number of vertices in the graph, this is the same as the number of time steps.

power: The exponent, the probability that a node gains a new edge is proportional to the number of edges it has gained recently (in the last *window* time steps) to *power*.

time_window: Integer constant, the size of the time window to use to count the number of recent edges.

m: Integer constant, the number of edges to add per time step if the *outseq* parameter is a null pointer or a zero-length vector.

outseq: The number of edges to add in each time step. This argument is ignored if it is a null pointer or a zero length vector. In this case the constant *m* parameter is used.

outpref: Boolean constant, if true the edges originated by a vertex also count as recent incident edges. For most applications it is reasonable to set it to false.

zero_appeal: Constant giving the attractiveness of the vertices which haven't gained any edge recently.

directed: Boolean constant, whether to generate a directed graph.

Returns:

Error code.

Time complexity: O(|V|*log(|V|)+|E|), |V| is the number of vertices, |E| is the number of edges in the graph.

igraph_recent_degree_aging_game — Preferential attachment based on the number of edges gained recently, with aging of vertices.

```
igraph_error_t igraph_recent_degree_aging_game(igraph_t *graph,
                                       igraph_int_t nodes,
                                       igraph_int_t m,
                                       const igraph_vector_int_t *outseq,
                                       igraph_bool_t outpref,
                                       igraph_real_t pa_exp,
                                       igraph_real_t aging_exp,
                                       igraph_int_t aging_bins,
                                       igraph_int_t time_window,
                                       igraph_real_t zero_appeal,
                                       igraph_bool_t directed);
```

This game is very similar to `igraph_barabasi_aging_game()`, except that instead of the total number of incident edges the number of edges gained in the last *time_window* time steps are counted.

The degree dependent part of the attractiveness is given by k to the power of *pa_exp* plus *zero_appeal*; the age dependent part is l to the power to *aging_exp*. k is the number of edges gained in the last *time_window* time steps, l is the age of the vertex.

Arguments:

graph: Pointer to an uninitialized graph object.

nodes: The number of vertices in the graph.

m: The number of edges to add in each time step. If the *outseq* argument is not a null vector or a zero-length vector then it is ignored.

outseq: Vector giving the number of edges to add in each time step. If it is a null pointer or a zero-length vector then it is ignored and the *m* argument is used.

outpref: Boolean constant, if true the edges initiated by a vertex are also counted. Normally it is false.

pa_exp: The exponent for the preferential attachment.

aging_exp: The exponent for the aging, normally it is negative: old vertices gain edges with less probability.

aging_bins: Integer constant, the number of age bins to use.

time_window: The time window to use to count the number of incident edges for the vertices.

zero_appeal: The degree dependent part of the attractiveness for zero degree vertices.

directed: Boolean constant, whether to create a directed graph.

Returns:

Error code.

Time complexity: O((|V|+|V|/aging_bins)*log(|V|)+|E|). |V| is the number of vertices, |E| the number of edges.

igraph_lastcit_game — Simulates a citation network, based on time passed since the last citation.

```
igraph_error_t igraph_lastcit_game(igraph_t *graph,
                        igraph_int_t nodes, igraph_int_t edges_per_node,
                        igraph_int_t agebins,
                        const igraph_vector_t *preference,
                        igraph_bool_t directed);
```

This is a quite special stochastic graph generator, it models an evolving graph. In each time step a single vertex is added to the network and it cites a number of other vertices (as specified by the *edges_per_step* argument). The cited vertices are selected based on the last time they were cited. Time is measured by the addition of vertices and it is binned into *agebins* bins. So if the current time step is `t` and the last citation to a given `i` vertex was made in time step `t0`, then `(t-t0) / binwidth` is calculated where binwidth is `nodes/agebins + 1`, in the last expression '/' denotes integer division, so the fraction part is omitted.

The *preference* argument specifies the preferences for the citation lags, i.e. its first elements contains the attractivity of the very recently cited vertices, etc. The last element is special, it contains the attractivity of the vertices which were never cited. This element should be bigger than zero.

Note that this function generates networks with multiple edges if *edges_per_step* is bigger than one, call `igraph_simplify()` on the result to get rid of these edges.

Arguments:

graph: Pointer to an uninitialized graph object, the result will be stored here.

nodes: The number of vertices in the network.

edges_per_node: The number of edges to add in each time step.

agebins: The number of age bins to use.

preference: Pointer to an initialized vector of length `agebins + 1`. This contains the "attractivity" of the various age bins, the last element is the attractivity of the vertices which were never cited, and it should be greater than zero. It is a good idea to have all positive values in this vector. Preferences cannot be negative.

directed: Boolean constant, whether to create directed networks.

Returns:

Error code.

See also:

`igraph_barabasi_aging_game()`.

Time complexity: O(|V|*a+|E|*log|V|), |V| is the number of vertices, |E| is the total number of edges, a is the *agebins* parameter.

Growing random graph models

In growing random graphs, vertices are added iteratively, and connected based on various rules. Preferential attachment models are documented in their own section.

`igraph_growing_random_game` — Generates a growing random graph.

```
igraph_error_t igraph_growing_random_game(igraph_t *graph, igraph_int_t n,
                                          igraph_int_t m, igraph_bool_t directed,
                                          igraph_bool_t citation);
```

This function simulates a growing random graph. We start out with one vertex. In each step a new vertex is added and a number of new edges are also added. These graphs are known to be different from standard (not growing) random graphs.

Arguments:

graph: Uninitialized graph object.

n: The number of vertices in the graph.

m: The number of edges to add in a time step (i.e. after adding a vertex).

directed: Boolean, whether to generate a directed graph.

citation: Boolean, if `true`, the edges always originate from the most recently added vertex and are connected to a previous vertex.

Returns:

Error code: `IGRAPH_EINVAL`: invalid *n* or *m* parameter.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

igraph_callaway_traits_game — Simulates a growing network with vertex types.

```
igraph_error_t igraph_callaway_traits_game(igraph_t *graph, igraph_int_t nodes,
                                   igraph_int_t types, igraph_int_t edges_per_step
                                   const igraph_vector_t *type_dist,
                                   const igraph_matrix_t *pref_matrix,
                                   igraph_bool_t directed,
                                   igraph_vector_int_t *node_type_vec);
```

The different types of vertices prefer to connect other types of vertices with a given probability.

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.

References:

D. S. Callaway, J. E. Hopcroft, J. M. Kleinberg, M. E. J. Newman, and S. H. Strogatz, Are randomly grown graphs really random? Phys. Rev. E 64, 041902 (2001). https://doi.org/10.1103/PhysRevE.64.041902

Arguments:

graph: Pointer to an uninitialized graph.

nodes: The number of nodes in the graph.

types: Number of node types.

edges_per_step: The number of connections tried in each time step.

type_dist: Vector giving the distribution of the vertex types. If NULL, the distribution is assumed to be uniform.

pref_matrix: Matrix giving the connection probabilities for the vertex types.

directed: Whether to generate a directed graph.

node_type_vec: An initialized vector or NULL. If not NULL, the type of each node will be stored here.

Returns:

Error code.

Added in version 0.2.

Time complexity: O(|V|*k*log(|V|)), |V| is the number of vertices, k is *edges_per_step*.

igraph_establishment_game — Generates a graph with a simple growing model with vertex types.

```
igraph_error_t igraph_establishment_game(igraph_t *graph, igraph_int_t nodes,
```

```
                                    igraph_int_t types, igraph_int_t k,
                                    const igraph_vector_t *type_dist,
                                    const igraph_matrix_t *pref_matrix,
                                    igraph_bool_t directed,
                                    igraph_vector_int_t *node_type_vec);
```

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.

Arguments:

graph: Pointer to an uninitialized graph.

nodes: The number of vertices in the graph.

types: The number of vertex types.

k: The number of connections tried in each time step.

type_dist: Vector giving the distribution of vertex types. If NULL, the distribution is assumed to be uniform.

pref_matrix: Matrix giving the connection probabilities for different vertex types.

directed: Whether to generate a directed graph.

node_type_vec: An initialized vector or NULL. If not NULL, the type of each node will be stored here.

Returns:

Error code.

Added in version 0.2.

Time complexity: O(|V|*k*log(|V|)), |V| is the number of vertices and k is the *k* parameter.

igraph_cited_type_game — Simulates a citation based on vertex types.

```
igraph_error_t igraph_cited_type_game(igraph_t *graph, igraph_int_t nodes,
                            const igraph_vector_int_t *types,
                            const igraph_vector_t *pref,
                            igraph_int_t edges_per_step,
                            igraph_bool_t directed);
```

Function to create a network based on some vertex categories. This function creates a citation network: in each step a single vertex and *edges_per_step* citing edges are added. Nodes with different categories may have different probabilities to get cited, as given by the *pref* vector.

Note that this function might generate networks with multiple edges if *edges_per_step* is greater than one. You might want to call igraph_simplify() on the result to remove multiple edges.

Arguments:

graph: Pointer to an uninitialized graph object.

nodes: The number of vertices in the network.

types: Numeric vector giving the categories of the vertices, so it should contain *nodes* non-negative integer numbers. Types are numbered from zero.

pref: The attractivity of the different vertex categories in a vector. Its length should be the maximum element in *types* plus one (types are numbered from zero).

edges_per_step: Integer constant, the number of edges to add in each time step.

directed: Boolean constant, whether to create a directed network.

Returns:

Error code.

See also:

`igraph_citing_cited_type_game()` for a bit more general game.

Time complexity: O((|V|+|E|)log|V|), |V| and |E| are number of vertices and edges, respectively.

`igraph_citing_cited_type_game` — Simulates a citation network based on vertex types.

```
igraph_error_t igraph_citing_cited_type_game(igraph_t *graph, igraph_int_t nodes
                                  const igraph_vector_int_t *types,
                                  const igraph_matrix_t *pref,
                                  igraph_int_t edges_per_step,
                                  igraph_bool_t directed);
```

This game is similar to `igraph_cited_type_game()` but here the category of the citing vertex is also considered.

An evolving citation network is modeled here, a single vertex and its *edges_per_step* citation are added in each time step. The odds the a given vertex is cited by the new vertex depends on the category of both the citing and the cited vertex and is given in the *pref* matrix. The categories of the citing vertex correspond to the rows, the categories of the cited vertex to the columns of this matrix. I.e. the element in row `i` and column `j` gives the probability that a `j` vertex is cited, if the category of the citing vertex is `i`.

Note that this function might generate networks with multiple edges if *edges_per_step* is greater than one. You might want to call `igraph_simplify()` on the result to remove multiple edges.

Arguments:

graph: Pointer to an uninitialized graph object.

nodes: The number of vertices in the network.

types: A numeric vector of length *nodes*, containing the categories of the vertices. The categories are numbered from zero.

pref: The preference matrix, a square matrix is required, both the number of rows and columns should be the maximum element in *types* plus one (types are numbered from zero).

edges_per_step: Integer constant, the number of edges to add in each time step.

directed: Boolean constant, whether to create a directed network.

Returns:

Error code.

Time complexity: O((|V|+|E|)log|V|), |V| and |E| are number of vertices and edges, respectively.

igraph_forest_fire_game — Generates a network according to the “forest fire game”.

```
igraph_error_t igraph_forest_fire_game(igraph_t *graph, igraph_int_t nodes,
                            igraph_real_t fw_prob, igraph_real_t bw_factor,
                            igraph_int_t pambs, igraph_bool_t directed);
```

The forest fire model intends to reproduce the following network characteristics, observed in real networks:

- Heavy-tailed in- and out-degree distributions.
- Community structure.
- 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 numbers, `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.

See also: 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.

Note however, that 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.

Arguments:

graph: Pointer to an uninitialized graph object.

nodes: The number of vertices in the graph.

fw_prob: The forward burning probability.

bw_factor: The backward burning ratio. The backward burning probability is calculated as `bw_factor * fw_prob`.

pambs: The number of ambassador vertices.

directed: Whether to create a directed graph.

Returns:

Error code.

Time complexity: TODO.

Degree-constrained models

Random graph models with hard or soft degree constraints.

igraph_degree_sequence_game — Generates a random graph with a given degree sequence.

```
igraph_error_t igraph_degree_sequence_game(
        igraph_t *graph,
        const igraph_vector_int_t *out_degrees,
        const igraph_vector_int_t *in_degrees,
        igraph_degseq_t method);
```

This function generates random graphs with a prescribed degree sequence. Several sampling methods are available, which respect different constraints (simple graph or multigraphs, connected graphs, etc.), and provide different tradeoffs between performance and unbiased sampling. See Section 2.1 of Horvát and Modes (2021) for an overview of sampling techniques for graphs with fixed degrees.

References:

Fabien Viger, and Matthieu Latapy: Efficient and Simple Generation of Random Simple Connected Graphs with Prescribed Degree Sequence, Journal of Complex Networks 4, no. 1, pp. 15–37 (2015). https://doi.org/10.1093/comnet/cnv013.

Szabolcs Horvát, and Carl D Modes: Connectedness Matters: Construction and Exact Random Sampling of Connected Networks, Journal of Physics: Complexity 2, no. 1, pp. 015008 (2021). https://doi.org/10.1088/2632-072x/abced5.

Arguments:

graph: Pointer to an uninitialized graph object.

out_degrees: A vector of integers specifying the degree sequence for undirected graphs or the out-degree sequence for directed graphs.

in_degrees: A vector of integers specifying the in-degree sequence for directed graphs. For undirected graphs, it must be `NULL`.

method: The method to generate the graph. Possible values:

`IGRAPH_DEGSEQ_CONFIGURATION`: This method 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 direct-

ed 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.

`IGRAPH_DEGSEQ_CONFIGURATION_SIMPLE`

This method is identical to `IGRAPH_DEGSEQ_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.

`IGRAPH_DEGSEQ_FAST_HEUR_SIMPLE`

This method generates simple graphs. It is similar to `IGRAPH_DEGSEQ_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.

`IGRAPH_DEGSEQ_EDGE_SWITCHING_SIMPLE`

This is an MCMC sampler based on degree-preserving edge switches. It generates simple undirected or directed graphs. It uses `igraph_realize_degree_sequence()` to construct an initial graph, then rewires it using `igraph_rewire()`.

`IGRAPH_DEGSEQ_VL`

This 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://www-complexnetworks.lip6.fr/~latapy/FV/generation.html and the paper https://arxiv.org/abs/cs/0502085

Returns:

Error code: `IGRAPH_ENOMEM`: there is not enough memory to perform the operation. `IGRAPH_EINVAL`: invalid method parameter, or invalid in- and/or out-degree vectors. The degree vectors should be non-negative, *`out_deg`* should sum up to an even integer for undirected graphs; the length and sum of *`out_deg`* and *`in_deg`* should match for directed graphs.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges for `IGRAPH_DEGSEQ_CONFIGURATION` and `IGRAPH_DEGSEQ_EDGE_SWITCHING_SIMPLE`. The time complexity of the other modes is not known.

See also:

`igraph_is_graphical()` to determine if there exist graphs with a certain degree sequence; `igraph_erdos_renyi_game_gnm()` to generate graphs with a fixed number of edges, without any degree constraints; `igraph_chung_lu_game()` and `igraph_static_fitness_game()` to sample random graphs with a prescribed *expected* degree sequence (but variable actual degrees); `igraph_realize_degree_sequence()` and `igraph_realize_bipartite_degree_sequence()` to generate a single (non-random) graph with given degrees.

Example 12.5. File `examples/simple/igraph_degree_sequence_game.c`

`igraph_k_regular_game` — Generates a random graph where each vertex has the same degree.

```
igraph_error_t igraph_k_regular_game(igraph_t *graph,
                           igraph_int_t no_of_nodes, igraph_int_t k,
                           igraph_bool_t directed, igraph_bool_t multiple);
```

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.

Currently, this game simply uses `igraph_degree_sequence_game` with the `IGRAPH_DEGSEQ_CONFIGURATION` or the `IGRAPH_DEGSEQ_FAST_SIMPLE` method and appropriately constructed degree sequences. Thefore, it does not sample uniformly: while it can generate all k-regular graphs with the given number of vertices, it does not generate each one with the same probability.

Arguments:

`graph`:	Pointer to an uninitialized graph object.
`no_of_nodes`:	The number of nodes in the generated graph.
`k`:	The degree of each vertex in an undirected graph, or the out-degree and in-degree of each vertex in a directed graph.
`directed`:	Whether the generated graph will be directed.
`multiple`:	Whether to allow multiple edges in the generated graph.

Returns:

Error code: `IGRAPH_EINVAL`: invalid parameter; e.g., negative number of nodes, or odd number of nodes and odd k for undirected graphs. `IGRAPH_ENOMEM`: there is not enough memory for the operation.

Time complexity: O(|V|+|E|) if `multiple` is true, otherwise not known.

`igraph_rewire` — Randomly rewires a graph while preserving its degree sequence.

```
igraph_error_t igraph_rewire(igraph_t *graph, igraph_int_t n, igraph_edge_type_
```

This function generates a new graph based on the original one by randomly "rewriting" edges while preserving the original graph's degree sequence. The rewiring is done "in place", so no new graph will be allocated. If you would like to keep the original graph intact, use `igraph_copy()` beforehand. All graph attributes will be lost.

The rewiring is performed with degree-preserving edge switches: Two arbitrary edges are picked uniformly at random, namely `(a, b)` and `(c, d)`, then they are replaced by `(a, d)` and `(b, c)` if this preserves the constraints specified by *`mode`*.

Arguments:

`graph`: The graph object to be rewired.

`n`: Number of rewiring trials to perform.

`allowed_edge_types`: The types of edges that rewiring may create in the graph. See `igraph_edge_type_sw_t` for details. Currently, the following are implemented:

`IGRAPH_SIMPLE_SW` simple graphs (i.e. no self-loops or multi-edges allowed).

`IGRAPH_LOOPS_SW` single self-loops are allowed, but not multi-edges.

Multigraphs are not yet supported.

`stats`: Counts of the number of different operations performed by the algorithm are stored here.

Returns:

Error code:

`IGRAPH_EINVMODE` Invalid rewiring mode.

`IGRAPH_ENOMEM` Not enough memory for temporary data.

Time complexity: TODO.

`igraph_chung_lu_game` — Samples graphs from the Chung-Lu model.

```
igraph_error_t igraph_chung_lu_game(igraph_t *graph,
                                    const igraph_vector_t *out_weights,
                                    const igraph_vector_t *in_weights,
                                    igraph_bool_t loops,
                                    igraph_chung_lu_t variant);
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

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.

In the original Chung-Lu model, each pair of vertices `i` and `j` is connected with independent probability `p_ij = 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^out_k = sum_k w^in_k`. The connection probability between `i` and `j` is `p_ij = w^out_i w^in_j / 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 `w^out (S - w^in) / S` and `w^in (S - w^out) / S`, respectively. If the graph is undirected, then the expected degrees with and without self-loops are `w (S + w) / S` and `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.

The 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 = w_i w_j / S`, or `q_ij = w^out_i w^in_j / 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 `IGRAPH_CHUNG_LU_ORIGINAL`, `p_ij = min(q_ij, 1)`. The `IGRAPH_CHUNG_LU_MAXENT` variant, sometiems referred to a the generalized random graph, uses `p_ij = 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) = 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 produces graphs with the same degree sequence with the same probability. A third variant can be requested with `IGRAPH_CHUNG_LU_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).

References:

Chung F and Lu L: Connected components in a random graph with given degree sequences. Annals of Combinatorics 6, 125-145 (2002). https://doi.org/10.1007/PL00012580

Miller JC and Hagberg A: Efficient Generation of Networks with Given Expected Degrees (2011). https://doi.org/10.1007/978-3-642-21286-4_10

Park J and Newman MEJ: Statistical mechanics of networks. Physical Review E 70, 066117 (2004). https://doi.org/10.1103/PhysRevE.70.066117

Britton T, Deijfen M, Martin-Löf A: Generating Simple Random Graphs with Prescribed Degree Distribution. J Stat Phys 124, 1377–1397 (2006). https://doi.org/10.1007/s10955-006-9168-x

Norros I and Reittu H: On a conditionally Poissonian graph process. Advances in Applied Probability 38, 59–75 (2006). https://doi.org/10.1239/aap/1143936140

Bollobás B, Janson S, Riordan O: The phase transition in inhomogeneous random graphs. Random Struct Algorithms 31, 3–122 (2007). https://doi.org/10.1002/rsa.20168

Van Der Hofstad R: Critical behavior in inhomogeneous random graphs. Random Struct Algorithms 42, 480–508 (2013). https://doi.org/10.1002/rsa.20450

Arguments:

graph: Pointer to an uninitialized graph object.

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.

loops: 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 = w_i w_j / S`, the following formulations are available:

- `IGRAPH_CHUNG_LU_ORIGINAL` — the original Chung-Lu model, `p_ij = min(q_ij, 1)`.
- `IGRAPH_CHUNG_LU_MAXENT` — maximum entropy model with fixed expected degrees, `p_ij = q_ij / (1 + q_ij)`.
- `IGRAPH_CHUNG_LU_NR` — Norros and Reittu's model, `p_ij = 1 - exp(-q_ij)`.

Returns:

Error code.

See also:

`igraph_static_fitness_game()` implements a similar model with a sharp constraint on the number of edges; `igraph_degree_sequence_game()` samples random graphs with sharply specified degrees; `igraph_erdos_renyi_game_gnp()` creates random graphs with a fixed connection probability `p` between all vertex pairs.

Time complexity: O(|E| + |V|), linear in the number of edges.

`igraph_static_fitness_game` — Non-growing random graph with edge probabilities proportional to node fitness scores.

```
igraph_error_t igraph_static_fitness_game(igraph_t *graph, igraph_int_t no_of_e
                                  const igraph_vector_t *fitness_out, const igraph_
```

```
                                        igraph_edge_type_sw_t allowed_edge_types);
```

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.

The *expected* degree (though not the actual degree) of each vertex will be proportional to its fitness. This is exactly true when self-loops and multi-edges are allowed, and approximately true otherwise. If you need to generate a graph with an exact degree sequence, consider `igraph_degree_sequence_game()` and `igraph_realize_degree_sequence()` instead.

To generate random undirected graphs with a given expected degree sequence, set *`fitness_out`* (and in the directed case *`fitness_out`*) to the desired expected degrees, and *`no_of_edges`* to the corresponding edge count, i.e. half the sum of expected degrees in the undirected case, and the sum of out- or in-degrees in the directed case.

This model is similar to the better-known Chung-Lu model, implemented in igraph as `igraph_chung_lu_game()`, but with a sharply fixed edge count.

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 `igraph_static_power_law_game()` which generates the fitnesses for you with a given exponent.

Reference:

Goh K-I, Kahng B, Kim D: Universal behaviour of load distribution in scale-free networks. Phys Rev Lett 87(27):278701, 2001 https://doi.org/10.1103/PhysRevLett.87.278701.

Arguments:

`graph`:	Pointer to an uninitialized graph object.
`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 generated graph will be undirected. If not `NULL`, this argument specifies the in-fitness of each vertex.
`allowed_edge_types`:	Controls whether multi-edges and self-loops are allowed in the generated graph. See `igraph_edge_type_sw_t`.

Returns:

Error code: `IGRAPH_EINVAL`: invalid parameter `IGRAPH_ENOMEM`: there is not enough memory for the operation.

See also:

`igraph_static_power_law_game()`, `igraph_chung_lu_game()`, `igraph_degree_sequence_game()`

Time complexity: O(|V| + |E| log |E|).

igraph_static_power_law_game — Generates a non-growing random graph with expected power-law degree distributions.

```
igraph_error_t igraph_static_power_law_game(igraph_t *graph,
                                    igraph_int_t no_of_nodes, igraph_int_t no_of_e
                                    igraph_real_t exponent_out, igraph_real_t expo
                                    igraph_edge_type_sw_t allowed_edge_types,
                                    igraph_bool_t finite_size_correction);
```

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 `igraph_static_fitness_game()` with appropriately constructed fitness vectors. In particular, the fitness of vertex `i` is `i^(-alpha)`, where `alpha = 1/(gamma-1)` and `gamma` 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 `igraph_static_fitness_game()` 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+i0-1)^(-alpha)`, where `i0` 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.

References:

Goh K-I, Kahng B, Kim D: Universal behaviour of load distribution in scale-free networks. Phys Rev Lett 87(27):278701, 2001. https://doi.org/10.1103/PhysRevLett.87.278701

Chung F and Lu L: Connected components in a random graph with given degree sequences. Annals of Combinatorics 6, 125-145, 2002. https://doi.org/10.1007/PL00012580

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. https://doi.org/10.1103/PhysRevLett.103.135702

Arguments:

graph:	Pointer to an uninitialized graph object.
no_of_nodes:	The number of nodes in the generated graph.
no_of_edges:	The number of edges in the generated graph.
exponent_out:	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 `IGRAPH_INFINITY` here, you will get back an Erd#s-Rényi random network.
exponent_in:	If negative, the generated graph will be undirected. If greater than or equal to 2, this argument specifies the exponent of the in-de-

gree distribution. If non-negative but less than 2, an error will be generated.

allowed_edge_types: Controls whether multi-edges and self-loops are allowed in the generated graph. See igraph_edge_type_sw_t.

finite_size_correction: Whether to use the proposed finite size correction of Cho et al.

Returns:

Error code: IGRAPH_EINVAL: invalid parameter IGRAPH_ENOMEM: there is not enough memory for the operation.

Time complexity: O(|V| + |E| log |E|).

Edge rewiring models

igraph_watts_strogatz_game — The Watts-Strogatz small-world model.

```
igraph_error_t igraph_watts_strogatz_game(
        igraph_t *graph, igraph_int_t dim,
        igraph_int_t size, igraph_int_t nei,
        igraph_real_t p,
        igraph_edge_type_sw_t allowed_edge_types);
```

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.

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.

Reference:

Duncan J Watts and Steven H Strogatz: Collective dynamics of “small world” networks, Nature 393, 440-442, 1998. https://doi.org/10.1038/30918

Arguments:

graph: The graph to initialize.

dim: The dimension of the lattice.

size: The size of the lattice along each dimension.

nei: The size of the neighborhood for each vertex. This is the same as the *order* argument of igraph_connect_neighborhood().

p: The rewiring probability. A real number between zero and one (inclusive).

allowed_edge_types: Controls whether multi-edges and self-loops are allowed in the generated graph. See igraph_edge_type_sw_t.

Returns:

Error code.

See also:

`igraph_square_lattice()`, `igraph_connect_neighborhood()` and `igraph_rewire_edges()` can be used if more flexibility is needed, e.g. a different type of lattice.

Time complexity: O(|V|*d^o+|E|), |V| and |E| are the number of vertices and edges, d is the average degree, o is the *`nei`* argument.

`igraph_rewire_edges` — Rewires the edges of a graph with constant probability.

```
igraph_error_t igraph_rewire_edges(igraph_t *graph, igraph_real_t prob,
                                   igraph_edge_type_sw_t allowed_edge_types);
```

This function rewires the edges of a graph with a constant probability. More precisely each end point of each edge is rewired to a uniformly randomly chosen vertex with constant probability *`prob`*.

Note that this function modifies the input *`graph`*, call `igraph_copy()` if you want to keep it.

Arguments:

`graph`:	The input graph, this will be rewired, it can be directed or undirected.
`prob`:	The rewiring probability a constant between zero and one (inclusive).
`allowed_edge_types`:	Controls whether multi-edges and self-loops are allowed in the new graph. See `igraph_edge_type_sw_t`.

Returns:

Error code.

See also:

`igraph_watts_strogatz_game()` uses this function for the rewiring.

Time complexity: O(|V|+|E|).

`igraph_rewire_directed_edges` — Rewires the chosen endpoint of directed edges.

```
igraph_error_t igraph_rewire_directed_edges(igraph_t *graph, igraph_real_t prob,
                                    igraph_bool_t loops, igraph_neimode_t mode);
```

This function rewires either the start or end of directed edges in a graph with a constant probability. Correspondingly, either the in-degree sequence or the out-degree sequence of the graph will be preserved.

Note that this function modifies the input *`graph`*, call `igraph_copy()` if you want to keep it.

This function can produce multiple edges between two vertices.

Arguments:

graph: The input graph, this will be rewired, it can be directed or undirected. If it is undirected or *mode* is set to IGRAPH_ALL, `igraph_rewire_edges()` will be called.

prob: The rewiring probability, a constant between zero and one (inclusive).

loops: Boolean, whether loop edges are allowed in the new graph, or not.

mode: The endpoints of directed edges to rewire. It is ignored for undirected graphs. Possible values:

`IGRAPH_OUT` rewire the end of each directed edge

`IGRAPH_IN` rewire the start of each directed edge

`IGRAPH_ALL` rewire both endpoints of each edge

Returns:

Error code.

See also:

`igraph_rewire_edges()`, `igraph_rewire()`

Time complexity: O(|E|).

Other random graphs

`igraph_grg_game` — Generates a geometric random graph.

```
igraph_error_t igraph_grg_game(igraph_t *graph, igraph_int_t nodes,
                    igraph_real_t radius, igraph_bool_t torus,
                    igraph_vector_t *x, igraph_vector_t *y);
```

A geometric random graph is created by dropping points (i.e. vertices) randomly on the unit square and then connecting all those pairs which are strictly less than `radius` apart in Euclidean distance.

Original code contributed by Keith Briggs, thanks Keith.

Arguments:

graph: Pointer to an uninitialized graph object.

nodes: The number of vertices in the graph.

radius: The radius within which the vertices will be connected.

torus: Boolean constant. If true, periodic boundary conditions will be used, i.e. the vertices are assumed to be on a torus instead of a square.

x: An initialized vector or `NULL`. If not `NULL`, the points' x coordinates will be returned here.

y: An initialized vector or `NULL`. If not `NULL`, the points' y coordinates will be returned here.

Returns:

Error code.

Time complexity: TODO, less than O(|V|^2+|E|).

Example 12.6. File `examples/simple/igraph_grg_game.c`

`igraph_dot_product_game` — Generates a random dot product graph.

```
igraph_error_t igraph_dot_product_game(igraph_t *graph, const igraph_matrix_t *v
                              igraph_bool_t directed);
```

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.

See also Christine Leigh Myers Nickel: Random dot product graphs, a model for social networks. Dissertation, Johns Hopkins University, Maryland, USA, 2006.

Arguments:

`graph`: The output graph is stored here.

`vecs`: A matrix in which each latent position vector is a column. 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.

`directed`: Should the generated graph be directed?

Returns:

Error code.

Time complexity: O(n*n*m), where n is the number of vertices, and m is the length of the latent vectors.

See also:

`igraph_rng_sample_dirichlet()`, `igraph_rng_sample_sphere_volume()`, `igraph_rng_sample_sphere_surface()` for functions to generate the latent vectors.

`igraph_simple_interconnected_islands_game` — Generates a random graph made of several interconnected islands, each island being a random graph.

```
igraph_error_t igraph_simple_interconnected_islands_game(
        igraph_t *graph,
        igraph_int_t islands_n,
        igraph_int_t islands_size,
        igraph_real_t islands_pin,
```

```
        igraph_int_t n_inter);
```

All islands are of the same size. Within an island, each edge is generated with the same probability. A fixed number of additional edges are then generated for each unordered pair of islands to connect them. The generated graph is guaranteed to be simple.

Arguments:

graph: Pointer to an uninitialized graph object.

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 within islands.

n_inter: The number of edges to create between two islands. It may be larger than *islands_size* squared, but in this case it is assumed to be *islands_size* squared.

Returns:

Error code: IGRAPH_EINVAL: invalid parameter IGRAPH_ENOMEM: there is not enough memory for the operation.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges in the graph.

igraph_tree_game — Generates a random tree with the given number of nodes.

```
igraph_error_t igraph_tree_game(igraph_t *graph, igraph_int_t n, igraph_bool_t
```

This function samples uniformly from the set of labelled trees, i.e. it generates each labelled tree with the same probability.

Note that for n=0, the null graph is returned, which is not considered to be a tree by igraph_is_tree().

Arguments:

graph: Pointer to an uninitialized graph object.

n: The number of nodes in the tree.

directed: Whether to create a directed tree. The edges are oriented away from the root.

method: The algorithm to use to generate the tree. Possible values:

IGRAPH_RANDOM_TREE_PRUFER	This algorithm samples Prüfer sequences uniformly, then converts them to trees. Directed trees are not currently supported.
IGRAPH_RANDOM_LERW	This algorithm effectively performs a loop-erased random walk on the complete graph to uniformly sample its spanning trees (Wilson's algorithm).

Returns:

Error code: IGRAPH_ENOMEM: there is not enough memory to perform the operation. IGRAPH_EINVAL: invalid tree size

See also:

igraph_from_prufer()

Common types and constants

igraph_edge_type_sw_t — What types of non-simple edges to allow?

```
typedef unsigned int igraph_edge_type_sw_t;
```

This type is used with multiple functions to specify what types of non-simple edges to allow, create or consider a graph. The constants below are treated as "switches" that can be turned on individually and combined using the bitwise-or operator. For example, IGRAPH_LOOPS_SW allows only self-loops but not multi-edges, while IGRAPH_LOOPS_SW | IGRAPH_MULTI_SW allows both.

Values:

IGRAPH_SIMPLE_SW: A shorthand for simple graphs only, which is the default assumption.

IGRAPH_LOOPS_SW: Allow or consider self-loops.

IGRAPH_MULTI_SW: Allow or consider multi-edges.

Chapter 13. Bipartite, i.e. two-mode graphs

Bipartite networks in igraph

A bipartite network contains two kinds of vertices and connections are only possible between two vertices of different kinds. There are many natural examples, e.g. movies and actors as vertices and a movie is connected to all participating actors, etc.

igraph does not have direct support for bipartite networks, at least not at the C language level. In other words the igraph_t structure does not contain information about the vertex types. The C functions for bipartite networks usually have an additional input argument to graph, called `types`, a boolean vector giving the vertex types.

Most functions creating bipartite networks are able to create this extra vector, you just need to supply an initialized boolean vector to them.

Create two-mode networks

`igraph_create_bipartite` — Create a bipartite graph.

```
igraph_error_t igraph_create_bipartite(igraph_t *graph, const igraph_vector_boo
                            const igraph_vector_int_t *edges,
                            igraph_bool_t directed);
```

This is a simple wrapper function to create a bipartite graph. It does a little more than `igraph_create()`, e.g. it checks that the graph is indeed bipartite with respect to the given *`types`* vector. If there is an edge connecting two vertices of the same kind, then an error is reported.

Arguments:

`graph`: Pointer to an uninitialized graph object, the result is created here.

`types`: Boolean vector giving the vertex types. The length of the vector defines the number of vertices in the graph.

`edges`: Vector giving the edges of the graph. The highest vertex ID in this vector must be smaller than the length of the *`types`* vector.

`directed`: Boolean, whether to create a directed graph.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

Example 13.1. File `examples/simple/igraph_bipartite_create.c`

`igraph_full_bipartite` — Creates a complete bipartite graph.

```
igraph_error_t igraph_full_bipartite(igraph_t *graph,
                           igraph_vector_bool_t *types,
                           igraph_int_t n1, igraph_int_t n2,
                           igraph_bool_t directed,
                           igraph_neimode_t mode);
```

A bipartite network contains two kinds of vertices and connections are only possible between two vertices of different kind. There are many natural examples, e.g. movies and actors as vertices and a movie is connected to all participating actors, etc.

igraph does not have direct support for bipartite networks, at least not at the C language level. In other words the igraph_t structure does not contain information about the vertex types. The C functions for bipartite networks usually have an additional input argument to graph, called *types*, a boolean vector giving the vertex types.

Most functions creating bipartite networks are able to create this extra vector, you just need to supply an initialized boolean vector to them.

Arguments:

graph: Pointer to an uninitialized graph object, the graph will be created here.

types: Pointer to a boolean vector. If not a null pointer, then the vertex types will be stored here.

n1: Integer, the number of vertices of the first kind.

n2: Integer, the number of vertices of the second kind.

directed: Boolean, whether to create a directed graph.

mode: A constant that gives the type of connections for directed graphs. If `IGRAPH_OUT`, then edges point from vertices of the first kind to vertices of the second kind; if `IGRAPH_IN`, then the opposite direction is realized; if `IGRAPH_ALL`, then mutual edges will be created.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

See also:

`igraph_full()` for non-bipartite complete graphs, `igraph_full_multipartite()` for complete multipartite graphs.

`igraph_bipartite_game_gnm` — Generate a random bipartite graph with a fixed number of edges.

```
igraph_error_t igraph_bipartite_game_gnm(
        igraph_t *graph,
        igraph_vector_bool_t *types,
        igraph_int_t n1, igraph_int_t n2, igraph_int_t m,
        igraph_bool_t directed, igraph_neimode_t mode,
        igraph_edge_type_sw_t allowed_edge_types,
```

```
        igraph_bool_t edge_labeled);
```

The `G(n1, n2, m)` model uniformly samples bipartite graphs with *`n1`* bottom vertices and *`n2`* top vertices, and precisely *`m`* edges.

Arguments:

`graph`: Pointer to an uninitialized igraph graph, the result is stored here.

`types`: Pointer to an initialized boolean vector, or a null pointer. If not a null pointer, then the vertex types are stored here. Bottom vertices come first, *`n1`* of them, then *`n2`* top vertices.

`n1`: The number of bottom vertices.

`n2`: The number of top vertices.

`m`: The number of edges.

`directed`: Boolean, whether to generate a directed graph. See also the *`mode`* argument.

`mode`: Specifies how to direct the edges in directed graphs. If it is `IGRAPH_OUT`, then directed edges point from bottom vertices to top vertices. If it is `IGRAPH_IN`, edges point from top vertices to bottom vertices. `IGRAPH_OUT` and `IGRAPH_IN` do not generate mutual edges. If this argument is `IGRAPH_ALL`, then each edge direction is considered independently and mutual edges might be generated. This argument is ignored for undirected graphs. *

`allowed_edge_types`: The types of edges to allow in the graph.

`IGRAPH_SIMPLE_SW` simple graph (i.e. no multi-edges allowed).

`IGRAPH_MULTI_SW` multi-edges are allowed

`edge_labeled`: If true, the sampling is done uniformly from the set of ordered edge lists. See `igraph_bipartite_iea_game()` for more information. Set this to `false` to select the classic Erd#s-Rényi model. The constants `IGRAPH_EDGE_UNLABELED` and `IGRAPH_EDGE_LABELED` may be used instead of `false` and `true` for better readability.

Returns:

Error code.

See also:

`igraph_erdos_renyi_game_gnm()` for the unipartite version, `igraph_bipartite_game_gnp()` for the `G(n1, n2, p)` model.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

`igraph_bipartite_game_gnp` — Generates a random bipartite graph with a fixed connection probability.

```
igraph_error_t igraph_bipartite_game_gnp(
        igraph_t *graph,
        igraph_vector_bool_t *types,
        igraph_int_t n1, igraph_int_t n2, igraph_real_t p,
        igraph_bool_t directed, igraph_neimode_t mode,
        igraph_edge_type_sw_t allowed_edge_types,
        igraph_bool_t edge_labeled);
```

In the `G(n1, n2, p)` model, every possible edge between the *n1* bottom vertices and *n2* top vertices is realized independently with probability *p*. This is equivalent to a maximum entropy model with a constraint on the *expected* total edge count. This view allows a multigraph extension, in which case *is* interpreted as the expected number of edges between any vertex pair. See `igraph_erdos_renyi_game_gnp()` for more details.

Arguments:

graph: Pointer to an uninitialized igraph graph, the result is stored here.

types: Pointer to an initialized boolean vector, or a null pointer. If not `NULL`, then the vertex types are stored here. Bottom vertices come first, *n1* of them, then *n2* top vertices.

n1: The number of bottom vertices.

n2: The number of top vertices.

p: The expected number of edges between any vertex pair. When multi-edges are disallowed, this is equivalent to the probability of having a connection between any two vertices.

directed: Whether to generate a directed graph. See also the *mode* argument.

mode: Specifies how to direct the edges in directed graphs. If it is `IGRAPH_OUT`, then directed edges point from bottom vertices to top vertices. If it is `IGRAPH_IN`, edges point from top vertices to bottom vertices. `IGRAPH_OUT` and `IGRAPH_IN` do not generate mutual edges. If this argument is `IGRAPH_ALL`, then each edge direction is considered independently and mutual edges might be generated. This argument is ignored for undirected graphs. *

allowed_edge_types: The types of edges to allow in the graph.

`IGRAPH_SIMPLE_SW` simple graph (i.e. no multi-edges allowed).

`IGRAPH_MULTI_SW` multi-edges are allowed

edge_labeled: If true, the model is defined over the set of ordered edge lists, i.e. over the set of edge-labeled graphs. Set it to `false` to select the classic bipartite Erd#s-Rényi model. The constants `IGRAPH_EDGE_UNLABELED` and `IGRAPH_EDGE_LABELED` may be used instead of `false` and `true` for better readability.

Returns:

Error code.

See also:

`igraph_erdos_renyi_game_gnp()` for the unipartite version, `igraph_bipartite_game_gnm()` for the `G(n1, n2, m)` model.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

igraph_bipartite_iea_game — Generates a random bipartite multigraph through independent edge assignment.

```
igraph_error_t igraph_bipartite_iea_game(
        igraph_t *graph, igraph_vector_bool_t *types,
        igraph_int_t n1, igraph_int_t n2, igraph_int_t m,
        igraph_bool_t directed, igraph_neimode_t mode);
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

This model generates random multigraphs with *n1* bottom vertices, *n2* top vertices and *m* edges through independent edge assignment (IEA). Each of the *m* edges is assigned uniformly at random to a vertex pair, independently of each other.

This model does not sample multigraphs uniformly. Undirected graphs are generated with probability proportional to

`(prod_(i<j) A_ij !)^(-1)`,

where A denotes the adjacency matrix. 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.

Arguments:

graph: Pointer to an uninitialized igraph graph, the result is stored here.

types: Pointer to an initialized boolean vector, or a NULL pointer. If not NULL, then the vertex types are stored here. Bottom vertices come first, *n1* of them, then *n2* top vertices.

n1: The number of bottom vertices.

n2: The number of top vertices.

m: The number of edges.

directed: Whether to generate a directed graph. See also the *mode* argument.

mode: Specifies how to direct the edges in directed graphs. If it is IGRAPH_OUT, then directed edges point from bottom vertices to top vertices. If it is IGRAPH_IN, edges point from top vertices to bottom vertices. IGRAPH_OUT and IGRAPH_IN do not generate mutual edges. If this argument is IGRAPH_ALL, then each edge direction is considered independently and mutual edges might be generated. This argument is ignored for undirected graphs.

Returns:

Error code.

See also:

`igraph_iea_game()` for the unipartite version; `igraph_bipartite_game_gnm()` to uniformly sample bipartite graphs with a given number of vertices and edges.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

Bipartite adjacency matrices

`igraph_biadjacency` — Creates a bipartite graph from a bipartite adjacency matrix.

```
igraph_error_t igraph_biadjacency(
        igraph_t *graph,
        igraph_vector_bool_t *types,
        const igraph_matrix_t *biadjmatrix,
        igraph_bool_t directed,
        igraph_neimode_t mode,
        igraph_bool_t multiple);
```

A bipartite (or two-mode) graph contains two types of vertices and edges always connect vertices of different types. A bipartite adjacency matrix is an *n* x *m* matrix, *n* and *m* are the number of vertices of the two types, respectively. Nonzero elements in the matrix denote edges between the two corresponding vertices.

This function 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 as many edges are created between two vertices as the corresponding matrix element. When *`multiple`* is set to `true`, matrix elements should be whole numbers. Otherwise their fractional part will be discarded.

Arguments:

`graph`: Pointer to an uninitialized graph object.

`types`: Pointer to an initialized boolean vector, or a null pointer. If not a null pointer, then the vertex types are stored here. It is resized as needed.

`biadjmatrix`: The bipartite adjacency matrix that serves as an input to this function.

`directed`: Specifies whether to create an undirected or a directed graph.

`mode`: Specifies the direction of the edges in a directed graph. If `IGRAPH_OUT`, then edges point from vertices of the first kind (corresponding to rows) to vertices of the second kind (corresponding to columns); if `IGRAPH_IN`, then the opposite direction is realized; if `IGRAPH_ALL`, then mutual edges will be created.

`multiple`: Whether to interpret matrix entries as edge multiplicities, see details above.

Returns:

Error code.

Time complexity: O(n*m), the size of the bipartite adjacency matrix.

igraph_weighted_biadjacency — Creates a bipartite graph from a weighted bipartite adjacency matrix.

```
igraph_error_t igraph_weighted_biadjacency(
        igraph_t *graph,
        igraph_vector_bool_t *types,
        igraph_vector_t *weights,
        const igraph_matrix_t *biadjmatrix,
        igraph_bool_t directed,
        igraph_neimode_t mode);
```

A bipartite (or two-mode) graph contains two types of vertices and edges always connect vertices of different types. A bipartite adjacency matrix is an *n* x *m* matrix, *n* and *m* are the number of vertices of the two types, respectively. Nonzero elements in the matrix denote edges between the two corresponding vertices.

Arguments:

graph: Pointer to an uninitialized graph object.

types: Pointer to an initialized boolean vector, or a null pointer. If not a null pointer, then the vertex types are stored here. It is resized as needed.

weights: Pointer to an initialized vector, the weights will be stored here.

biadjmatrix: The bipartite adjacency matrix that serves as an input to this function.

directed: Specifies whether to create an undirected or a directed graph.

mode: Specifies the direction of the edges in a directed graph. If `IGRAPH_OUT`, then edges point from vertices of the first kind (corresponding to rows) to vertices of the second kind (corresponding to columns); if `IGRAPH_IN`, then the opposite direction is realized; if `IGRAPH_ALL`, then mutual edges will be created.

Returns:

Error code.

Time complexity: O(n*m), the size of the bipartite adjacency matrix.

igraph_get_biadjacency — Converts a bipartite graph into a bipartite adjacency matrix.

```
igraph_error_t igraph_get_biadjacency(
    const igraph_t *graph, const igraph_vector_bool_t *types,
    const igraph_vector_t *weights,
    igraph_matrix_t *res, igraph_vector_int_t *row_ids,
    igraph_vector_int_t *col_ids
);
```

In a bipartite adjacency matrix `A`, element `A_ij` gives the number of edges between the `i`th vertex of the first partition and the `j`th vertex of the second partition.

If the graph contains edges within the same partition, this function issues a warning.

Arguments:

graph: The input graph, edge directions are ignored.

types: Boolean vector containing the vertex types. Vertices belonging to the first partition have type `false`, the one in the second partition type `true`.

weights: A vector specifying a weight for each edge or `NULL`. If `NULL`, all edges are assumed to have weight 1.

res: Pointer to an initialized matrix, the result is stored here. An element of the matrix gives the number of edges (irrespectively of their direction), or sum of edge weights, between the two corresponding vertices. The rows will correspond to vertices with type `false`, the columns correspond to vertices with type `true`.

row_ids: Pointer to an initialized vector or `NULL`. If not a null pointer, then the IDs of vertices with type `false` are stored here, with the same ordering as the rows of the biadjacency matrix.

col_ids: Pointer to an initialized vector or `NULL`. If not a null pointer, then the IDs of vertices with type `true` are stored here, with the same ordering as the columns of the biadjacency matrix.

Returns:

Error code.

Time complexity: O(|E|) where |E| is the number of edges.

See also:

`igraph_biadjacency()` for the opposite operation.

Project two-mode graphs

`igraph_bipartite_projection_size` — Calculate the number of vertices and edges in the bipartite projections.

```
igraph_error_t igraph_bipartite_projection_size(const igraph_t *graph,
                                        const igraph_vector_bool_t *types,
                                        igraph_int_t *vcount1,
                                        igraph_int_t *ecount1,
                                        igraph_int_t *vcount2,
                                        igraph_int_t *ecount2);
```

This function calculates the number of vertices and edges in the two projections of a bipartite network. This is useful if you have a big bipartite network and you want to estimate the amount of memory you would need to calculate the projections themselves.

Arguments:

graph: The input graph.

types: Boolean vector giving the vertex types of the graph.

vcount1: Pointer to an `igraph_int_t`, the number of vertices in the first projection is stored here. May be `NULL` if not needed.

ecount1: Pointer to an `igraph_int_t`, the number of edges in the first projection is stored here. May be `NULL` if not needed.

vcount2: Pointer to an `igraph_int_t`, the number of vertices in the second projection is stored here. May be `NULL` if not needed.

ecount2: Pointer to an `igraph_int_t`, the number of edges in the second projection is stored here. May be `NULL` if not needed.

Returns:

Error code.

See also:

`igraph_bipartite_projection()` to calculate the actual projection.

Time complexity: O(|V|*d^2+|E|), |V| is the number of vertices, |E| is the number of edges, d is the average (total) degree of the graphs.

`igraph_bipartite_projection` — Create one or both projections of a bipartite (two-mode) network.

```
igraph_error_t igraph_bipartite_projection(const igraph_t *graph,
                                           const igraph_vector_bool_t *types,
                                           igraph_t *proj1,
                                           igraph_t *proj2,
                                           igraph_vector_int_t *multiplicity1,
                                           igraph_vector_int_t *multiplicity2,
                                           igraph_int_t probe1);
```

Creates one or both projections of a bipartite graph.

A graph is called bipartite if its vertices can be partitioned into two sets, V1 and V2, so that connections only run between V1 and V2, but not within V1 or within V2. The *types* parameter specifies which vertex should be considered a member of one or the other partition. The projection to V1 has vertex set V1, and two vertices are connected if they have at least one common neighbour in V2. The number of common neighbours is returned in *multiplicity1*, if requested.

Arguments:

graph: The bipartite input graph. Directedness of the edges is ignored.

types: Boolean vector giving the vertex types of the graph.

proj1: Pointer to an uninitialized graph object, the first projection will be created here. It a null pointer, then it is ignored, see also the *probe1* argument.

proj2: Pointer to an uninitialized graph object, the second projection is created here, if it is not a null pointer. See also the *probe1* argument.

multiplicity1: Pointer to a vector, or a null pointer. If not the latter, then the multiplicity of the edges is stored here. 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.

multiplicity2: The same as `multiplicity1`, but for the other projection.

probe1: This argument can be used to specify the order of the projections in the resulting list. When it is non-negative, then it is considered as a vertex ID and the projection containing this vertex will be the first one in the result. Setting this argument to a non-negative value implies that `proj1` must be a non-null pointer. If you don't care about the ordering of the projections, pass -1 here.

Returns:

Error code.

See also:

`igraph_bipartite_projection_size()` to calculate the number of vertices and edges in the projections, without creating the projection graphs themselves.

Time complexity: O(|V|*d^2+|E|), |V| is the number of vertices, |E| is the number of edges, d is the average (total) degree of the graphs.

Other operations on bipartite graphs

`igraph_is_bipartite` — Check whether a graph is bipartite.

```
igraph_error_t igraph_is_bipartite(const igraph_t *graph,
                        igraph_bool_t *res,
                        igraph_vector_bool_t *types);
```

This function checks whether a graph is 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 be 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.

Arguments:

graph: The input graph.

res: Pointer to a boolean, the result is stored here.

types: Pointer to an initialized boolean vector, or a null pointer. If not a null pointer and a mapping was found, then it is stored here. If not a null pointer, but no mapping was found, the contents of this vector is invalid.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

See also:

igraph_is_bipartite_coloring() to determine if all edges connect vertices of different types, given a specific type vector.

Chapter 14. Spatial graphs

Metrics

`igraph_metric_t` — Metric functions for use with spatial computation.

```
typedef enum {
    IGRAPH_METRIC_EUCLIDEAN = 0,
    IGRAPH_METRIC_L2 = IGRAPH_METRIC_EUCLIDEAN,
    IGRAPH_METRIC_MANHATTAN = 1,
    IGRAPH_METRIC_L1 = IGRAPH_METRIC_MANHATTAN
} igraph_metric_t;
```

Values:

`IGRAPH_METRIC_EUCLIDEAN`: The Euclidean distance, i.e. L2 metric.

`IGRAPH_METRIC_MANHATTAN`: The Manhattan distance, i.e. L1 metric.

Spatial graph generators

`igraph_delaunay_graph` — Computes the Delaunay graph of a spatial point set.

```
igraph_error_t igraph_delaunay_graph(igraph_t *graph, const igraph_matrix_t *po
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

This function constructs the graph corresponding to the Delaunay triangulation of an n-dimensional spatial point set.

The current implementation uses Qhull.

Reference:

Barber, C. Bradford, David P. Dobkin, and Hannu Huhdanpaa. The Quickhull Algorithm for Convex Hulls. ACM Transactions on Mathematical Software 22, no. 4 (1996): 469–83. https://doi.org/10.1145/235815.235821.

Arguments:

`graph`: A pointer to the graph that will be created.

`points`: A matrix containing the points that will be used to create the graph. Each row is a point, dimensionality is inferred from the column count. There must not be duplicate points.

Returns:

Error code.

Time complexity: According to Theorem 3.2 in the Qhull paper, O(n log n) for d <= 3 and O(n^floor(d/2) / floor(d/2)!) where n is the number of points and d is the dimensionality of the point set.

`igraph_nearest_neighbor_graph` — Computes the nearest neighbor graph for a spatial point set.

```
igraph_error_t igraph_nearest_neighbor_graph(igraph_t *graph,
        const igraph_matrix_t *points,
        igraph_metric_t metric,
        igraph_int_t k,
        igraph_real_t cutoff,
        igraph_bool_t directed);
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

This function constructs the *k* nearest neighbor graph of a given point set. Each point is connected to at most *k* spatial neighbors within a radius of *cutoff*.

Arguments:

graph: A pointer to the graph that will be created.

points: A matrix containing the points that will be used to create the graph. Each row is a point, dimensionality is inferred from the column count.

metric: The distance metric to use. See `igraph_metric_t`.

k: At most how many neighbors will be added for each vertex, set to a negative value to ignore.

cutoff: Maximum distance at which connections will be made, set to a negative value or `IGRAPH_INFINITY` to ignore.

directed: Whether to create a directed graph.

Returns:

Error code.

Time complexity: O(n log(n)) where n is the number of points.

`igraph_gabriel_graph` — The Gabriel graph of a point set.

```
igraph_error_t igraph_gabriel_graph(igraph_t *graph, const igraph_matrix_t *poi
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

In the Gabriel graph of a point set, two points A and B are connected if there is no other point C within the closed ball of which AB is a diameter. The Gabriel graph is connected, and in 2D it is planar. igraph supports computing the Gabriel graph of arbitrary dimensional point sets.

The Gabriel graph is a special case of lune-based and circle-based β-skeletons with `#=1`.

Arguments:

`graph`: A pointer to the graph to be created.

`points`: The point set that will be used. Each row is a point, dimensionality is inferred from column count.

Returns:

Error Code.

See also:

The Gabriel graph is a special case of `igraph_lune_beta_skeleton()` and `igraph_circle_beta_skeleton()` where `# = 1`.

Time Complexity: Around O(n^floor(d/2) log n), where n is the number of points and d is the dimensionality of the point set.

`igraph_relative_neighborhood_graph` — The relative neighborhood graph of a point set.

```
igraph_error_t igraph_relative_neighborhood_graph(igraph_t *graph, const igraph
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

The relative neighborhood graph is constructed from a set of points in space. Two points A and B are connected if and only if there is no other point C so that AC < AB and BC < AB, with the inequalities being strict.

Most authors define the relative neighborhood graph to coincide with a lune-based β-skeleton for `# = 2`. In igraph, there is a subtle difference: the `# = 2` skeleton connects points A and B when there is no point C so that AC <= AB and BC <= AB. Therefore, three points forming an equilateral triangle are connected in the relative neighborhood graph, but disconnected in the `# = 2` skeleton.

With these definitions, the relative neighborhood graph is always connected, while the `# = 2` skeleton is always triangle-free.

Arguments:

graph: A pointer to the graph that will be created.

points: The point set that will be used, each row is a point. Dimensionality is inferred from the column number.

Returns:

Error code.

See also:

`igraph_lune_beta_skeleton()` to compute the lune based β-skeleton for `# = 2` or other β values.

Time Complexity: Around O(n^floor(d/2) log n), where n is the number of points and d is the dimensionality of the point set.

`igraph_lune_beta_skeleton` — The lune based β-skeleton of a spatial point set.

```
igraph_error_t igraph_lune_beta_skeleton(igraph_t *graph, const igraph_matrix_t
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

This function constructs the lune-based β-skeleton of an n-dimensional spatial point set.

A larger β results in a larger region, and a sparser graph. Values of $\beta < 1$ are only supported in 2D, and are considerably slower.

The Gabriel graph is a special case of beta skeleton where `# = 1`.

The Relative Neighborhood graph is a special case of beta skeleton where β approaches

Arguments:

graph: A pointer to the graph that will be created.

points: A matrix containing the points that will be used to create the graph. Each row is a point, dimensionality is inferred from the column count.

Returns:

Error code.

Time Complexity: Around O(n^floor(d/2) log n), where n is the number of points and d is the dimensionality of the point set.

`igraph_circle_beta_skeleton` — The circle based β-skeleton of a 2D spatial point set.

```
igraph_error_t igraph_circle_beta_skeleton(igraph_t *graph, const igraph_matrix
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

This function constructs the circle based β-skeleton of a 2D spatial point set.

A larger *beta* value results in a larger region, and a sparser graph. Values of beta < 1 are considerably slower

Arguments:

graph: A pointer to the graph that will be created.

points: An n-by-2 matrix containing the points that will be used to create the graph. Each row is a point.

beta: A positive real value used to parameterize the graph.

Returns:

Error code.

Time Complexity: Around O(n^floor(d/2) log n), where n is the number of points and d is the dimensionality of the point set.

igraph_beta_weighted_gabriel_graph — A Gabriel graph, with edges weighted by the β value at which it disappears.

```
igraph_error_t igraph_beta_weighted_gabriel_graph(
    igraph_t *graph,
    igraph_vector_t *weights,
    const igraph_matrix_t *points,
    igraph_real_t max_beta);
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

This function generates a Gabriel graph, and for each edge of this graph it computes the threshold β value at which the edge ceases to be part of the lune-based β-skeleton. For edges that continue to be part of β-skeletons for arbitrarily large β, IGRAPH_INFINIT# is returned.

The *max_beta* cutoff parameter controls the largest β value to consider For edges that persist above this β value, IGRAPH_INFINIT# is returned. This parameter serves to improve performance: the smaller this cutoff, the faster the computation. Pass IGRAPH_INFINITY to use no cutoff.

Arguments:

graph: A pointer to the graph that will be created.

weights: Will contain the edge weights corresponding to the edge indices from the graph.

points: A matrix containing the points that will be used. Each row is a point, dimensionality is inferred from the column count. There must be no duplicate points.

max_beta: Maximum value of beta to search to, higher values will be represented as `IGRAPH_INFINITY`.

Returns:

Error code.

See also:

`igraph_lune_beta_skeleton()` or `igraph_circle_beta_skeleton()` to generate a graph with a given value of beta; `igraph_gabriel_graph()` to only generate a Gabriel graph, without edge weights.

Time Complexity: Around O(n^floor(d/2) log n), where n is the number of points and d is the dimensionality of the point set. Though large values of max_beta can cause long run times if there are edges that disappear only at large betas.

Properties of spatial graphs

`igraph_spatial_edge_lengths` — Edge lengths based on spatial vertex coordinates.

```
igraph_error_t igraph_spatial_edge_lengths(
        const igraph_t *graph,
        igraph_vector_t *lengths,
        const igraph_matrix_t *points,
        igraph_metric_t metric);
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

The length of each edge is computed based on spatial coordinates. The length can be employed by several igraph functions, such as `igraph_voronoi()`, `igraph_betweenness()`, `igraph_closeness()` and others.

Arguments:

graph: The graph whose edge lengths are to be computed.

lengths: An initialized vector. Length will be stored here, in the order of edge IDs. It will be resized as needed.

points: A matrix of vertex coordinates. Each row contains the coordinates of the corresponding vertex, in the order of vertex IDs. Arbitrary dimensional point sets are supported.

metric: The distance metric to use. See `igraph_metric_t` for valid values.

Returns:

Error code.

See also:

`igraph_nearest_neighbor_graph()` computes a k nearest neighbor graph and `igraph_delaunay_graph()` computes a Delaunay graph based on a set of spatial points.

Time complexity: O(|E| d) where |E| is the number of edges and d is the dimensionality of the point set.

Non-graph related spatial processing

`igraph_convex_hull_2d` — Determines the convex hull of a given set of points in the 2D plane.

```
igraph_error_t igraph_convex_hull_2d(
        const igraph_matrix_t *data,
        igraph_vector_int_t *resverts,
        igraph_matrix_t *rescoords);
```

The convex hull is determined by the Graham scan algorithm. See the following reference for details:

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.

Arguments:

data: vector containing the coordinates. The length of the vector must be even, since it contains X-Y coordinate pairs.

resverts: the vector containing the result, e.g. the vector of vertex indices used as the corners of the convex hull. Supply `NULL` here if you are only interested in the coordinates of the convex hull corners.

rescoords: the matrix containing the coordinates of the selected corner vertices. Supply `NULL` here if you are only interested in the vertex indices.

Returns:

Error code: `IGRAPH_ENOMEM`: not enough memory

Time complexity: O(n log(n)) where n is the number of vertices.

Chapter 15. Graph operators

Union and intersection

`igraph_disjoint_union` — Creates the union of two disjoint graphs.

```
igraph_error_t igraph_disjoint_union(igraph_t *res,
                                     const igraph_t *left,
                                     const igraph_t *right);
```

First the vertices of the second graph will be relabeled with new vertex IDs to have two disjoint sets of vertex IDs, then the union of the two graphs will be formed. If the two graphs have |V1| and |V2| vertices and |E1| and |E2| edges respectively then the new graph will have |V1|+|V2| vertices and |E1|+|E2| edges.

The vertex and edge ordering of the graphs will be preserved. In other words, the vertex and edge IDs of the first graph map to identical values in the new graph, while the vertex and edge IDs of the second graph map to IDs incremented by the vertex and edge count of the first graph.

Both graphs need to have the same directedness, i.e. either both directed or both undirected.

The current version of this function cannot handle graph, vertex and edge attributes, they will be lost.

Arguments:

`res`: Pointer to an uninitialized graph object, the result will stored here.

`left`: The first graph.

`right`: The second graph.

Returns:

Error code.

See also:

`igraph_disjoint_union_many()` for creating the disjoint union of more than two graphs, `igraph_union()` for non-disjoint union.

Time complexity: O(|V1|+|V2|+|E1|+|E2|).

Example 15.1. File `examples/simple/igraph_disjoint_union.c`

`igraph_disjoint_union_many` — The disjoint union of many graphs.

```
igraph_error_t igraph_disjoint_union_many(igraph_t *res,
                                          const igraph_vector_ptr_t *graphs);
```

First the vertices in the graphs will be relabeled with new vertex IDs to have pairwise disjoint vertex ID sets and then the union of the graphs is formed. The number of vertices and edges in the result is the total number of vertices and edges in the graphs.

The vertex and edge ordering of the input graphs is preserved in the output graph.

All graphs need to have the same directedness, i.e. either all directed or all undirected. If the graph list has length zero, the result will be a *directed* graph with no vertices.

The current version of this function cannot handle graph, vertex and edge attributes, they will be lost.

Arguments:

res: Pointer to an uninitialized graph object, the result of the operation will be stored here.

graphs: Pointer vector, contains pointers to initialized graph objects.

Returns:

Error code.

See also:

`igraph_disjoint_union()` for an easier syntax if you have only two graphs, `igraph_union_many()` for non-disjoint union.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges in the result.

`igraph_join` — Creates the join of two disjoint graphs.

```
igraph_error_t igraph_join(igraph_t *res,
                           const igraph_t *left,
                           const igraph_t *right);
```

First the vertices of the second graph will be relabeled with new vertex IDs to have two disjoint sets of vertex IDs, then the union of the two graphs will be formed. Finally, the vertces from the first graph will have edges added to each vertex from the second. If the two graphs have |V1| and |V2| vertices and |E1| and |E2| edges respectively then the new graph will have |V1|+|V2| vertices and |E1|+|E2|+|V1|*|V2| edges.

The vertex ordering of the graphs will be preserved. In other words, the vertex IDs of the first graph map to identical values in the new graph, while the vertex IDs of the second graph map to IDs incremented by the vertex count of the first graph. The new edges will be grouped with the other edges that share a from vertex.

Both graphs need to have the same directedness, i.e. either both directed or both undirected. If both graphs are directed, then for each vertex v, u in graphs G1, G2 we add edges (v, u), (u, v) to maintain completeness.

The current version of this function cannot handle graph, vertex and edge attributes, they will be lost.

Arguments:

res: Pointer to an uninitialized graph object, the result will be stored here.

left: The first graph.

right: The second graph.

Returns:

Error code.

Time complexity: O(|V1|*|V2|+|E1|+|E2|).

igraph_union — Calculates the union of two graphs.

```
igraph_error_t igraph_union(
        igraph_t *res,
        const igraph_t *left, const igraph_t *right,
        igraph_vector_int_t *edge_map1, igraph_vector_int_t *edge_map2);
```

The number of vertices in the result is that of the larger graph from the two arguments. The result graph contains edges which are present in at least one of the operand graphs.

The directedness of the operand graphs must be the same.

Edge multiplicities are handled by taking the *larger* of the two multiplicities in the input graphs. In other words, if the first graph has N edges between a vertex pair (u, v) and the second graph has M edges, the result graph will have max(N, M) edges between them.

Arguments:

res: Pointer to an uninitialized graph object, the result will be stored here.

left: The first graph.

right: The second graph.

edge_map1: Pointer to an initialized vector or a null pointer. If not a null pointer, it will contain a mapping from the edges of the first argument graph (*left*) to the edges of the result graph.

edge_map2: The same as *edge_map1*, but for the second graph, *right*.

Returns:

Error code.

See also:

`igraph_union_many()` for the union of many graphs, `igraph_intersection()` and `igraph_difference()` for other operators.

Time complexity: O(|V|+|E|), |V| is the number of vertices, |E| the number of edges in the result graph.

Example 15.2. File `examples/simple/igraph_union.c`

igraph_union_many — Creates the union of many graphs.

```
igraph_error_t igraph_union_many(
    igraph_t *res, const igraph_vector_ptr_t *graphs,
```

```
    igraph_vector_int_list_t *edgemaps
);
```

The result graph will contain as many vertices as the largest graph among the arguments does, and an edge will be included in it if it is part of at least one operand graph.

The number of vertices in the result graph will be the maximum number of vertices in the argument graphs.

The directedness of the argument graphs must be the same. If the graph list has length zero, the result will be a *directed* graph with no vertices.

Edge multiplicities are handled by taking the *maximum* multiplicity of the all multiplicities for the same vertex pair (u, v) in the input graphs; this will be the multiplicity of (u, v) in the result graph.

Arguments:

res: Pointer to an uninitialized graph object, this will contain the result.

graphs: Pointer vector, contains pointers to the operands of the union operator, graph objects of course.

edgemaps: If not a null pointer, then it must be an initialized list of integer vectors, and the mappings of edges from the graphs to the result graph will be stored here, in the same order as *graphs*. Each mapping is stored in a separate igraph_vector_int_t object.

Returns:

Error code.

See also:

`igraph_union()` for the union of two graphs, `igraph_intersection_many()`, `igraph_intersection()` and `igraph_difference` for other operators.

Time complexity: O(|V|+|E|), |V| is the number of vertices in largest graph and |E| is the number of edges in the result graph.

`igraph_intersection` — Collect the common edges from two graphs.

```
igraph_error_t igraph_intersection(
        igraph_t *res,
        const igraph_t *left, const igraph_t *right,
        igraph_vector_int_t *edge_map1, igraph_vector_int_t *edge_map2);
```

The result graph contains only edges present both in the first and the second graph. The number of vertices in the result graph is the same as the larger from the two arguments.

The directedness of the operand graphs must be the same.

Edge multiplicities are handled by taking the *smaller* of the two multiplicities in the input graphs. In other words, if the first graph has N edges between a vertex pair (u, v) and the second graph has M edges, the result graph will have min(N, M) edges between them.

Arguments:

res: Pointer to an uninitialized graph object. This will contain the result of the operation.

left: The first operand, a graph object.

right: The second operand, a graph object.

edge_map1: Null pointer, or an initialized vector. If the latter, then a mapping from the edges of the result graph, to the edges of the *left* input graph is stored here. For the edges that are not in the intersection, -1 is stored.

edge_map2: Null pointer, or an initialized vector. The same as *edge_map1*, but for the *right* input graph. For the edges that are not in the intersection, -1 is stored.

Returns:

Error code.

See also:

`igraph_intersection_many()` to calculate the intersection of many graphs at once, `igraph_union()`, `igraph_difference()` for other operators.

Time complexity: O(|V|+|E|), |V| is the number of nodes, |E| is the number of edges in the smaller graph of the two. (The one containing less vertices is considered smaller.)

Example 15.3. File `examples/simple/igraph_intersection.c`

`igraph_intersection_many` — The intersection of more than two graphs.

```
igraph_error_t igraph_intersection_many(
    igraph_t *res, const igraph_vector_ptr_t *graphs,
    igraph_vector_int_list_t *edgemaps
);
```

This function calculates the intersection of the graphs stored in the *graphs* argument. Only those edges will be included in the result graph which are part of every graph in *graphs*.

The number of vertices in the result graph will be the maximum number of vertices in the argument graphs.

The directedness of the argument graphs must be the same. If the graph list has length zero, the result will be a *directed* graph with no vertices.

Edge multiplicities are handled by taking the *minimum* multiplicity of the all multiplicities for the same vertex pair (u, v) in the input graphs; this will be the multiplicity of (u, v) in the result graph.

Arguments:

res: Pointer to an uninitialized graph object, the result of the operation will be stored here.

graphs: Pointer vector, contains pointers to graphs objects, the operands of the intersection operator.

edgemaps: If not a null pointer, then it must be an initialized list of integer vectors, and the mappings of edges from the graphs to the result graph will be stored here, in the same order as *graphs*. Each mapping is stored in a separate igraph_vector_int_t object. For the edges that are not in the intersection, -1 is stored.

Returns:

Error code.

See also:

`igraph_intersection()` for the intersection of two graphs, `igraph_union_many()`, `igraph_union()` and `igraph_difference()` for other operators.

Time complexity: O(|V|+|E|), |V| is the number of vertices, |E| is the number of edges in the smallest graph (i.e. the graph having the less vertices).

Other set-like operators

`igraph_difference` — Calculates the difference of two graphs.

```
igraph_error_t igraph_difference(igraph_t *res,
                      const igraph_t *orig, const igraph_t *sub);
```

The number of vertices in the result is the number of vertices in the original graph, i.e. the left, first operand. In the results graph only edges will be included from *`orig`* which are not present in *`sub`*.

Arguments:

`res`: Pointer to an uninitialized graph object, the result will be stored here.

`orig`: The left operand of the operator, a graph object.

`sub`: The right operand of the operator, a graph object.

Returns:

Error code.

See also:

`igraph_intersection()` and `igraph_union()` for other operators.

Time complexity: O(|V|+|E|), |V| is the number vertices in the smaller graph, |E| is the number of edges in the result graph.

Example 15.4. File `examples/simple/igraph_difference.c`

`igraph_complementer` — Creates the complementer of a graph.

```
igraph_error_t igraph_complementer(igraph_t *res, const igraph_t *graph,
                        igraph_bool_t loops);
```

The complementer graph means that all edges which are not part of the original graph will be included in the result.

Arguments:

res: Pointer to an uninitialized graph object.

graph: The original graph.

loops: Whether to add loop edges to the complementer graph.

Returns:

Error code.

See also:

`igraph_union()`, `igraph_intersection()` and `igraph_difference()`.

Time complexity: O(|V|+|E1|+|E2|), |V| is the number of vertices in the graph, |E1| is the number of edges in the original and |E2| in the complementer graph.

Example 15.5. File `examples/simple/igraph_complementer.c`

`igraph_compose` — Calculates the composition of two graphs.

```
igraph_error_t igraph_compose(igraph_t *res,
                              const igraph_t *g1,
                              const igraph_t *g2,
                              igraph_vector_int_t *edge_map1,
                              igraph_vector_int_t *edge_map2);
```

The composition of graphs contains the same number of vertices as the bigger graph of the two operands. It contains an (i,j) edge if and only if there is a k vertex, such that the first graph contains an (i,k) edge and the second graph a (k,j) edge.

This is of course exactly the composition of two binary relations.

The two graphs must have the same directedness, otherwise the function returns with an error. Note that for undirected graphs the two relations are by definition symmetric.

Arguments:

res: Pointer to an uninitialized graph object, the result will be stored here.

g1: The first operand, a graph object.

g2: The second operand, another graph object.

edge_map1: If not a null pointer, then it must be a pointer to an initialized vector, and a mapping from the edges of the result graph to the edges of the first graph is stored here.

edge_map2: If not a null pointer, then it must be a pointer to an initialized vector, and a mapping from the edges of the result graph to the edges of the second graph is stored here.

Returns:

Error code.

Time complexity: O(|V|*d1*d2), |V| is the number of vertices in the first graph, d1 and d2 the average degree in the first and second graphs.

Example 15.6. File `examples/simple/igraph_compose.c`

Miscellaneous operators

`igraph_connect_neighborhood` — Connects each vertex to its neighborhood.

```
igraph_error_t igraph_connect_neighborhood(igraph_t *graph, igraph_int_t order,
                                igraph_neimode_t mode);
```

This function adds new edges to the input graph. Each vertex is connected to all vertices reachable by at most *`order`* steps from it (unless a connection already existed).

Note that the input graph is modified in place, no new graph is created. Call `igraph_copy()` if you want to keep the original graph as well.

For undirected graphs reachability is always symmetric: if vertex A can be reached from vertex B in at most *`order`* steps, then the opposite is also true. Only one undirected (A,B) edge will be added in this case.

Arguments:

`graph`: The input graph. It will be modified in-place.

`order`: Integer constant, it gives the distance within which the vertices will be connected to the source vertex.

`mode`: Constant, it specifies how the neighborhood search is performed for directed graphs. If `IGRAPH_OUT` then vertices reachable from the source vertex will be connected, `IGRAPH_IN` is the opposite. If `IGRAPH_ALL` then the directed graph is considered as an undirected one.

Returns:

Error code.

See also:

`igraph_graph_power()` to compute the kth power of a graph; `igraph_square_lattice()` uses this function to connect the neighborhood of the vertices.

Time complexity: O(|V|*d^k), |V| is the number of vertices in the graph, d is the average degree and k is the *`order`* argument.

`igraph_contract_vertices` — Replace multiple vertices with a single one.

```
igraph_error_t igraph_contract_vertices(igraph_t *graph,
                             const igraph_vector_int_t *mapping,
                             const igraph_attribute_combination_t *vertex_comb)
```

This function modifies the graph by merging several vertices into one. The vertices in the modified graph correspond to groups of vertices in the input graph. No edges are removed, thus the modified graph will typically have self-loops (corresponding to in-group edges) and multi-edges (corresponding to multiple connections between two groups). Use `igraph_simplify()` to eliminate self-loops and merge multi-edges.

Arguments:

graph: The input graph. It will be modified in-place.

mapping: A vector giving the mapping. For each vertex in the original graph, it should contain its desired ID in the result graph. In order not to create "orphan vertices" that have no corresponding vertices in the original graph, ensure that the IDs are consecutive integers starting from zero.

vertex_comb: What to do with the vertex attributes. `NULL` means that vertex attributes are not kept after the contraction (not even for unaffected vertices). See the igraph manual section about attributes for details.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number or vertices plus edges.

Example 15.7. File `examples/simple/igraph_contract_vertices.c`

`igraph_graph_power` — The k-th power of a graph.

```
igraph_error_t igraph_graph_power(const igraph_t *graph, igraph_t *res,
                                  igraph_int_t order, igraph_bool_t directed);
```

The k-th power of a graph G is a simple graph where vertex `u` is connected to `v` by a single edge if `v` is reachable from `u` in G within at most k steps. By convention, the zeroth power of a graph has no edges. The first power is identical to the original graph, except that multiple edges and self-loops are removed.

Graph power is usually defined only for undirected graphs. igraph extends the concept to directed graphs. To ignore edge directions in the input, set the *`directed`* parameter to `false`. In this case, the result will be an undirected graph.

Graph and vertex attributes are preserved, but edge attributes are discarded.

Arguments:

graph: The input graph.

res: The graph power of the given *`order`*.

order: Non-negative integer, the power to raise the graph to. In other words, vertices within a distance *`order`* will be connected.

directed: Whether to take edge directions into account.

Returns:

Error code.

See also:

`igraph_connect_neighborhood()` to connect each vertex to its neighborhood, modifying a graph in-place.

Time complexity: O(|V|*d^k), |V| is the number of vertices in the graph, d is the average degree and k is the *`order`* argument.

`igraph_product` — The graph product of two graphs, according to the chosen product type.

```
igraph_error_t igraph_product(igraph_t *res,
                              const igraph_t *g1,
                              const igraph_t *g2,
                              igraph_product_t type);
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

This function computes the product of two graphs using the graph product concept selected by the *`type`* parameter. The two graphs must be of the same type, either directed or undirected. If a product of an undirected and a directed graph is required, convert one of them to the appropriate type using `igraph_to_directed()` or `igraph_to_undirected()`.

Each vertex of the product graph corresponds to a pair `(u, v)`, where `u` is a vertex from the first graph and `v` is a vertex from the second graph. Thus the number of vertices in the product graph is `|V1| |V2|`, where `|V1|` and `|V2|` are the sizes of the vertex set of the operands. The pair `(u, v)` is mapped to a unique vertex index in the product graph using `index = u |V2| + v`.

All implemented graph products are associative, but not all are commutative.

The supported graph product types are detailed below. The notation `u ~ v` is used to indicate that vertices `u` and `v` are adjacent, i.e. there is a connection from `u` to `v`.

`IGRAPH_PRODUCT_CARTESIAN` Computes the cartesian product of two graphs. In the product graph, there is a connection from `(u1, v1)` to `(u2, v2)` if and only if `u1 = u2` and `v1 ~ v2` or `u1 ~ u2` and `v1 = v2`. Thus, the number of edges in the product graph is `|V1| |E2| + |V2| |E1|`.

Time complexity: O(|V1| |V2| + |V1| |E2| + |V2| |E1|) where |V1| and |V2| are the number of vertices, and |E1| and |E2| are the number of edges of the operands.

`IGRAPH_PRODUCT_LEXICOGRAPHIC` Computes the lexicographic product of two graphs. In the product graph, there is a connection from `(u1, v1)` to `(u2, v2)` if and only if `u1 = u2` and `v1 ~ v2` or `u1 ~ u2`. Thus, the number of edges in the product graph is `|V1| |E2| + |V2|^2 |E1|`. Unlike most other graph products, the lexicographic product is not commutative.

Time complexity: O(|V1| |V2| + |V1| |E2| + |V2|^2 |E1|) where |V1| and |V2| are the number of vertices, and |E1| and |E2| are the number of edges of the operands.

`IGRAPH_PRODUCT_STRONG` Computes the strong product (also known as normal product) of two graphs. In the product graph, there is a connection from `(u1, v1)` to `(u2, v2)` if and only if `u1 = u2` and `v1 ~ v2` or `u1 ~ u2` and `v1 = v2` or `u1 ~ u2` and `v1 ~ v2`. Thus, the number of edges in the product graph is `|V1| |E2| + |V2| |E1| + |E1| |E2|` in the directed case and `|V1| |E2| + |V2| |E1| + 2 |E1| |E2|` in the undirected case.

Time complexity: O(|V1| |V2| + |V1| |E2| + |V2| |E1| + |E1| |E2|) where |V1| and |V2| are the number of vertices, and |E1| and |E2| are the number of edges of the operands.

`IGRAPH_PRODUCT_TENSOR` Computes the tensor product (also known as categorial product) of two graphs. In the product graph, there is a connection from `(u1, v1)` to `(u2, v2)` if and only if `u1 ~ u2` and `v1 ~ v2`. Thus, the number of edges in the product is `|E1| |E2|` in the directed case and `2 |E1| |E2|` in the undirected case.

Time complexity: O(|V1| |V2| + |E1| |E2|) where |V1| and |V2| are the number of vertices, and |E1| and |E2| are the number of edges of the operands.

`IGRAPH_PRODUCT_MODULAR` Computes the modular product of two graphs. In the product graph, there is a connection from `(u1, v1)` to `(u2, v2)` if and only if `u1 ~ u2` and `v1 ~ v2` or `NOT (u1 ~ u2)` and `NOT (v1 ~ v2)`. The modular product requires both graphs to be simple. Thus, the number of edges in the product is `|E1| |E2| + |E1'| |E2'|` in the directed case and `2 |E1| |E2| + 2 |E1'| |E2'|` in the undirected case.

Time complexity: O(|V1| |V2| + |E1| |E2| + |E1'| |E2'|) where |V1| and |V2| are the number of vertices, |E1| and |E2| are the number of edges of the operands, and |E1'| and |E2'| are the number of edges of their complement.

Reference:

Richard Hammack, Wilfried Imrich, and Sandi Klavžar (2011). Handbook of Product Graphs (2nd ed.). CRC Press. https://doi.org/10.1201/b10959

Arguments:

res: Pointer to an uninitialized graph object. The product graph will be stored here.

g1: The first operand graph.

g2: The second operand graph. It must have the same directedness as *g1*.

type: The type of graph product to compute.

Returns:

Error code: `IGRAPH_EINVAL` if the specified *type* is unsupported or the input graphs *g1* and *g2* are incompatible for the requested product.

See also:

`igraph_rooted_product()` for the rooted product.

`igraph_rooted_product` — The rooted graph product of two graphs.

```
igraph_error_t igraph_rooted_product(igraph_t *res,
                                     const igraph_t *g1,
                                     const igraph_t *g2,
                                     const igraph_int_t root);
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

This function computes the rooted product of two graphs. The two graphs must be of the same type, either directed or undirected. If a product of an undirected and a directed graph is required, convert one of them to the appropriate type using `igraph_to_directed()` or `igraph_to_undirected()`.

The vertex IDs in the product graph related to the IDs in the operands in the same convention as in `igraph_product()`.

In the rooted product graph of G and H, with root vertex ID *`root`* in H, there is a connection from `(u1, v1)` to `(u2, v2)` if and only if `u1 = u2` and `v1 ~ v2` or `u1 ~ u2` and `v1 = v2 = root`. Thus, the number of edges in the product graph is `|V1| |E2| + |E1|`.

Arguments:

`res`: Pointer to an uninitialized graph object. The product graph will be stored here.

`g1`: The first operand graph.

`g2`: The second operand graph. It must have the same directedness as *`g1`*.

`root`: The root vertex id of the second graph.

Returns:

Error code: `IGRAPH_EINVAL` if the specified *`type`* is unsupported or the input graphs *`g1`* and *`g2`* are incompatible for the requested product. `IGRAPH_EINVVID` if invalid vertex ID passed as *`root`*.

See also:

`igraph_product()` for other types of graph products.

Time complexity: O(|V1| |V2| + |V1| |E2| + |E1|) where |V1| and |V2| are the number of vertices, and |E1| and |E2| are the number of edges of the operands.

`igraph_induced_subgraph` — Creates a subgraph induced by the specified vertices.

```
igraph_error_t igraph_induced_subgraph(const igraph_t *graph, igraph_t *res,
```

```
                                        const igraph_vs_t vids,
                                        igraph_subgraph_implementation_t impl);
```

This function collects the specified vertices and all edges between them to a new graph. As vertex IDs are always contiguos integers starting at zero, the IDs in the created subgraph will be different from the IDs in the original graph. To get the mappings between them, use `igraph_induced_subgraph_map()`

Arguments:

graph: The graph object.

res: The subgraph, another graph object will be stored here, do *not* initialize this object before calling this function, and call `igraph_destroy()` on it if you don't need it any more.

vids: A vertex selector describing which vertices to keep. A vertex may appear more than once in the selector, but it will be considered only once (i.e. it is not possible to duplicate a vertex by adding its ID more than once to the selector). The order in which the vertices appear in the vertex selector is ignored; the returned subgraph will always contain the vertices of the original graph in increasing order of vertex IDs.

impl: This parameter selects which implementation should we use when constructing the new graph. Basically there are two possibilities: `IGRAPH_SUBGRAPH_COPY_AND_DELETE` copies the existing graph and deletes the vertices that are not needed in the new graph, while `IGRAPH_SUBGRAPH_CREATE_FROM_SCRATCH` constructs the new graph from scratch without copying the old one. The latter is more efficient if you are extracting a relatively small subpart of a very large graph, while the former is better if you want to extract a subgraph whose size is comparable to the size of the whole graph. There is a third possibility: `IGRAPH_SUBGRAPH_AUTO` will select one of the two methods automatically based on the ratio of the number of vertices in the new and the old graph.

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory for temporary data. `IGRAPH_EINVVID`, invalid vertex ID in *vids*.

Time complexity: O(|V|+|E|), |V| and |E| are the number of vertices and edges in the original graph.

See also:

`igraph_induced_subgraph_map()` to also retrieve the vertex ID mapping between the graph and the extracted subgraph; `igraph_delete_vertices()` to delete the specified set of vertices from a graph, the opposite of this function.

`igraph_induced_subgraph_map` — Creates an induced subraph and returns the mapping from the original.

```
igraph_error_t igraph_induced_subgraph_map(const igraph_t *graph, igraph_t *res
                                           const igraph_vs_t vids,
                                           igraph_subgraph_implementation_t imp
                                           igraph_vector_int_t *map,
                                           igraph_vector_int_t *invmap);
```

This function collects the specified vertices and all edges between them to a new graph. As vertex IDs are always contiguos integers starting at zero, the IDs in the created subgraph will be different

from the IDs in the original graph. The mapping between the vertex IDs in the graph and the extracted subgraphs are returned in *map* and *invmap*.

Arguments:

graph: The graph object.

res: The subgraph, another graph object will be stored here, do *not* initialize this object before calling this function, and call `igraph_destroy()` on it if you don't need it any more.

vids: A vertex selector describing which vertices to keep.

impl: This parameter selects which implementation should be used when constructing the new graph. Basically there are two possibilities: `IGRAPH_SUBGRAPH_COPY_AND_DELETE` copies the existing graph and deletes the vertices that are not needed in the new graph, while `IGRAPH_SUBGRAPH_CREATE_FROM_SCRATCH` constructs the new graph from scratch without copying the old one. The latter is more efficient if you are extracting a relatively small subpart of a very large graph, while the former is better if you want to extract a subgraph whose size is comparable to the size of the whole graph. There is a third possibility: `IGRAPH_SUBGRAPH_AUTO` will select one of the two methods automatically based on the ratio of the number of vertices in the new and the old graph.

map: Returns a map of the vertices in *graph* to the vertices in *res*. -1 indicates a vertex is not mapped. A value of `i` at position `j` indicates the vertex `j` in *graph* is mapped to vertex i in *res*.

invmap: Returns a map of the vertices in *res* to the vertices in *graph*. A value of `i` at position `j` indicates that vertex `i` in *graph* is mapped to vertex `j` in *res*.

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory for temporary data. `IGRAPH_EINVVID`, invalid vertex ID in *vids*.

Time complexity: O(|V|+|E|), |V| and |E| are the number of vertices and edges in the original graph.

See also:

`igraph_delete_vertices()` to delete the specified set of vertices from a graph, the opposite of this function.

`igraph_induced_subgraph_edges` — The edges contained within an induced sugraph.

```
igraph_error_t igraph_induced_subgraph_edges(const igraph_t *graph, igraph_vs_t
```

This function finds the IDs of those edges which connect vertices from a given list, passed in the *vids* parameter.

Arguments:

graph: The graph.

vids: A vertex selector specifying the vertices that make up the subgraph.

edges: Integer vector. The IDs of edges within the subgraph induces by *vids* will be stored here.

Returns:

Error code.

Time complexity: O(mv log(nv)) where nv is the number of vertices in *vids* and mv is the sum of degrees of vertices in *vids*.

igraph_linegraph — Create the line graph of a graph.

```
igraph_error_t igraph_linegraph(const igraph_t *graph, igraph_t *linegraph);
```

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 different vertices in L(G) are connected by an edge if their corresponding edges share an end point. In a multigraph, if two end points are shared, two edges are created. The single vertex of an undirected self-loop is counted as two end points.

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.

Self-loops are considered self-adjacent, thus their corresponding vertex in the line graph will also a have a single self-loop, in both undirected and directed graphs.

Edge *i* in the original graph will correspond to vertex *i* in the line graph.

The first version of this function was contributed by Vincent Matossian, thanks.

Arguments:

graph: The input graph, may be directed or undirected.

linegraph: Pointer to an uninitialized graph object, the result is stored here.

Returns:

Error code.

Time complexity: O(|V|+|E|), the number of edges plus the number of vertices.

igraph_mycielskian — Generate the Mycielskian of a graph with *k* iterations.

```
igraph_error_t igraph_mycielskian(const igraph_t *graph, igraph_t *res, igraph_
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

The Mycielskian of a graph is a larger graph formed using a construction due to Jan Mycielski that increases the chromatic number by one while preserving the triangle-free property. The Mycielski construction can be used to create triangle-free graphs with an arbitrarily large chromatic number.

Let the n vertices of the given graph G be v_1, ..., v_n. The Mycielskian of G, denoted M(G), contains G itself as a subgraph, together with n+1 additional vertices:

- A vertex u_i corresponding to each vertex v_i of G.
- An extra vertex w.

The edges are added as follows:

- Each vertex u_i is connected to w, forming a star.
- For each edge (v_i, v_j) in G, two new edges are added: (u_i, v_j) and (v_i, u_j).

Thus, if G has n vertices and m edges, the Mycielskian M(G) has 2n + 1 vertices, and 3m + n edges.

igraph uses an alternative construction in two special cases:

- The Mycielskian of the null graph is the singleton graph.
- The Mycielskian of the singleton graph is the two-path.

This ensures that iterative applications of the construction, starting from the null or singleton graph, always yields connected graphs. In fact these are the Mycielski graphs that igraph_mycielski_graph() produces.

igraph extends the construction to directed graphs, as well as to non-simple graphs, by following the above constructions rules literally.

This function can apply the Mycielski transformation an arbitrary number of times, controlled by the parameter *k*. The k-th iterated Mycielskian has n_k = (n + 1) * 2^k - 1 vertices and m_k = ((2m + 2n + 1) * 3^k - n_{k+1}) / 2 edges, where n and m are the vertex and edge count of the original graph, respectively.

Arguments:

graph: Pointer to the input graph.

res: Pointer to an uninitialized graph object where the Mycielskian of the input graph will be stored.

k: Integer, the number of Mycielskian iterations (must be non-negative).

Returns:

Error code.

See also:

igraph_mycielski_graph().

Time complexity: O(|V| 2^k + |E| 3^k) where |V| and |E| are the vertex and edge counts, respectively.

igraph_simplify — Removes loop and/or multiple edges from the graph.

```
igraph_error_t igraph_simplify(igraph_t *graph,
                               igraph_bool_t remove_multiple, igraph_bool_t rem
                               const igraph_attribute_combination_t *edge_comb)
```

This function merges parallel edges and removes self-loops, according to the *multiple* and *loops* parameters. Note that this function may change the edge order, even if the input was already a simple graph.

Arguments:

graph: The graph object.

remove_multiple: If true, multiple edges will be removed.

remove_loops: If true, loops (self edges) will be removed.

edge_comb: What to do with the edge attributes. NULL means to discard the edge attributes after the operation, even for edges that were unaffected. See the igraph manual section about attributes for details.

Returns:

Error code: IGRAPH_ENOMEM if we are out of memory.

Time complexity: O(|V|+|E|).

Example 15.8. File examples/simple/igraph_simplify.c

igraph_subgraph_from_edges — Creates a subgraph with the specified edges and their endpoints.

```
igraph_error_t igraph_subgraph_from_edges(
    const igraph_t *graph, igraph_t *res, const igraph_es_t eids,
    igraph_bool_t delete_vertices
);
```

This function collects the specified edges and their endpoints to a new graph. As the edge IDs in a graph are always contiguous integers starting at zero, the edge IDs in the extracted subgraph will be different from those in the original graph. Vertex IDs will also be reassigned if *delete_vertices* is set to true. Attributes are preserved.

Arguments:

graph: The graph object.

res: The subgraph, another graph object will be stored here, do *not* initialize this object before calling this function, and call igraph_destroy() on it if you don't need it any more.

eids: An edge selector describing which edges to keep.

delete_vertices: Whether to delete the vertices not incident on any of the specified edges as well. If false, the number of vertices in the result graph will always be equal to the number of vertices in the input graph.

Returns:

Error code: IGRAPH_ENOMEM, not enough memory for temporary data. IGRAPH_EINVEID, invalid edge ID in *eids*.

Time complexity: O(|V|+|E|), |V| and |E| are the number of vertices and edges in the original graph.

See also:

`igraph_delete_edges()` to delete the specified set of edges from a graph, the opposite of this function.

`igraph_reverse_edges` — Reverses some edges of a directed graph.

```
igraph_error_t igraph_reverse_edges(igraph_t *graph, const igraph_es_t eids);
```

This function reverses some edges of a directed graph. The modification is done in place. All attributes, as well as the ordering of edges and vertices are preserved.

Note that is rarely necessary to reverse *all* edges, as almost all functions that handle directed graphs take a `mode` argument that can be set to `IGRAPH_IN` to effectively treat edges as reversed.

Arguments:

`graph`: The graph whose edges will be reversed.

`eids`: The edges to be reversed. Pass `igraph_ess_all(IGRAPH_EDGEORDER_ID)` to reverse all edges.

Returns:

Error code.

Time complexity: O(1) if all edges are reversed, otherwise O(|E|) where |E| is the number of edges in the graph.

Chapter 16. Graph visitors

Breadth-first search

`igraph_bfs` — Breadth-first search.

```
igraph_error_t igraph_bfs(const igraph_t *graph,
               igraph_int_t root, const igraph_vector_int_t *roots,
               igraph_neimode_t mode, igraph_bool_t unreachable,
               const igraph_vector_int_t *restricted,
               igraph_vector_int_t *order, igraph_vector_int_t *rank,
               igraph_vector_int_t *parents,
               igraph_vector_int_t *pred, igraph_vector_int_t *succ,
               igraph_vector_int_t *dist, igraph_bfshandler_t *callback,
               void *extra);
```

A simple breadth-first search, with a lot of different results and the possibility to call a callback whenever a vertex is visited. It is allowed to supply null pointers as the output arguments the user is not interested in, in this case they will be ignored.

If not all vertices can be reached from the supplied root vertex, then additional root vertices will be used, in the order of their vertex IDs.

Consider using `igraph_bfs_simple` instead if you set most of the output arguments provided by this function to a null pointer.

Arguments:

graph: The input graph.

root: The id of the root vertex. It is ignored if the `roots` argument is not a null pointer.

roots: Pointer to an initialized vector, or a null pointer. If not a null pointer, then it is a vector containing root vertices to start the BFS from. The vertices are considered in the order they appear. If a root vertex was already found while searching from another one, then no search is conducted from it.

mode: For directed graphs, it defines which edges to follow. `IGRAPH_OUT` means following the direction of the edges, `IGRAPH_IN` means the opposite, and `IGRAPH_ALL` ignores the direction of the edges. This parameter is ignored for undirected graphs.

unreachable: Boolean, whether the search should visit the vertices that are unreachable from the given root node(s). If true, then additional searches are performed until all vertices are visited.

restricted: If not a null pointer, then it must be a pointer to a vector containing vertex IDs. The BFS is carried out only on these vertices.

order: If not null pointer, then the vertex IDs of the graph are stored here, in the same order as they were visited.

rank: If not a null pointer, then the rank of each vertex is stored here.

parents: If not a null pointer, then the id of the parent of each vertex is stored here. When a vertex was not visited during the traversal, -2 will be stored as the ID of its parent.

When a vertex was visited during the traversal and it was one of the roots of the search trees, -1 will be stored as the ID of its parent.

pred: If not a null pointer, then the id of vertex that was visited before the current one is stored here. If there is no such vertex (the current vertex is the root of a search tree), then -1 is stored as the predecessor of the vertex. If the vertex was not visited at all, then -2 is stored for the predecessor of the vertex.

succ: If not a null pointer, then the id of the vertex that was visited after the current one is stored here. If there is no such vertex (the current one is the last in a search tree), then -1 is stored as the successor of the vertex. If the vertex was not visited at all, then -2 is stored for the successor of the vertex.

dist: If not a null pointer, then the distance from the root of the current search tree is stored here for each vertex. If a vertex was not reached during the traversal, its distance will be -1 in this vector.

callback: If not null, then it should be a pointer to a function of type `igraph_bfshandler_t`. This function will be called, whenever a new vertex is visited.

extra: Extra argument to pass to the callback function.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

Example 16.1. File `examples/simple/igraph_bfs.c`

Example 16.2. File `examples/simple/igraph_bfs_callback.c`

`igraph_bfs_simple` — Breadth-first search, single-source version

```
igraph_error_t igraph_bfs_simple(
    const igraph_t *graph, igraph_int_t root, igraph_neimode_t mode,
    igraph_vector_int_t *order, igraph_vector_int_t *layers,
    igraph_vector_int_t *parents
);
```

An alternative breadth-first search implementation to cater for the simpler use-cases when only a single breadth-first search has to be conducted from a source node and most of the output arguments from `igraph_bfs` are not needed. It is allowed to supply null pointers as the output arguments the user is not interested in, in this case they will be ignored.

Arguments:

graph: The input graph.

root: The id of the root vertex.

mode: For directed graphs, it defines which edges to follow. `IGRAPH_OUT` means following the direction of the edges, `IGRAPH_IN` means the opposite, and `IGRAPH_ALL` ignores the direction of the edges. This parameter is ignored for undirected graphs.

order: If not a null pointer, then an initialized vector must be passed here. The IDs of the vertices visited during the traversal will be stored here, in the same order as they were visited.

layers: If not a null pointer, then an initialized vector must be passed here. The i-th element of the vector will contain the index into `order` where the vertices that are at distance i from the root are stored. In other words, if you are interested in the vertices that are at distance i from the root, you need to look in the `order` vector from `layers`[i] to `layers`[i+1].

parents: If not a null pointer, then an initialized vector must be passed here. The vector will be resized so its length is equal to the number of nodes, and it will contain the index of the parent node for each *visited* node. The values in the vector are set to -2 for vertices that were *not* visited, and -1 for the root vertex.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

Example 16.3. File `examples/simple/igraph_bfs_simple.c`

`igraph_bfshandler_t` — Callback type for BFS function.

```
typedef igraph_error_t igraph_bfshandler_t(const igraph_t *graph,
        igraph_int_t vid,
        igraph_int_t pred,
        igraph_int_t succ,
        igraph_int_t rank,
        igraph_int_t dist,
        void *extra);
```

`igraph_bfs()` is able to call a callback function, whenever a new vertex is found, while doing the breadth-first search. This callback function must be of type `igraph_bfshandler_t`. It has the following arguments:

Arguments:

graph: The graph that the algorithm is working on. Of course this must not be modified.

vid: The id of the vertex just found by the breadth-first search.

pred: The id of the previous vertex visited. It is -1 if there is no previous vertex, because the current vertex is the root is a search tree.

succ: The id of the next vertex that will be visited. It is -1 if there is no next vertex, because the current vertex is the last one in a search tree.

rank: The rank of the current vertex, it starts with zero.

dist: The distance (number of hops) of the current vertex from the root of the current search tree.

extra: The extra argument that was passed to `igraph_bfs()`.

Returns:

`IGRAPH_SUCCESS` if the BFS should continue, `IGRAPH_STOP` if the BFS should stop and return to the caller normally. Any other value is treated as an igraph error code, terminating the search and returning to the caller with the same error code. If a BFS is is terminated prematurely, then all elements of the result vectors that were not yet calculated at the point of the termination contain negative values.

See also:

`igraph_bfs()`

Depth-first search

`igraph_dfs` — Depth-first search.

```
igraph_error_t igraph_dfs(const igraph_t *graph, igraph_int_t root,
               igraph_neimode_t mode, igraph_bool_t unreachable,
               igraph_vector_int_t *order,
               igraph_vector_int_t *order_out, igraph_vector_int_t *parents,
               igraph_vector_int_t *dist, igraph_dfshandler_t *in_callback,
               igraph_dfshandler_t *out_callback,
               void *extra);
```

A simple depth-first search, with the possibility to call a callback whenever a vertex is discovered and/or whenever a subtree is finished. It is allowed to supply null pointers as the output arguments the user is not interested in, in this case they will be ignored.

If not all vertices can be reached from the supplied root vertex, then additional root vertices will be used, in the order of their vertex IDs.

Arguments:

`graph`: The input graph.

`root`: The id of the root vertex.

`mode`: For directed graphs, it defines which edges to follow. `IGRAPH_OUT` means following the direction of the edges, `IGRAPH_IN` means the opposite, and `IGRAPH_ALL` ignores the direction of the edges. This parameter is ignored for undirected graphs.

`unreachable`: Boolean, whether the search should visit the vertices that are unreachable from the given root node(s). If true, then additional searches are performed until all vertices are visited.

`order`: If not null pointer, then the vertex IDs of the graph are stored here, in the same order as they were discovered. The tail of the vector will be padded with -1 to ensure that the length of the vector is the same as the number of vertices, even if some vertices were not visited during the traversal.

`order_out`: If not a null pointer, then the vertex IDs of the graphs are stored here, in the order of the completion of their subtree. The tail of the vector will be padded with -1 to ensure that the length of the vector is the same as the number of vertices, even if some vertices were not visited during the traversal.

parents: If not a null pointer, then the id of the parent of each vertex is stored here. -1 will be stored for the root of the search tree; -2 will be stored for vertices that were not visited.

dist: If not a null pointer, then the distance from the root of the current search tree is stored here. -1 will be stored for vertices that were not visited.

in_callback: If not null, then it should be a pointer to a function of type `igraph_dfshandler_t`. This function will be called, whenever a new vertex is discovered.

out_callback: If not null, then it should be a pointer to a function of type `igraph_dfshandler_t`. This function will be called, whenever the subtree of a vertex is completed.

extra: Extra argument to pass to the callback function(s).

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

`igraph_dfshandler_t` — Callback type for the DFS function.

```
typedef igraph_error_t igraph_dfshandler_t(const igraph_t *graph,
        igraph_int_t vid,
        igraph_int_t dist,
        void *extra);
```

`igraph_dfs()` is able to call a callback function, whenever a new vertex is discovered, and/or whenever a subtree is completed. These callbacks must be of type `igraph_dfshandler_t`. They have the following arguments:

Arguments:

graph: The graph that the algorithm is working on. Of course this must not be modified.

vid: The id of the vertex just found by the depth-first search.

dist: The distance (number of hops) of the current vertex from the root of the current search tree.

extra: The extra argument that was passed to `igraph_dfs()`.

Returns:

`IGRAPH_SUCCESS` if the DFS should continue, `IGRAPH_STOP` if the DFS should stop and return to the caller normally. Any other value is treated as an igraph error code, terminating the search and returning to the caller with the same error code. If a DFS is is terminated prematurely, then all elements of the result vectors that were not yet calculated at the point of the termination contain negative values.

See also:

`igraph_dfs()`

Random walks

`igraph_random_walk` — Performs a random walk on a graph.

```
igraph_error_t igraph_random_walk(const igraph_t *graph,
                      const igraph_vector_t *weights,
                      igraph_vector_int_t *vertices,
                      igraph_vector_int_t *edges,
                      igraph_int_t start,
                      igraph_neimode_t mode,
                      igraph_int_t steps,
                      igraph_random_walk_stuck_t stuck);
```

Performs a random walk with a given length on a graph, from the given start vertex. Edge directions are (potentially) considered, depending on the *mode* argument.

Arguments:

graph: The input graph, it can be directed or undirected. Multiple edges are respected, so are loop edges.

weights: A vector of non-negative edge weights. It is assumed that at least one strictly positive weight is found among the outgoing edges of each vertex. Additionally, no edge weight may be NaN. If either case does not hold, an error is returned. If it is `NULL`, all edges are considered to have equal weight.

vertices: An allocated vector, the result is stored here as a list of vertex IDs. It will be resized as needed. It includes the vertex IDs of starting and ending vertices. Length of the vertices vector: *steps* + 1

edges: An initialized vector, the indices of traversed edges are stored here. It will be resized as needed. Length of the edges vector: *steps*

start: The start vertex for the walk.

mode: How to walk along the edges in directed graphs. `IGRAPH_OUT` means following edge directions, `IGRAPH_IN` means going opposite the edge directions, `IGRAPH_ALL` means ignoring edge directions. This argument is ignored for undirected graphs.

steps: The number of steps to take. If the random walk gets stuck, then the *stuck* argument specifies what happens. *steps* is the number of edges to traverse during the walk.

stuck: What to do if the random walk gets stuck. `IGRAPH_RANDOM_WALK_STUCK_RETURN` means that the function returns with a shorter walk; `IGRAPH_RANDOM_WALK_STUCK_ERROR` means that an `IGRAPH_ERWSTUCK` error is reported. In both cases, *vertices* and *edges* are truncated to contain the actual interrupted walk.

Returns:

Error code: `IGRAPH_ERWSTUCK` if the walk got stuck.

Time complexity: O(l + d) for unweighted graphs and O(l * log(k) + d) for weighted graphs, where `l` is the length of the walk, `d` is the total degree of the visited nodes and `k` is the average degree of vertices of the given graph.

Chapter 17. Structural properties of graphs

These functions usually calculate some structural property of a graph, like its diameter, the degree of the nodes, etc.

Basic properties

`igraph_are_adjacent` — Decides whether two vertices are adjacent.

```
igraph_error_t igraph_are_adjacent(const igraph_t *graph,
                        igraph_int_t v1, igraph_int_t v2,
                        igraph_bool_t *res);
```

Decides whether there are any edges that have *v1* and *v2* as endpoints. This function is of course symmetric for undirected graphs.

Arguments:

graph: The graph object.

v1: The first vertex.

v2: The second vertex.

res: Boolean, `true` if there is an edge from *v1* to *v2*, `false` otherwise.

Returns:

The error code `IGRAPH_EINVVID` is returned if an invalid vertex ID is given.

Time complexity: O(min(log(d1), log(d2))), d1 is the (out-)degree of *v1* and d2 is the (in-)degree of *v2*.

Sparsifiers

`igraph_spanner` — Calculates a spanner of a graph with a given stretch factor.

```
igraph_error_t igraph_spanner(const igraph_t *graph, igraph_vector_int_t *spanne
        igraph_real_t stretch, const igraph_vector_t *weights);
```

A spanner of a graph `G = (V,E)` with a stretch `t` is a subgraph `H = (V,Es)` such that `Es` is a subset of `E` and the distance between any pair of nodes in `H` is at most `t` times the distance in `G`. The returned graph is always a spanner of the given graph with the specified stretch. For weighted graphs the number of edges in the spanner is `O(k n^(1 + 1 / k))`, where `k` is `k = (t + 1) / 2`, `m` is the number of edges and `n` is the number of nodes in `G`. For unweighted graphs the number of edges is `O(n^(1 + 1 / k) + kn)`.

This function is based on the algorithm of Baswana and Sen: "A Simple and Linear Time Randomized Algorithm for Computing Sparse Spanners in Weighted Graphs". https://doi.org/10.1002/rsa.20130

Arguments:

graph: An undirected connected graph object. If the graph is directed, the directions of the edges will be ignored.

spanner: An initialized vector, the IDs of the edges that constitute the calculated spanner will be returned here. Use `igraph_subgraph_from_edges()` to extract the spanner as a separate graph object.

stretch: The stretch factor `t` of the spanner.

weights: The edge weights or `NULL`.

Returns:

Error code.

Time complexity: The algorithm is a randomized Las Vegas algorithm. The expected running time is O(km) where k is the value mentioned above and m is the number of edges.

(Shortest)-path related functions

`igraph_distances` — Length of the shortest paths between vertices.

```
igraph_error_t igraph_distances(
        const igraph_t *graph,
        const igraph_vector_t *weights,
        igraph_matrix_t *res,
        const igraph_vs_t from, const igraph_vs_t to,
        igraph_neimode_t mode);
```

Arguments:

graph: The graph object.

weights: Optional edge weights. If `NULL`, the graph is considered unweighted, i.e. all edge weights are 1.

res: The result of the calculation, a matrix. A pointer to an initialized matrix, to be more precise. The matrix will be resized if needed. It will have the same number of rows as the length of the *from* argument, and its number of columns is the number of vertices in the *to* argument. One row of the matrix shows the distances from/to a given vertex to the ones in *to*. For the unreachable vertices `IGRAPH_INFINITY` is returned.

from: The source vertices.

to: The target vertices. It is not allowed to include a vertex twice or more.

mode: The type of shortest paths to be used for the calculation in directed graphs. Possible values:

`IGRAPH_OUT` the lengths of the outgoing paths are calculated.

`IGRAPH_IN` the lengths of the incoming paths are calculated.

`IGRAPH_ALL` the directed graph is considered as an undirected one for the computation.

Returns:

Error code:

`IGRAPH_ENOMEM` not enough memory for temporary data.

`IGRAPH_EINVVID` invalid vertex ID passed.

`IGRAPH_EINVMODE` invalid mode argument.

Time complexity: O(n(|V|+|E|)), n is the number of vertices to calculate, |V| and |E| are the number of vertices and edges in the graph.

See also:

`igraph_get_shortest_paths()` to get the paths themselves, `igraph_distances_dijkstra()` for the weighted version with non-negative weights, `igraph_distances_bellman_ford()` if you also have negative weights.

Example 17.1. File `examples/simple/distances.c`

`igraph_distances_cutoff` — Length of the shortest paths between vertices, with cutoff.

```
igraph_error_t igraph_distances_cutoff(
        const igraph_t *graph,
        const igraph_vector_t *weights,
        igraph_matrix_t *res,
        const igraph_vs_t from, const igraph_vs_t to,
        igraph_neimode_t mode,
        igraph_real_t cutoff);
```

This function is similar to `igraph_distances()`, but paths longer than *`cutoff`* will not be considered.

Arguments:

`graph`: The graph object.

`weights`: Optional edge weights. If `NULL`, the graph is considered unweighted, i.e. all edge weights are equal to 1.

`res`: The result of the calculation, a matrix. A pointer to an initialized matrix, to be more precise. The matrix will be resized if needed. It will have the same number of rows as the length of the *`from`* argument, and its number of columns is the number of vertices in the *`to`* argument. One row of the matrix shows the distances from/to a given vertex to the ones in *`to`*. For the unreachable vertices `IGRAPH_INFINITY` is returned.

`from`: The source vertices._d

`to`: The target vertices. It is not allowed to include a vertex twice or more.

mode: The type of shortest paths to be used for the calculation in directed graphs. Possible values:

- `IGRAPH_OUT` the lengths of the outgoing paths are calculated.
- `IGRAPH_IN` the lengths of the incoming paths are calculated.
- `IGRAPH_ALL` the directed graph is considered as an undirected one for the computation.

cutoff: The maximal length of paths that will be considered. When the distance of two vertices is greater than this value, it will be returned as `IGRAPH_INFINITY`. Negative cutoffs and `IGRAPH_UNLIMITED` are treated as infinity.

Returns:

Error code:

- `IGRAPH_ENOMEM` not enough memory for temporary data.
- `IGRAPH_EINVVID` invalid vertex ID passed.
- `IGRAPH_EINVMODE` invalid mode argument.

Time complexity: O(s |E| + |V|), where s is the number of source vertices to use, and |V| and |E| are the number of vertices and edges in the graph.

See also:

`igraph_distances_dijkstra_cutoff()` for the weighted version with non-negative weights.

Example 17.2. File `examples/simple/distances.c`

`igraph_distances_dijkstra` — Weighted shortest path lengths between vertices.

```
igraph_error_t igraph_distances_dijkstra(
        const igraph_t *graph,
        igraph_matrix_t *res,
        const igraph_vs_t from,
        const igraph_vs_t to,
        const igraph_vector_t *weights,
        igraph_neimode_t mode);
```

This function implements Dijkstra's algorithm, which can find the weighted shortest path lengths from a source vertex to all other vertices. This function allows specifying a set of source and target vertices. The algorithm is run independently for each source and the results are retained only for the specified targets. This implementation uses a binary heap for efficiency.

Arguments:

graph: The input graph, can be directed.

res: The result, a matrix. A pointer to an initialized matrix should be passed here. The matrix will be resized as needed. Each row contains the distances from a single source, to the

vertices given in the *to* argument. Unreachable vertices have distance IGRAPH_INFINITY.

from: The source vertices.

to: The target vertices. It is not allowed to include a vertex twice or more.

weights: The edge weights. All edge weights must be non-negative for Dijkstra's algorithm to work. Additionally, no edge weight may be NaN. If either case does not hold, an error is returned. If this is a null pointer, then the unweighted version, igraph_distances() is called.

mode: For directed graphs; whether to follow paths along edge directions (IGRAPH_OUT), or the opposite (IGRAPH_IN), or ignore edge directions completely (IGRAPH_ALL). It is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(s*|E|log|V|+|V|), where |V| is the number of vertices, |E| the number of edges and s the number of sources.

See also:

igraph_distances() for a non-algorithm-specific interface or igraph_distances_bellman_ford() for a weighted variant that works in the presence of negative edge weights (but no negative loops)

Example 17.3. File examples/simple/distances.c

igraph_distances_dijkstra_cutoff — Weighted shortest path lengths between vertices, with cutoff.

```
igraph_error_t igraph_distances_dijkstra_cutoff(
        const igraph_t *graph,
        igraph_matrix_t *res,
        const igraph_vs_t from,
        const igraph_vs_t to,
        const igraph_vector_t *weights,
        igraph_neimode_t mode,
        igraph_real_t cutoff);
```

This function is similar to igraph_distances_dijkstra(), but paths longer than *cutoff* will not be considered.

Arguments:

graph: The input graph, can be directed.

res: The result, a matrix. A pointer to an initialized matrix should be passed here. The matrix will be resized as needed. Each row contains the distances from a single source, to the vertices given in the *to* argument. Vertices that are not reachable within distance *cutoff* will be assigned distance IGRAPH_INFINITY.

from: The source vertices.

to: The target vertices. It is not allowed to include a vertex twice or more.

weights: The edge weights. All edge weights must be non-negative for Dijkstra's algorithm to work. Additionally, no edge weight may be NaN. If either case does not hold, an error is returned. If this is a null pointer, then the unweighted version, `igraph_distances()` is called. Edges with positive infinite weights are ignored.

mode: For directed graphs; whether to follow paths along edge directions (`IGRAPH_OUT`), or the opposite (`IGRAPH_IN`), or ignore edge directions completely (`IGRAPH_ALL`). It is ignored for undirected graphs.

cutoff: The maximal length of paths that will be considered. When the distance of two vertices is greater than this value, it will be returned as `IGRAPH_INFINITY`. Negative cutoffs and `IGRAPH_UNLIMITED` are treated as infinity.

Returns:

Error code.

Time complexity: at most O(s |E| log|V| + |V|), where |V| is the number of vertices, |E| the number of edges and s the number of sources. The *cutoff* parameter will limit the number of edges traversed from each source vertex, which reduces the computation time.

See also:

`igraph_distances_cutoff()` for a (slightly) faster unweighted version.

Example 17.4. File `examples/simple/distances.c`

`igraph_distances_bellman_ford` — Weighted shortest path lengths between vertices, allowing negative weights.

```
igraph_error_t igraph_distances_bellman_ford(
        const igraph_t *graph,
        igraph_matrix_t *res,
        const igraph_vs_t from,
        const igraph_vs_t to,
        const igraph_vector_t *weights,
        igraph_neimode_t mode);
```

This function implements the Bellman-Ford algorithm to find the weighted shortest paths to all vertices from a single source, allowing negative weights. It is run independently for the given sources. If there are no negative weights, you are better off with `igraph_distances_dijkstra()`.

Arguments:

graph: The input graph, can be directed.

res: The result, a matrix. A pointer to an initialized matrix should be passed here, the matrix will be resized if needed. Each row contains the distances from a single source, to all vertices in the graph, in the order of vertex IDs. For unreachable vertices the matrix contains `IGRAPH_INFINITY`.

from: The source vertices.

to: The target vertices.

weights: The edge weights. There must not be any cycle in the graph that has a negative total weight (since this would allow us to decrease the weight of any path containing at least a single vertex of this cycle infinitely). Additionally, no edge weight may be NaN. If either case does not hold, an error is returned. If this is a null pointer, then the unweighted version, `igraph_distances()` is called.

mode: For directed graphs; whether to follow paths along edge directions (`IGRAPH_OUT`), or the opposite (`IGRAPH_IN`), or ignore edge directions completely (`IGRAPH_ALL`). It is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(s*|E|*|V|), where |V| is the number of vertices, |E| the number of edges and s the number of sources.

See also:

`igraph_distances()` for a non-algorithm-specific interface; `igraph_distances_dijkstra()` if you do not have negative edge weights.

Example 17.5. File `examples/simple/bellman_ford.c`

`igraph_distances_johnson` — Weighted shortest path lengths between vertices, using Johnson's algorithm.

```
igraph_error_t igraph_distances_johnson(
        const igraph_t *graph,
        igraph_matrix_t *res,
        igraph_vs_t from, igraph_vs_t to,
        const igraph_vector_t *weights,
        igraph_neimode_t mode);
```

This algorithm supports directed graphs with negative edge weights, and performs better than the Bellman-Ford method when distances are calculated from many different sources, the typical use case being all-pairs distance calculations. It works by using a single-source Bellman-Ford run to transform all edge weights to non-negative ones, then invoking Dijkstra's algorithm with the new weights. See the Wikipedia page for more details: http://en.wikipedia.org/wiki/Johnson's_algorithm.

If no edge weights are supplied, then the unweighted version, `igraph_distances()` is called. If none of the supplied edge weights are negative, then Dijkstra's algorithm is used by calling `igraph_distances_dijkstra()`.

Note that Johnson's algorithm applies only to directed graphs. This function rejects undirected graphs with *any* negative edge weights, even when the *from* and *to* vertices are all in connected components that are free of negative weights.

References:

Donald B. Johnson: Efficient Algorithms for Shortest Paths in Sparse Networks. J. ACM 24, 1 (1977), 1–13. https://doi.org/10.1145/321992.321993

Arguments:

graph: The input graph. If negative weights are present, it should be directed.

res: Pointer to an initialized matrix, the result will be stored here, one line for each source vertex, one column for each target vertex.

from: The source vertices.

to: The target vertices. It is not allowed to include a vertex twice or more.

weights: Optional edge weights. If it is a null-pointer, then the unweighted breadth-first search based `igraph_distances()` will be called. Edges with positive infinite weights are ignored.

mode: For directed graphs; whether to follow paths along edge directions (`IGRAPH_OUT`), or the opposite (`IGRAPH_IN`), or ignore edge directions completely (`IGRAPH_ALL`). It is ignored for undirected graphs. `IGRAPH_ALL` should not be used with negative weights.

Returns:

Error code.

Time complexity: O(s|V|log|V|+|V||E|), |V| and |E| are the number of vertices and edges, s is the number of source vertices.

See also:

`igraph_distances()` for a faster unweighted version, `igraph_distances_dijkstra()` if you do not have negative edge weights, `igraph_distances_bellman_ford()` if you only need to calculate shortest paths from a couple of sources.

`igraph_distances_floyd_warshall` — Weighted all-pairs shortest path lengths with the Floyd-Warshall algorithm.

```
igraph_error_t igraph_distances_floyd_warshall(
        const igraph_t *graph, igraph_matrix_t *res,
        igraph_vs_t from, igraph_vs_t to,
        const igraph_vector_t *weights, igraph_neimode_t mode,
        const igraph_floyd_warshall_algorithm_t method);
```

The Floyd-Warshall algorithm computes weighted shortest path lengths between all pairs of vertices at the same time. It is useful with very dense weighted graphs, as its running time is primarily determined by the vertex count, and is not sensitive to the graph density. In sparse graphs, other methods such as the Dijkstra or Bellman-Ford algorithms will perform significantly better.

In addition to the original Floyd-Warshall algorithm, igraph contains implementations of variants that offer better asymptotic complexity as well as better practical running times for most instances. See the reference below for more information.

Note that internally this function always computes the distance matrix for all pairs of vertices. The *from* and *to* parameters only serve to subset this matrix, but do not affect the time or memory taken by the calculation.

Reference:

Brodnik, A., Grgurovi#, M., Požar, R.: Modifications of the Floyd-Warshall algorithm with nearly quadratic expected-time, Ars Mathematica Contemporanea, vol. 22, issue 1, p. #P1.01 (2021). https://doi.org/10.26493/1855-3974.2467.497

Arguments:

graph: The graph object.

res: An intialized matrix, the distances will be stored here.

from: The source vertices.

to: The target vertices.

weights: The edge weights. If `NULL`, all weights are assumed to be 1. Negative weights are allowed, but the graph must not contain negative cycles. Edges with positive infinite weights are ignored.

mode: The type of shortest paths to be use for the calculation in directed graphs. Possible values:

- `IGRAPH_OUT` the outgoing paths are calculated.
- `IGRAPH_IN` the incoming paths are calculated.
- `IGRAPH_ALL` the directed graph is considered as an undirected one for the computation.

method: The type of the algorithm used.

- `IGRAPH_FLOYD_WARSHALL_AUTOMATIC` tries to select the best performing variant for the current graph; presently this option always uses the "Tree" method.
- `IGRAPH_FLOYD_WARSHALL_ORIGINAL` the basic Floyd-Warshall algorithm.
- `IGRAPH_FLOYD_WARSHALL_TREE` the "Tree" speedup of Brodnik et al., faster than the original algorithm in most cases.

Returns:

Error code. `IGRAPH_ENEGCYCLE` is returned if a negative-weight cycle is found.

See also:

`igraph_distances()`, `igraph_distances_dijkstra()`, `igraph_distances_bellman_ford()`, `igraph_distances_johnson()`

Time complexity: The original variant has complexity O(|V|^3 + |E|). The "Tree" variant has expected-case complexity of O(|V|^2 log^2 |V|) according to Brodnik et al., while its worst-time complexity remains O(|V|^3). Here |V| denotes the number of vertices and |E| is the number of edges.

`igraph_get_shortest_paths` — Shortest paths from a vertex.

```
igraph_error_t igraph_get_shortest_paths(
        const igraph_t *graph,
        const igraph_vector_t *weights,
        igraph_vector_int_list_t *vertices,
        igraph_vector_int_list_t *edges,
        igraph_int_t from, const igraph_vs_t to,
        igraph_neimode_t mode,
        igraph_vector_int_t *parents,
        igraph_vector_int_t *inbound_edges);
```

Finds unweighted shortest paths from a single source vertex to the specified sets of target vertices. If there is more than one geodesic between two vertices, this function gives only one of them. Use `igraph_get_all_shortest_paths()` to find *all* shortest paths.

Arguments:

graph: The graph object.

weights: Optional edge weights. If `NULL`, the graph is considered unweighted, i.e. all edge weights are equal to 1.

vertices: The result, the IDs of the vertices along the paths. This is a list of integer vectors where each element is an `igraph_vector_int_t` object. The list will be resized as needed. Supply a null pointer here if you don't need these vectors.

edges: The result, the IDs of the edges along the paths. This is a list of integer vectors where each element is an `igraph_vector_int_t` object. The list will be resized as needed. Supply a null pointer here if you don't need these vectors.

from: The ID of the vertex from/to which the geodesics are calculated.

to: Vertex sequence with the IDs of the vertices to/from which the shortest paths will be calculated. A vertex might be given multiple times.

mode: The type of shortest paths to be used for the calculation in directed graphs. Possible values:

- `IGRAPH_OUT` the outgoing paths are calculated.
- `IGRAPH_IN` the incoming paths are calculated.
- `IGRAPH_ALL` the directed graph is considered as an undirected one for the computation.

parents: A pointer to an initialized igraph vector or `NULL`. If not `NULL`, a vector containing the parent of each vertex in the single source shortest path tree is returned here. The parent of vertex `i` in the tree is the vertex from which vertex `i` was reached. The parent of the start vertex (in the *from* argument) is -1. If the parent is -2, 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: A pointer to an initialized igraph vector or `NULL`. If not `NULL`, a vector containing the inbound edge of each vertex in the single source shortest path tree is returned here. 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 -1 in the corresponding entry of the vector. Note that the search terminates if all the vertices in *to* are reached.

Returns:

Error code:

`IGRAPH_ENOMEM` not enough memory for temporary data.

`IGRAPH_EINVVID` *`from`* is invalid vertex ID

`IGRAPH_EINVMODE` invalid mode argument.

Time complexity: O(|V|+|E|), |V| is the number of vertices, |E| the number of edges in the graph.

See also:

`igraph_distances()` if you only need the path lengths but not the paths themselves; `igraph_get_shortest_paths_dijkstra()` for the weighted version; `igraph_get_all_shortest_paths()` to return all shortest paths between (source, target) pairs.

Example 17.6. File `examples/simple/igraph_get_shortest_paths.c`

`igraph_get_shortest_path` — Shortest path from one vertex to another one.

```
igraph_error_t igraph_get_shortest_path(
        const igraph_t *graph,
        const igraph_vector_t *weights,
        igraph_vector_int_t *vertices,
        igraph_vector_int_t *edges,
        igraph_int_t from, igraph_int_t to,
        igraph_neimode_t mode);
```

Calculates and returns a single unweighted shortest path from a given vertex to another one. If there is more than one shortest path between the two vertices, then an arbitrary one is returned.

This function is a wrapper to `igraph_get_shortest_paths()` for the special case when only one target vertex is considered.

Arguments:

`graph`: The input graph, it can be directed or undirected. Directed paths are considered in directed graphs.

`weights`: Optional edge weights. If `NULL`, the graph is considered unweighted, i.e. all edge weights are equal to 1.

`vertices`: Pointer to an initialized vector or a null pointer. If not a null pointer, then the vertex IDs along the path are stored here, including the source and target vertices.

`edges`: Pointer to an initialized vector or a null pointer. If not a null pointer, then the edge IDs along the path are stored here.

`from`: The ID of the source vertex.

`to`: The ID of the target vertex.

mode: A constant specifying how edge directions are considered in directed graphs. Valid modes are: `IGRAPH_OUT`, follows edge directions; `IGRAPH_IN`, follows the opposite directions; and `IGRAPH_ALL`, ignores edge directions. This argument is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges in the graph.

See also:

`igraph_get_shortest_paths()` for the version with more target vertices.

`igraph_get_shortest_paths_dijkstra` — Weighted shortest paths from a vertex.

```
igraph_error_t igraph_get_shortest_paths_dijkstra(
        const igraph_t *graph,
        igraph_vector_int_list_t *vertices,
        igraph_vector_int_list_t *edges,
        igraph_int_t from,
        igraph_vs_t to,
        const igraph_vector_t *weights,
        igraph_neimode_t mode,
        igraph_vector_int_t *parents,
        igraph_vector_int_t *inbound_edges);
```

Finds weighted shortest paths from a single source vertex to the specified sets of target vertices using Dijkstra's algorithm. If there is more than one path with the smallest weight between two vertices, this function gives only one of them. To find all such paths, use `igraph_get_all_shortest_paths_dijkstra()`.

Arguments:

graph: The graph object.

vertices: The result, the IDs of the vertices along the paths. This is a list of integer vectors where each element is an `igraph_vector_int_t` object. The list will be resized as needed. Supply a null pointer here if you don't need these vectors.

edges: The result, the IDs of the edges along the paths. This is a list of integer vectors where each element is an `igraph_vector_int_t` object. The list will be resized as needed. Supply a null pointer here if you don't need these vectors.

from: The id of the vertex from/to which the geodesics are calculated.

to: Vertex sequence with the IDs of the vertices to/from which the shortest paths will be calculated. A vertex might be given multiple times. *

weights: The edge weights. All edge weights must be non-negative for Dijkstra's algorithm to work. Additionally, no edge weight may be NaN. If either case does not hold, an error is returned. If this is a null pointer, then the unweighted version, `igraph_get_shortest_paths()` is called.

mode: The type of shortest paths to be use for the calculation in directed graphs. Possible values:

`IGRAPH_OUT` the outgoing paths are calculated.

`IGRAPH_IN` the incoming paths are calculated.

`IGRAPH_ALL` the directed graph is considered as an undirected one for the computation.

parents: A pointer to an initialized igraph vector or null. If not null, a vector containing the parent of each vertex in the single source shortest path tree is returned here. The parent of vertex i in the tree is the vertex from which vertex i was reached. The parent of the start vertex (in the `from` argument) is -1. If the parent is -2, 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: A pointer to an initialized igraph vector or null. If not null, a vector containing the inbound edge of each vertex in the single source shortest path tree is returned here. 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 -1 in the corresponding entry of the vector. Note that the search terminates if all the vertices in `to` are reached.

Returns:

Error code:

`IGRAPH_ENOMEM` not enough memory for temporary data.

`IGRAPH_EINVVID` *from* is invalid vertex ID

`IGRAPH_EINVMODE` invalid mode argument.

Time complexity: O(|E|log|V|+|V|), where |V| is the number of vertices and |E| is the number of edges

See also:

`igraph_distances_dijkstra()` if you only need the path lengths but not the paths themselves; `igraph_get_all_shortest_paths_dijkstra()` to find all shortest paths between (source, target) pairs; `igraph_get_shortest_paths()` for a non-algorithm-specific interface.

Example 17.7. File `examples/simple/igraph_get_shortest_paths_dijkstra.c`

`igraph_get_shortest_path_dijkstra` — Weighted shortest path from one vertex to another one (Dijkstra).

```
igraph_error_t igraph_get_shortest_path_dijkstra(const igraph_t *graph,
                                        igraph_vector_int_t *vertices,
                                        igraph_vector_int_t *edges,
                                        igraph_int_t from,
                                        igraph_int_t to,
```

```
                                        const igraph_vector_t *weights,
                                        igraph_neimode_t mode);
```

Finds a weighted shortest path from a single source vertex to a single target, using Dijkstra's algorithm. If more than one shortest path exists, an arbitrary one is returned.

This function is a special case (and a wrapper) to `igraph_get_shortest_paths_dijkstra()`.

Arguments:

graph: The input graph, it can be directed or undirected.

vertices: Pointer to an initialized vector or a null pointer. If not a null pointer, then the vertex IDs along the path are stored here, including the source and target vertices.

edges: Pointer to an initialized vector or a null pointer. If not a null pointer, then the edge IDs along the path are stored here.

from: The ID of the source vertex.

to: The ID of the target vertex.

weights: The edge weights. All edge weights must be non-negative for Dijkstra's algorithm to work. Additionally, no edge weight may be NaN. If either case does not hold, an error is returned. If this is a null pointer, then the unweighted version, `igraph_get_shortest_paths()` is called.

mode: A constant specifying how edge directions are considered in directed graphs. `IGRAPH_OUT` follows edge directions, `IGRAPH_IN` follows the opposite directions, and `IGRAPH_ALL` ignores edge directions. This argument is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(|E|log|V|+|V|), |V| is the number of vertices, |E| is the number of edges in the graph.

See also:

`igraph_get_shortest_paths_dijkstra()` for the version with more target vertices.

`igraph_get_shortest_paths_bellman_ford` — Weighted shortest paths from a vertex, allowing negative weights.

```
igraph_error_t igraph_get_shortest_paths_bellman_ford(
        const igraph_t *graph,
        igraph_vector_int_list_t *vertices,
        igraph_vector_int_list_t *edges,
        igraph_int_t from,
        igraph_vs_t to,
        const igraph_vector_t *weights,
```

```
        igraph_neimode_t mode,
        igraph_vector_int_t *parents,
        igraph_vector_int_t *inbound_edges);
```

This function calculates weighted shortest paths from or to a single vertex using the Bellman-Ford algorithm, whihc can handle negative weights. When there is more than one shortest path between two vertices, only one of them is returned. When there are no negative weights, `igraph_get_shortest_paths_dijkstra()` is likely to be faster.

Arguments:

graph: The input graph, can be directed.

vertices: The result, the IDs of the vertices along the paths. This is a list of integer vectors where each element is an `igraph_vector_int_t` object. The list will be resized as needed. Supply a null pointer here if you don't need these vectors.

edges: The result, the IDs of the edges along the paths. This is a list of integer vectors where each element is an `igraph_vector_int_t` object. The list will be resized as needed. Supply a null pointer here if you don't need these vectors.

from: The id of the vertex from/to which the geodesics are calculated.

to: Vertex sequence with the IDs of the vertices to/from which the shortest paths will be calculated. A vertex might be given multiple times.

weights: The edge weights. There must not be any cycle in the graph that has a negative total weight (since this would allow us to decrease the weight of any path containing at least a single vertex of this cycle infinitely). If this is a null pointer, then the unweighted version, `igraph_get_shortest_paths()` is called. Edges with positive infinite weights are ignored.

mode: For directed graphs; whether to follow paths along edge directions (`IGRAPH_OUT`), or the opposite (`IGRAPH_IN`), or ignore edge directions completely (`IGRAPH_ALL`). It is ignored for undirected graphs.

parents: A pointer to an initialized igraph vector or null. If not null, a vector containing the parent of each vertex in the single source shortest path tree is returned here. The parent of vertex i in the tree is the vertex from which vertex i was reached. The parent of the start vertex (in the `from` argument) is -1. If the parent is -2, 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: A pointer to an initialized igraph vector or null. If not null, a vector containing the inbound edge of each vertex in the single source shortest path tree is returned here. 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 -1 in the corresponding entry of the vector. Note that the search terminates if all the vertices in `to` are reached.

Returns:

Error code:

`IGRAPH_ENOMEM` Not enough memory for temporary data.

`IGRAPH_EINVAL` The weight vector doesn't math the number of edges.

`IGRAPH_EINVVID` *from* is invalid vertex ID

`IGRAPH_ENEGCYCLE` Bellman-ford algorithm encounted a negative cycle.

Time complexity: O(|E|*|V|), where |V| is the number of vertices, |E| the number of edges.

See also:

`igraph_distances_bellman_ford()` to compute only shortest path lengths, but not the paths themselves; `igraph_get_shortest_paths()` for a non-algorithm-specific interface.

`igraph_get_shortest_path_bellman_ford` — Weighted shortest path from one vertex to another one (Bellman-Ford).

```
igraph_error_t igraph_get_shortest_path_bellman_ford(const igraph_t *graph,
                                          igraph_vector_int_t *vertices,
                                          igraph_vector_int_t *edges,
                                          igraph_int_t from,
                                          igraph_int_t to,
                                          const igraph_vector_t *weights,
                                          igraph_neimode_t mode);
```

Finds a weighted shortest path from a single source vertex to a single target using the Bellman-Ford algorithm.

This function is a special case (and a wrapper) to `igraph_get_shortest_paths_bellman_ford()`.

Arguments:

`graph`: The input graph, it can be directed or undirected.

`vertices`: Pointer to an initialized vector or a null pointer. If not a null pointer, then the vertex IDs along the path are stored here, including the source and target vertices.

`edges`: Pointer to an initialized vector or a null pointer. If not a null pointer, then the edge IDs along the path are stored here.

`from`: The ID of the source vertex.

`to`: The ID of the target vertex.

`weights`: The edge weights. There must not be any cycle in the graph that has a negative total weight (since this would allow us to decrease the weight of any path containing at least a single vertex of this cycle infinitely). If this is a null pointer, then the unweighted version is called.

`mode`: A constant specifying how edge directions are considered in directed graphs. `IGRAPH_OUT` follows edge directions, `IGRAPH_IN` follows the opposite directions, and `IGRAPH_ALL` ignores edge directions. This argument is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(|E|log|E|+|V|), |V| is the number of vertices, |E| is the number of edges in the graph.

See also:

`igraph_get_shortest_paths_bellman_ford()` for the version with more target vertices.

`igraph_get_shortest_path_astar` — A* gives the shortest path from one vertex to another, with heuristic.

```
igraph_error_t igraph_get_shortest_path_astar(const igraph_t *graph,
                                              igraph_vector_int_t *vertices,
                                              igraph_vector_int_t *edges,
                                              igraph_int_t from,
                                              igraph_int_t to,
                                              const igraph_vector_t *weights,
                                              igraph_neimode_t mode,
                                              igraph_astar_heuristic_func_t *heuristic,
                                              void *extra);
```

Calculates a shortest path from a single source vertex to a single target, using the A* algorithm. A* tries to find a shortest path by starting at *`from`* and moving to vertices that lie on a path with the lowest estimated length. This length estimate is the sum of two numbers: the distance from the source (*`from`*) to the intermediate vertex, and the value returned by the heuristic function. The heuristic function provides an estimate the distance between intermediate candidate vertices and the target vertex *`to`*. The A* algorithm is guaranteed to give the correct shortest path (if one exists) only if the heuristic does not overestimate distances, i.e. if the heuristic function is *admissible*.

Arguments:

`graph`: The input graph, it can be directed or undirected.

`vertices`: Pointer to an initialized vector or the `NULL` pointer. If not `NULL`, then the vertex IDs along the path are stored here, including the source and target vertices.

`edges`: Pointer to an initialized vector or the `NULL` pointer. If not `NULL`, then the edge IDs along the path are stored here.

`from`: The ID of the source vertex.

`to`: The ID of the target vertex.

`weights`: Optional edge weights. Supply `NULL` for unweighted graphs. All edge weights must be non-negative. Additionally, no edge weight may be NaN. If either case does not hold, an error is returned. Edges with positive infinite weights are ignored.

`mode`: A constant specifying how edge directions are considered in directed graphs. `IGRAPH_OUT` follows edge directions, `IGRAPH_IN` follows the opposite directions, and `IGRAPH_ALL` ignores edge directions. This argument is ignored for undirected graphs.

`heuristic`: A function that provides distance estimates to the target vertex. See `igraph_astar_heuristic_func_t` for more information.

`extra`: This is passed on to the heuristic function.

Returns:

Error code.

Time complexity: In the worst case, O(|E|log|V|+|V|), where |V| is the number of vertices and |E| is the number of edges in the graph. The running time depends on the accuracy of the distance estimates returned by the heuristic function. Assuming that the heuristic is admissible, the better the estimates, the shortert the running time.

igraph_astar_heuristic_func_t — Distance estimator for A* algorithm.

```
typedef igraph_error_t igraph_astar_heuristic_func_t(
            igraph_real_t *result,
            igraph_int_t from, igraph_int_t to,
            void *extra);
```

`igraph_get_shortest_path_astar()` uses a heuristic based on a distance estimate to the target vertex to guide its search, and determine which vertex to try next. The heuristic function is expected to compute an estimate of the distance between *`from`* and *`to`*. In order for `igraph_get_shortest_path_astar()` to find an exact shortest path, the distance must not be overestimated, i.e. the heuristic function must be *admissible.*

Arguments:

`result`: The result of the heuristic, i.e. the estimated distance. A lower value will mean this vertex will be a better candidate for exploration.

`from`: The vertex ID of the candidate vertex will be passed here.

`to`: The vertex ID of the endpoint of the path, i.e. the `to` parameter given to `igraph_get_shortest_path_astar()`, will be passed here.

`extra`: The `extra` argument that was passed to `igraph_get_shortest_path_astar()`.

Returns:

Error code. Must return `IGRAPH_SUCCESS` if there were no errors. This can be used to break off the algorithm if something unexpected happens, like a failed memory allocation (`IGRAPH_ENOMEM`).

See also:

`igraph_get_shortest_path_astar()`

igraph_get_all_shortest_paths — All shortest paths (geodesics) from a vertex.

```
igraph_error_t igraph_get_all_shortest_paths(
        const igraph_t *graph,
        const igraph_vector_t *weights,
        igraph_vector_int_list_t *vertices,
```

```
        igraph_vector_int_list_t *edges,
        igraph_vector_int_t *nrgeo,
        igraph_int_t from, const igraph_vs_t to,
        igraph_neimode_t mode);
```

When there is more than one shortest path between two vertices, all of them will be returned. Every edge is considered separately, therefore in graphs with multi-edges, this function may produce a very large number of results.

Arguments:

graph: The graph object.

vertices: The result, the IDs of the vertices along the paths. This is a list of integer vectors where each element is an `igraph_vector_int_t` object. Each vector object contains the vertices along a shortest path from *from* to another vertex. The vectors are ordered according to their target vertex: first the shortest paths to vertex 0, then to vertex 1, etc. No data is included for unreachable vertices. The list will be resized as needed. Supply a null pointer here if you don't need these vectors.

edges: The result, the IDs of the edges along the paths. This is a list of integer vectors where each element is an `igraph_vector_int_t` object. Each vector object contains the edges along a shortest path from *from* to another vertex. The vectors are ordered according to their target vertex: first the shortest paths to vertex 0, then to vertex 1, etc. No data is included for unreachable vertices. The list will be resized as needed. Supply a null pointer here if you don't need these vectors.

nrgeo: Pointer to an initialized `igraph_vector_int_t` object or `NULL`. If not `NULL` the number of shortest paths from *from* are stored here for every vertex in the graph. Note that the values will be accurate only for those vertices that are in the target vertex sequence (see *to*), since the search terminates as soon as all the target vertices have been found.

from: The id of the vertex from/to which the geodesics are calculated.

to: Vertex sequence with the IDs of the vertices to/from which the shortest paths will be calculated. A vertex might be given multiple times.

mode: The type of shortest paths to be use for the calculation in directed graphs. Possible values:

- `IGRAPH_OUT` the lengths of the outgoing paths are calculated.
- `IGRAPH_IN` the lengths of the incoming paths are calculated.
- `IGRAPH_ALL` the directed graph is considered as an undirected one for the computation.

Returns:

Error code:

- `IGRAPH_ENOMEM` not enough memory for temporary data.
- `IGRAPH_EINVVID` *from* is invalid vertex ID.
- `IGRAPH_EINVMODE` invalid mode argument.

Added in version 0.2.

Time complexity: O(|V|+|E|) for most graphs, O(|V|^2) in the worst case.

`igraph_get_all_shortest_paths_dijkstra` — All weighted shortest paths (geodesics) from a vertex.

```
igraph_error_t igraph_get_all_shortest_paths_dijkstra(const igraph_t *graph,
        igraph_vector_int_list_t *vertices,
        igraph_vector_int_list_t *edges,
        igraph_vector_int_t *nrgeo,
        igraph_int_t from, igraph_vs_t to,
        const igraph_vector_t *weights,
        igraph_neimode_t mode);
```

Arguments:

graph: The graph object.

vertices: Pointer to an initialized integer vector list or NULL. If not NULL, then each vector object contains the vertices along a shortest path from *from* to another vertex. The vectors are ordered according to their target vertex: first the shortest paths to vertex 0, then to vertex 1, etc. No data is included for unreachable vertices.

edges: Pointer to an initialized integer vector list or NULL. If not NULL, then each vector object contains the edges along a shortest path from *from* to another vertex. The vectors are ordered according to their target vertex: first the shortest paths to vertex 0, then to vertex 1, etc. No data is included for unreachable vertices.

nrgeo: Pointer to an initialized igraph_vector_int_t object or NULL. If not NULL the number of shortest paths from *from* are stored here for every vertex in the graph. Note that the values will be accurate only for those vertices that are in the target vertex sequence (see *to*), since the search terminates as soon as all the target vertices have been found.

from: The id of the vertex from/to which the geodesics are calculated.

to: Vertex sequence with the IDs of the vertices to/from which the shortest paths will be calculated. A vertex might be given multiple times.

weights: The edge weights. All edge weights must be non-negative for Dijkstra's algorithm to work. Additionally, no edge weight may be NaN. If either case does not hold, an error is returned. If this is a null pointer, then the unweighted version, `igraph_get_all_shortest_paths()` is called.

mode: The type of shortest paths to be use for the calculation in directed graphs. Possible values:

- `IGRAPH_OUT` the outgoing paths are calculated.
- `IGRAPH_IN` the incoming paths are calculated.
- `IGRAPH_ALL` the directed graph is considered as an undirected one for the computation.

Returns:

Error code:

- `IGRAPH_ENOMEM` not enough memory for temporary data.
- `IGRAPH_EINVVID` *from* is an invalid vertex ID

`IGRAPH_EINVMODE` invalid mode argument.

Time complexity: O(|E|log|V|+|V|), where |V| is the number of vertices and |E| is the number of edges

See also:

`igraph_distances_dijkstra()` if you only need the path lengths but not the paths themselves, `igraph_get_all_shortest_paths()` if all edge weights are equal.

Example 17.8. File `examples/simple/igraph_get_all_shortest_paths_dijkstra.c`

`igraph_get_k_shortest_paths` — k shortest paths between two vertices.

```
igraph_error_t igraph_get_k_shortest_paths(
    const igraph_t *graph, const igraph_vector_t *weights,
    igraph_vector_int_list_t *vertex_paths,
    igraph_vector_int_list_t *edge_paths,
    igraph_int_t k, igraph_int_t from, igraph_int_t to,
    igraph_neimode_t mode
);
```

This function returns the *k* shortest paths between two vertices, in order of increasing lengths.

Reference:

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) https://doi.org/10.1090/qam/253822

Arguments:

graph: The graph object.

weights: The edge weights of the graph. Can be `NULL` for an unweighted graph. Infinite weights will be treated as missing edges.

vertex_paths: Pointer to an initialized list of integer vectors, the result will be stored here in `igraph_vector_int_t` objects. Each vector object contains the vertex IDs along the `k`th shortest path between *from* and *to*, where `k` is the vector list index. May be `NULL` if the vertex paths are not needed.

edge_paths: Pointer to an initialized list of integer vectors, the result will be stored here in `igraph_vector_int_t` objects. Each vector object contains the edge IDs along the `k`th shortest path between *from* and *to*, where `k` is the vector list index. May be `NULL` if the edge paths are not needed.

k: The number of paths.

from: The ID of the vertex from which the paths are calculated.

to: The ID of the vertex to which the paths are calculated.

mode: The type of paths to be used for the calculation in directed graphs. Possible values:

IGRAPH_OUT	The outgoing paths of *from* are calculated.
IGRAPH_IN	The incoming paths of *from* are calculated.
IGRAPH_ALL	The directed graph is considered as an undirected one for the computation.

Returns:

Error code:

IGRAPH_ENOMEM	Not enough memory for temporary data.
IGRAPH_EINVVID	*from* or *to* is an invalid vertex id.
IGRAPH_EINVMODE	Invalid mode argument.
IGRAPH_EINVAL	Invalid argument.

Time complexity: k |V| (|V| log|V| + |E|), where |V| is the number of vertices, and |E| is the number of edges.

See also:

igraph_get_all_simple_paths(), igraph_get_shortest_paths(), igraph_get_shortest_paths_dijkstra()

igraph_get_all_simple_paths — List all simple paths from one source.

```
igraph_error_t igraph_get_all_simple_paths(
        const igraph_t *graph,
        igraph_vector_int_list_t *res,
        igraph_int_t from, const igraph_vs_t to,
        igraph_neimode_t mode,
        igraph_int_t minlen, igraph_int_t maxlen,
        igraph_int_t max_results);
```

A path is simple if its vertices are unique, i.e. no vertex is visited more than once. This function returns paths in terms of their vertices and ignores multi-edges.

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 when the graph has many cycles. Consider using the *minlen* and *maxlen* parameters to restrict the paths that are returned.

Arguments:

graph:	The input graph.
res:	Initialized integer vector list. The paths are returned here in terms of their vertices. The paths are included in arbitrary order, as they are found.
from:	The start vertex.
to:	The target vertices.
mode:	The type of paths to be used for the calculation in directed graphs. Possible values:

IGRAPH_OUT outgoing paths are calculated.

IGRAPH_IN incoming paths are calculated.

IGRAPH_ALL the directed graph is considered as an undirected one for the computation.

minlen: Minimum length of paths that is considered. If negative or IGRAPH_UNLIMITED, no lower bound is used on the path lengths.

maxlen: Maximum length of paths that is considered. If negative or IGRAPH_UNLIMITED, no upper bound is used on the path lengths.

max_results: At most this many paths will be recorded. If negative, or IGRAPH_UNLIMITED, no limit is applied.

Returns:

Error code.

See also:

igraph_get_k_shortest_paths()

Time complexity: O(n!) in the worst case, n is the number of vertices.

igraph_average_path_length — The average shortest path length between all vertex pairs.

```
igraph_error_t igraph_average_path_length(
    const igraph_t *graph, const igraph_vector_t *weights, igraph_real_t *res,
    igraph_real_t *unconn_pairs, igraph_bool_t directed, igraph_bool_t unconn
);
```

If no vertex pairs can be included in the calculation, for example because the graph has fewer than two vertices, or if the graph has no edges and unconn is set to true, NaN is returned.

All distinct ordered vertex pairs are taken into account.

Arguments:

graph: The graph object.

weights: The edge weights. All edge weights must be non-negative for Dijkstra's algorithm to work. Additionally, no edge weight may be NaN. If either case does not hold, an error is returned. Passa a null pointer here if the graph is unweighted. Edges with positive infinite weight are ignored.

res: Pointer to a real number, this will contain the result.

unconn_pairs: Pointer to a real number. If not a null pointer, the number of ordered vertex pairs where the second vertex is unreachable from the first one will be stored here.

directed: Boolean, whether to consider directed paths. Ignored for undirected graphs.

unconn: If true, only those pairs are considered for the calculation between which there is a path. If false, IGRAPH_INFINITY is returned for disconnected graphs.

Returns:

Error code:

`IGRAPH_ENOMEM` not enough memory for data structures

`IGRAPH_EINVAL` invalid weight vector

Time complexity: O(|V| |E| log|E| + |V|), where |V| is the number of vertices and |E| is the number of edges.

Example 17.9. File `examples/simple/igraph_grg_game.c`

`igraph_path_length_hist` — Create a histogram of all shortest path lengths.

```
igraph_error_t igraph_path_length_hist(const igraph_t *graph, igraph_vector_t *
                            igraph_real_t *unconnected, igraph_bool_t directed)
```

This function calculates a histogram by calculating the shortest path length between all pairs of vertices. In directed graphs, both directions are considered, meaning that each vertex pair appears twice in the histogram.

Arguments:

`graph`: The input graph.

`res`: Pointer to an initialized vector, the result is stored here. The first (i.e. index 0) element contains the number of shortest paths of length 1, the second of length 2, etc. The supplied vector is resized as needed.

`unconnected`: Pointer to a real number, the number of vertex pairs for which the second vertex is not reachable from the first is stored here.

`directed`: Whether to consider directed paths in a directed graph. This argument is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(|V||E|), the number of vertices times the number of edges.

See also:

`igraph_average_path_length()` and `igraph_distances()`

`igraph_diameter` — Calculates the weighted diameter of a graph using Dijkstra's algorithm.

```
igraph_error_t igraph_diameter(
    const igraph_t *graph, const igraph_vector_t *weights, igraph_real_t *res,
    igraph_int_t *from, igraph_int_t *to, igraph_vector_int_t *vertex_path,
```

```
    igraph_vector_int_t *edge_path, igraph_bool_t directed, igraph_bool_t uncon
);
```

This function computes the weighted diameter of a graph, defined as the longest weighted shortest path, or the maximum weighted eccentricity of the graph's vertices. A corresponding shortest path, as well as its endpoints, can also be optionally computed.

If the graph has no vertices, `IGRAPH_NAN` is returned.

Arguments:

graph: The input graph, can be directed or undirected.

weights: The edge weights of the graph. Can be `NULL` for an unweighted graph. Edges with positive infinite weight are ignored.

res: Pointer to a real number, if not `NULL` then it will contain the diameter (the actual distance).

from: Pointer to an integer, if not `NULL` it will be set to the source vertex of the diameter path. If the graph has no diameter path, it will be set to -1.

to: Pointer to an integer, if not `NULL` it will be set to the target vertex of the diameter path. If the graph has no diameter path, it will be set to -1.

vertex_path: Pointer to an initialized vector. If not `NULL` the actual longest geodesic path in terms of vertices will be stored here. The vector will be resized as needed.

edge_path: Pointer to an initialized vector. If not `NULL` the actual longest geodesic path in terms of edges will be stored here. The vector will be resized as needed.

directed: Boolean, whether to consider directed paths. Ignored for undirected graphs.

unconn: What to do if the graph is not connected. If `true` the longest geodesic within a component will be returned, otherwise `IGRAPH_INFINITY` is returned.

Returns:

Error code.

Time complexity: O(|V||E|*log|E|), |V| is the number of vertices, |E| is the number of edges.

See also:

`igraph_radius()` for the minimum eccentricity.

Example 17.10. File `examples/simple/igraph_diameter.c`

`igraph_girth` — The girth of a graph is the length of the shortest cycle in it.

```
igraph_error_t igraph_girth(const igraph_t *graph, igraph_real_t *girth,
                 igraph_vector_int_t *circle);
```

The current implementation works for undirected graphs only, directed graphs are treated as undirected graphs. Self-loops and multiple edges are ignored, i.e. cycles of length 1 or 2 are not considered.

For graphs that contain no cycles, and only for such graphs, infinity is returned.

The first implementation of this function was done by Keith Briggs, thanks Keith.

Reference:

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. https://doi.org/10.1145/800105.803390

Arguments:

graph: The input graph. Edge directions will be ignored.

girth: Pointer to an `igraph_real_t`, if not `NULL` then the result will be stored here.

circle: Pointer to an initialized vector, the vertex IDs in the shortest circle will be stored here. If `NULL` then it is ignored.

Returns:

Error code.

Time complexity: O((|V|+|E|)^2), |V| is the number of vertices, |E| is the number of edges in the general case. If the graph has no cycles at all then the function needs O(|V|+|E|) time to realize this and then it stops.

Example 17.11. File `examples/simple/igraph_girth.c`

`igraph_eccentricity` — Eccentricity of some vertices.

```
igraph_error_t igraph_eccentricity(
    const igraph_t *graph, const igraph_vector_t *weights, igraph_vector_t *res
    igraph_vs_t vids, igraph_neimode_t mode
);
```

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.

Arguments:

graph: The input graph, it can be directed or undirected.

weights: The edge weights. All edge weights must be non-negative for Dijkstra's algorithm to work. Additionally, no edge weight may be NaN. If either case does not hold, an error is returned. Use a null pointer to calculate the unweighted eccentricities. Edges with positive infinite weights are ignored.

res: Pointer to an initialized vector, the result is stored here.

vids: The vertices for which the eccentricity is calculated.

mode: What kind of paths to consider for the calculation: `IGRAPH_OUT`, paths that follow edge directions; `IGRAPH_IN`, paths that follow the opposite directions; and `IGRAPH_ALL`, paths that ignore edge directions. This argument is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(|V| |E| log|V| + |V|), where |V| is the number of vertices, |E| the number of edges.

Example 17.12. File `examples/simple/igraph_eccentricity.c`

`igraph_radius` — Radius of a graph, using weighted edges.

```
igraph_error_t igraph_radius(
    const igraph_t *graph, const igraph_vector_t *weights,
    igraph_real_t *radius, igraph_neimode_t mode
);
```

The radius of a graph is the defined as the minimum eccentricity of its vertices, see `igraph_eccentricity()`.

Arguments:

`graph`: The input graph, it can be directed or undirected.

`weights`: The edge weights. All edge weights must be non-negative for Dijkstra's algorithm to work. Additionally, no edge weight may be NaN. If either case does not hold, an error is returned. If this is a null pointer, then the unweighted version, `igraph_radius()` is called. Edges with positive infinite weights are ignored.

`radius`: Pointer to a real variable, the result is stored here.

`mode`: What kind of paths to consider for the calculation: `IGRAPH_OUT`, paths that follow edge directions; `IGRAPH_IN`, paths that follow the opposite directions; and `IGRAPH_ALL`, paths that ignore edge directions. This argument is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(|V| |E| log|V| + |V|), where |V| is the number of vertices, |E| the number of edges.

See also:

`igraph_diameter()` for the maximum eccentricity, `igraph_eccentricity()` for eccentricities of all vertices.

`igraph_graph_center` — Central vertices of a graph.

```
igraph_error_t igraph_graph_center(
    const igraph_t *graph, const igraph_vector_t *weights, igraph_vector_int_t
    igraph_neimode_t mode
);
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

The central vertices of a graph are calculated by finding the vertices with the minimum eccentricity. The concept of the graph center is typically applied to (strongly) connected graphs. In disconnected graphs, the smallest eccentricity is taken across all components.

Arguments:

graph: The input graph, it can be directed or undirected.

weights: The edge weights. All edge weights must be non-negative for Dijkstra's algorithm to work. Additionally, no edge weight may be NaN. If either case does not hold, an error is returned. Pass a null pointer here if all edges have equal weight. Edges with positive infinite weights are ignored.

res: Pointer to an initialized vector, the result is stored here.

mode: What kind of paths to consider for the calculation: `IGRAPH_OUT`, paths that follow edge directions; `IGRAPH_IN`, paths that follow the opposite directions; and `IGRAPH_ALL`, paths that ignore edge directions. This argument is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(|V| |E| log|V| + |V|), where |V| is the number of vertices, |E| the number of edges.

See also:

`igraph_graph_center()`, `igraph_eccentricity()`, `igraph_radius()`

`igraph_pseudo_diameter` — Approximation and lower bound of the diameter of a graph.

```
igraph_error_t igraph_pseudo_diameter(
    const igraph_t *graph, const igraph_vector_t *weights,
    igraph_real_t *diameter, igraph_int_t vid_start,
    igraph_int_t *from, igraph_int_t *to,
    igraph_bool_t directed, igraph_bool_t unconn
);
```

This algorithm finds a pseudo-peripheral vertex and returns its eccentricity. This value can be used as an approximation and lower bound of the diameter of a graph.

A pseudo-peripheral vertex is a vertex v, such that for every vertex u which is as far away from v as possible, v is also as far away from u as possible. The process of finding one depends on where the search starts, and for a disconnected graph the maximum diameter found will be that of the component *vid_start* is in.

If the graph has no vertices, `IGRAPH_NAN` is returned.

Arguments:

graph: The input graph, can be directed or undirected.

weights: The edge weights of the graph. Can be NULL for an unweighted graph. All weights should be non-negative. Edges with positive infinite weights are ignored.

diameter: This will contain the pseudo-diameter.

vid_start: Id of the starting vertex. If this is negative, a random starting vertex is chosen.

from: If not NULL this will be set to the source vertex of the diameter path. If the graph has no diameter path, it will be set to -1.

to: If not NULL this will be set to the target vertex of the diameter path. If the graph has no diameter path, it will be set to -1.

directed: Boolean, whether to consider directed paths. Ignored for undirected graphs.

unconn: What to do if the graph is not connected. If true the longest geodesic within a component will be returned, otherwise IGRAPH_INFINITY is returned.

Returns:

Error code.

Time complexity: O(|V| |E| log|E|), |V| is the number of vertices, |E| is the number of edges.

See also:

igraph_diameter()

igraph_voronoi — Voronoi partitioning of a graph.

```
igraph_error_t igraph_voronoi(
        const igraph_t *graph,
        igraph_vector_int_t *membership,
        igraph_vector_t *distances,
        const igraph_vector_int_t *generators,
        const igraph_vector_t *weights,
        igraph_neimode_t mode,
        igraph_voronoi_tiebreaker_t tiebreaker);
```

To obtain a Voronoi partitioning of a graph, we start with a set of generator vertices, which will define the partitions. Each vertex is assigned to the generator vertex from (or to) which it is closest.

This function uses a BFS search for unweighted graphs and Dijkstra's algorithm for weights ones.

Arguments:

graph: The graph to partition.

membership: If not NULL, the Voronoi partition of each vertex will be stored here. membership[v] will be set to the index in *generators* of the generator vertex that v belongs to. For vertices that are not reachable from any generator, -1 is returned.

distances: If not NULL, the distance of each vertex to its respective generator will be stored here. For vertices which are not reachable from any generator, IGRAPH_INFINITY is returned.

generators: Vertex IDs of the generator vertices.

weights: The edge weights, interpreted as lengths in the shortest path calculation. All weights must be non-negative.

mode: In directed graphs, whether to compute distances *from* generator vertices to other vertices (IGRAPH_OUT), *to* generator vertices from other vertices (IGRAPH_IN), or ignore edge directions entirely (IGRAPH_ALL).

tiebreaker: Controls which generator vertex to assign a vertex to when it is at equal distance from/to multiple generator vertices.

IGRAPH_VORONOI_FIRST assign the vertex to the first generator vertex.

IGRAPH_VORONOI_LAST assign the vertex to the last generator vertex.

IGRAPH_VORONOI_RANDOM assign the vertex to a random generator vertex.
Note that IGRAPH_VORONOI_RANDOM does not guarantee that all partitions will be contiguous. For example, if 1 and 2 are chosen as generators for the graph 1-3, 2-3, 3-4, then 3 and 4 are at equal distance from both generators. If 3 is assigned to 2 but 4 is assigned to 1, then the partition {1, 4} will not induce a connected subgraph.

Returns:

Error code.

Time complexity: In weighted graphs, O((log |S|) |E| (log |V|) + |V|), and in unweighted graphs O((log |S|) |E| + |V|), where |S| is the number of generator vertices, and |V| and |E| are the number of vertices and edges in the graph.

See also:

igraph_distances(), igraph_distances_dijkstra().

igraph_vertex_path_from_edge_path — Converts a walk of edge IDs to the traversed vertex IDs.

```
igraph_error_t igraph_vertex_path_from_edge_path(
   const igraph_t *graph, igraph_int_t start,
   const igraph_vector_int_t *edge_path, igraph_vector_int_t *vertex_path,
   igraph_neimode_t mode
);
```

This function is useful when you have a sequence of edge IDs representing a continuous walk in a graph and you would like to obtain the vertex IDs that the walk traverses. The function is used implicitly by several shortest path related functions to convert a path of edge IDs to the corresponding representation that describes the path in terms of vertex IDs instead.

The result will always contain one more vertex than the number of provided edges. If no edges are given, the output will contain only the start vertex.

The walk is allowed to traverse the same vertex more than once. It is suitable for use on paths, cycles, or arbitrary walks.

Arguments:

graph: The graph that the edge IDs refer to.

start: The start vertex of the path. If negative, it is determined automatically. This is only possible if the walk contains at least one edge. If only one edge is present in an undirected walk, one of its endpoints will be selected arbitrarily.

edge_path: The sequence of edge IDs that describe the path.

vertex_path: The sequence of vertex IDs traversed will be returned here.

mode: A constant specifying how edge directions are considered in directed graphs. `IGRAPH_OUT` follows edge directions, `IGRAPH_IN` follows the opposite directions, and `IGRAPH_ALL` ignores edge directions. This argument is ignored for undirected graphs.

Returns:

Error code: `IGRAPH_ENOMEM` if there is not enough memory, `IGRAPH_EINVVID` if the start vertex is invalid, `IGRAPH_EINVAL` if the edge walk does not start at the given vertex or if there is at least one edge whose start vertex does not match the end vertex of the previous edge.

Time complexity: O(n) where n is the length of the walk.

Widest-path related functions

`igraph_get_widest_path` — Widest path from one vertex to another one.

```
igraph_error_t igraph_get_widest_path(const igraph_t *graph,
                                      igraph_vector_int_t *vertices,
                                      igraph_vector_int_t *edges,
                                      igraph_int_t from,
                                      igraph_int_t to,
                                      const igraph_vector_t *weights,
                                      igraph_neimode_t mode);
```

Calculates a single widest path from a single vertex to another one, using Dijkstra's algorithm.

This function is a special case (and a wrapper) to `igraph_get_widest_paths()`.

Arguments:

graph: The input graph, it can be directed or undirected.

vertices: Pointer to an initialized vector or `NULL`. If not `NULL`, then the vertex IDs along the path are stored here, including the source and target vertices.

edges: Pointer to an initialized vector or `NULL`. If not `NULL`, then the edge IDs along the path are stored here.

from: The ID of the source vertex.

to: The ID of the target vertex.

weights: The edge weights, interpreted as widths. Edge weights can be negative, but must not be NaN. Edges with negative infinite weight are ignored. The weight vector is required: if `NULL` is passed, an error is raised.

mode: The type of widest paths to be used for the calculation in directed graphs. Possible values:

`IGRAPH_OUT` the outgoing paths are calculated.

`IGRAPH_IN` the incoming paths are calculated.

`IGRAPH_ALL` the directed graph is considered as an undirected one for the computation.

Returns:

Error code.

Time complexity: O(|E|log|E|+|V|), |V| is the number of vertices, |E| is the number of edges in the graph.

See also:

`igraph_get_widest_paths()` for the version with more target vertices.

`igraph_get_widest_paths` — Widest paths from a single vertex.

```
igraph_error_t igraph_get_widest_paths(const igraph_t *graph,
                                       igraph_vector_int_list_t *vertices,
                                       igraph_vector_int_list_t *edges,
                                       igraph_int_t from,
                                       igraph_vs_t to,
                                       const igraph_vector_t *weights,
                                       igraph_neimode_t mode,
                                       igraph_vector_int_t *parents,
                                       igraph_vector_int_t *inbound_edges);
```

Calculates the widest paths from a single vertex to all other specified vertices, using a modified Dijkstra's algorithm. The width of a path is defined as the width of the narrowest edge in the path.If there is more than one path with the largest width between two vertices, this function gives only one of them.

Arguments:

graph: The graph object.

vertices: The result, the IDs of the vertices along the paths. This is a list of integer vectors where each element is an `igraph_vector_int_t` object. The list will be resized as needed. Supply a null pointer here if you don't need these vectors.

edges: The result, the IDs of the edges along the paths. This is a list of integer vectors where each element is an `igraph_vector_int_t` object. The vector list will be resized as needed. Supply a null pointer here if you don't need these vectors.

from: The ID of the vertex from/to which the widest paths are calculated.

to: Vertex sequence with the IDs of the vertices to/from which the widest paths will be calculated. A vertex may be given multiple times.

weights: The edge weights, interpreted as widths. Edge weights can be negative, but must not be NaN. Edges with negative infinite weight are ignored. The weight vector is required: if `NULL` is passed, an error is raised.

mode: The type of widest paths to be used for the calculation in directed graphs. Possible values:

`IGRAPH_OUT` the outgoing paths are calculated.

`IGRAPH_IN` the incoming paths are calculated.

`IGRAPH_ALL` the directed graph is considered as an undirected one for the computation.

parents: A pointer to an initialized igraph vector or null. If not null, a vector containing the parent of each vertex in the single source widest path tree is returned here. The parent of vertex `i` in the tree is the vertex from which vertex `i` was reached. The parent of the start vertex (in the *from* argument) is -1. If the parent is -2, it means that the given vertex was not reached from the source during the search. The search terminates when all the vertices in *to* have been reached.

inbound_edges: A pointer to an initialized igraph vector or `NULL`. If not `NULL`, a vector containing the inbound edge of each vertex in the single source widest path tree is returned here. 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 -1 in the corresponding entry of the vector. The search terminates when all the vertices in *to* have been reached.

Returns:

Error code:

`IGRAPH_ENOMEM` not enough memory for temporary data.

`IGRAPH_EINVVID` *from* is invalid vertex ID

`IGRAPH_EINVMODE` invalid mode argument.

Time complexity: O(|E|log|E|+|V|), where |V| is the number of vertices in the graph and |E| is the number of edges

See also:

`igraph_widest_path_widths_dijkstra()` or `igraph_widest_path_widths_floyd_warshall()` if you only need the widths of the paths but not the paths themselves.

`igraph_widest_path_widths_dijkstra` — Widths of widest paths between vertices.

```
igraph_error_t igraph_widest_path_widths_dijkstra(const igraph_t *graph,
                                    igraph_matrix_t *res,
                                    const igraph_vs_t from,
                                    const igraph_vs_t to,
```

```
                                        const igraph_vector_t *weights,
                                        igraph_neimode_t mode);
```

This function implements a modified Dijkstra's algorithm, which can find the widest path widths from a source vertex to all other vertices. The width of a path is defined as the width of the narrowest edge in the path.

This function allows specifying a set of source and target vertices. The algorithm is run independently for each source and the results are retained only for the specified targets. This implementation uses a binary heap for efficiency.

Arguments:

graph: The input graph, can be directed.

res: An initialized matrix, the result will be written here. The matrix will be resized as needed. Each row will contain the widths from a single source to the vertices given in the *to* argument. Unreachable vertices have width `-IGRAPH_INFINITY`, and vertices have a width of `IGRAPH_INFINITY` to themselves.

from: The source vertices.

to: The target vertices. It is not allowed to include a vertex twice or more.

weights: The edge weights, interpreted as widths. Edge weights can be negative, but must not be NaN. Edges with negative infinite weight are ignored. The weight vector is required: if `NULL` is passed, an error is raised.

mode: For directed graphs; whether to follow paths along edge directions (`IGRAPH_OUT`), or the opposite (`IGRAPH_IN`), or ignore edge directions completely (`IGRAPH_ALL`). It is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(s*(|E|log|E|+|V|)), where |V| is the number of vertices in the graph, |E| the number of edges and s the number of sources.

See also:

`igraph_widest_path_widths_floyd_warshall()` for a variant that runs faster on dense graphs.

`igraph_widest_path_widths_floyd_warshall` — Widths of widest paths between vertices.

```
igraph_error_t igraph_widest_path_widths_floyd_warshall(const igraph_t *graph,
                                        igraph_matrix_t *res,
                                        const igraph_vs_t from,
                                        const igraph_vs_t to,
                                        const igraph_vector_t *weights,
                                        igraph_neimode_t mode);
```

This function implements a modified Floyd-Warshall algorithm, to find the widest path widths between a set of source and target vertices. The width of a path is defined as the width of the narrowest edge in the path.

This algorithm is primarily useful for all-pairs path widths in very dense graphs, as its running time is manily determined by the vertex count, and is not sensitive to the graph density. In sparse graphs, other methods such as Dijkstra's algorithm, implemented in `igraph_widest_path_widths_dijkstra()` will perform better.

Note that internally this function always computes the path width matrix for all pairs of vertices. The *`from`* and *`to`* parameters only serve to subset this matrix, but do not affect the time taken by the calculation.

Arguments:

`graph`: The input graph, can be directed.

`res`: An initialized matrix, the result will be written here. The matrix will be resized as needed. Each row will contain the widths from a single source to the vertices given in the *`to`* argument. Unreachable vertices have width `-IGRAPH_INFINITY`, and vertices have a width of `IGRAPH_INFINITY` to themselves.

`from`: The source vertices.

`to`: The target vertices.

`weights`: The edge weights, interpreted as widths. Edge weights can be negative, but must not be NaN. Edges with negative infinite weight are ignored. The weight vector is required: if `NULL` is passed, an error is raised.

`mode`: For directed graphs; whether to follow paths along edge directions (`IGRAPH_OUT`), or the opposite (`IGRAPH_IN`), or ignore edge directions completely (`IGRAPH_ALL`). It is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(|V|^3), where |V| is the number of vertices in the graph.

See also:

`igraph_widest_path_widths_dijkstra()` for a variant that runs faster on sparse graphs.

Efficiency measures

`igraph_global_efficiency` — Calculates the global efficiency of a network.

```
igraph_error_t igraph_global_efficiency(
    const igraph_t *graph, const igraph_vector_t *weights,
    igraph_real_t *res, igraph_bool_t directed
);
```

The global efficiency of a network is defined as the average of inverse distances between all pairs of vertices: `E_g = 1/(N*(N-1)) sum_{i!=j} 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.

Reference:

V. Latora and M. Marchiori, Efficient Behavior of Small-World Networks, Phys. Rev. Lett. 87, 198701 (2001). https://dx.doi.org/10.1103/PhysRevLett.87.198701

Arguments:

graph: The graph object.

weights: The edge weights. All edge weights must be non-negative for Dijkstra's algorithm to work. Additionally, no edge weight may be NaN. If either case does not hold, an error is returned. If this is a null pointer, then the unweighted version, `igraph_average_path_length()` is used in calculating the global efficiency. Edges with positive infinite weights are ignored.

res: Pointer to a real number, this will contain the result.

directed: Boolean, whether to consider directed paths. Ignored for undirected graphs.

Returns:

Error code:

`IGRAPH_ENOMEM` not enough memory for data structures

`IGRAPH_EINVAL` invalid weight vector

Time complexity: O(|V| |E| log|E| + |V|) for weighted graphs and O(|V| |E|) for unweighted ones. |V| denotes the number of vertices and |E| denotes the number of edges.

See also:

`igraph_local_efficiency()`, `igraph_average_local_efficiency()`

`igraph_local_efficiency` — Calculates the local efficiency around each vertex in a network.

```
igraph_error_t igraph_local_efficiency(
    const igraph_t *graph, const igraph_vector_t *weights, igraph_vector_t *res,
    const igraph_vs_t vids, igraph_bool_t directed, igraph_neimode_t mode
);
```

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.

Reference:

I. Vragovi#, E. Louis, and A. Díaz-Guilera, Efficiency of informational transfer in regular and complex networks, Phys. Rev. E 71, 1 (2005). http://dx.doi.org/10.1103/PhysRevE.71.036122

Arguments:

graph: The graph object.

weights: The edge weights. All edge weights must be non-negative. Additionally, no edge weight may be NaN. If either case does not hold, an error is returned. If this is a null pointer, then the unweighted version, `igraph_average_path_length()` is called. Edges with positive infinite weights are ignored.

res: Pointer to an initialized vector, this will contain the result.

vids: The vertices around which the local efficiency will be calculated.

directed: Boolean, whether to consider directed paths. Ignored for undirected graphs.

mode: How to determine the local neighborhood of each vertex in directed graphs. Ignored in undirected graphs.

- `IGRAPH_ALL` take both in- and out-neighbours; this is a reasonable default for high-level interfaces.
- `IGRAPH_OUT` take only out-neighbours
- `IGRAPH_IN` take only in-neighbours

Returns:

Error code:

`IGRAPH_ENOMEM` not enough memory for data structures

`IGRAPH_EINVAL` invalid weight vector

Time complexity: O(|E|^2 log|E|) for weighted graphs and O(|E|^2) for unweighted ones. |E| denotes the number of edges.

See also:

`igraph_average_local_efficiency()`, `igraph_global_efficiency()`

`igraph_average_local_efficiency` — Calculates the average local efficiency in a network.

```
igraph_error_t igraph_average_local_efficiency(
    const igraph_t *graph, const igraph_vector_t *weights, igraph_real_t *res,
    igraph_bool_t directed, igraph_neimode_t mode
);
```

For the null graph, zero is returned by convention.

Arguments:

graph: The graph object.

weights: The edge weights. They must be all non-negative. If a null pointer is given, all weights are assumed to be 1. Edges with positive infinite weight are ignored.

res: Pointer to a real number, this will contain the result.

directed: Boolean, whether to consider directed paths. Ignored for undirected graphs.

mode: How to determine the local neighborhood of each vertex in directed graphs. Ignored in undirected graphs.

- `IGRAPH_ALL` take both in- and out-neighbours; this is a reasonable default for high-level interfaces.
- `IGRAPH_OUT` take only out-neighbours
- `IGRAPH_IN` take only in-neighbours

Returns:

Error code:

- `IGRAPH_ENOMEM` not enough memory for data structures
- `IGRAPH_EINVAL` invalid weight vector

Time complexity: O(|E|^2 log|E|) for weighted graphs and O(|E|^2) for unweighted ones. |E| denotes the number of edges.

See also:

`igraph_local_efficiency()`

Neighborhood of a vertex

`igraph_neighborhood_size` — Calculates the size of the neighborhood of a given vertex.

```
igraph_error_t igraph_neighborhood_size(const igraph_t *graph, igraph_vector_in
                              igraph_vs_t vids, igraph_int_t order,
                              igraph_neimode_t mode,
                              igraph_int_t mindist);
```

The neighborhood of a given order of a vertex includes all vertices which are closer to the vertex than the order. I.e., order 0 is always the vertex itself, order 1 is the vertex plus its immediate neighbors, order 2 is order 1 plus the immediate neighbors of the vertices in order 1, etc.

This function calculates the size of the neighborhood of the given order for the given vertices.

Arguments:

graph: The input graph.

res: Pointer to an initialized vector, the result will be stored here. It will be resized as needed.

vids: The vertices for which the calculation is performed.

order: Integer giving the order of the neighborhood. Negative values are treated as infinity.

mode: Specifies how to use the direction of the edges if a directed graph is analyzed. For `IGRAPH_OUT` only the outgoing edges are followed, so all vertices reachable from the source vertex in at most `order` steps are counted. For `IGRAPH_IN` all vertices from which the source vertex is reachable in at most `order` steps are counted. `IGRAPH_ALL` ignores the direction of the edges. This argument is ignored for undirected graphs.

mindist: The minimum distance to include a vertex in the counting. Vertices reachable with a path shorter than this value are excluded. If this is one, then the starting vertex is not counted. If this is two, then its neighbors are not counted either, etc.

Returns:

Error code.

See also:

`igraph_neighborhood()` for calculating the actual neighborhood; `igraph_neighborhood_graphs()` for creating separate graphs from the neighborhoods.

Time complexity: O(n*d*o), where n is the number vertices for which the calculation is performed, d is the average degree, o is the order.

`igraph_neighborhood` — Calculate the neighborhood of vertices.

```
igraph_error_t igraph_neighborhood(const igraph_t *graph, igraph_vector_int_lis
                        igraph_vs_t vids, igraph_int_t order,
                        igraph_neimode_t mode, igraph_int_t mindist);
```

The neighborhood of a given order of a vertex includes all vertices which are closer to the vertex than the order. I.e., order 0 is always the vertex itself, order 1 is the vertex plus its immediate neighbors, order 2 is order 1 plus the immediate neighbors of the vertices in order 1, etc.

This function calculates the vertices within the neighborhood of the specified vertices.

Arguments:

graph: The input graph.

res: An initialized list of integer vectors. The result of the calculation will be stored here. The list will be resized as needed.

vids: The vertices for which the calculation is performed.

order: Integer giving the order of the neighborhood. Negative values are treated as infinity.

mode: Specifies how to use the direction of the edges if a directed graph is analyzed. For `IGRAPH_OUT` only the outgoing edges are followed, so all vertices reachable from the source vertex in at most *`order`* steps are included. For `IGRAPH_IN` all vertices from which the source vertex is reachable in at most *`order`* steps are included. `IGRAPH_ALL` ignores the direction of the edges. This argument is ignored for undirected graphs.

mindist: The minimum distance to include a vertex in the counting. Vertices reachable with a path shorter than this value are excluded. If this is one, then the starting vertex is not counted. If this is two, then its neighbors are not counted either, etc.

Returns:

Error code.

See also:

`igraph_neighborhood_size()` to calculate the size of the neighborhood; `igraph_neighborhood_graphs()` for creating graphs from the neighborhoods; `igraph_subcomponent()` to find vertices reachable from a single vertex.

Time complexity: O(n*d*o), n is the number of vertices for which the calculation is performed, d is the average degree, o is the order.

`igraph_neighborhood_graphs` — Create graphs from the neighborhood(s) of some vertex/vertices.

```
igraph_error_t igraph_neighborhood_graphs(const igraph_t *graph, igraph_graph_l
                                igraph_vs_t vids, igraph_int_t order,
                                igraph_neimode_t mode,
                                igraph_int_t mindist);
```

The neighborhood of a given order of a vertex includes all vertices which are closer to the vertex than the order. Ie. order 0 is always the vertex itself, order 1 is the vertex plus its immediate neighbors, order 2 is order 1 plus the immediate neighbors of the vertices in order 1, etc.

This function finds every vertex in the neighborhood of a given parameter vertex and creates the induced subgraph from these vertices.

The first version of this function was written by Vincent Matossian, thanks Vincent.

Arguments:

graph: The input graph.

res: Pointer to a list of graphs, the result will be stored here. Each item in the list is an igraph_t object. The list will be resized as needed.

vids: The vertices for which the calculation is performed.

order: Integer giving the order of the neighborhood. Negative values are treated as infinity.

mode: Specifies how to use the direction of the edges if a directed graph is analyzed. For `IGRAPH_OUT` only the outgoing edges are followed, so all vertices reachable from the source vertex in at most *order* steps are counted. For `IGRAPH_IN` all vertices from which the source vertex is reachable in at most *order* steps are counted. `IGRAPH_ALL` ignores the direction of the edges. This argument is ignored for undirected graphs.

mindist: The minimum distance to include a vertex in the counting. Vertices reachable with a path shorter than this value are excluded. If this is one, then the starting vertex is not counted. If this is two, then its neighbors are not counted either, etc.

Returns:

Error code.

See also:

`igraph_neighborhood_size()` for calculating the neighborhood sizes only; `igraph_neighborhood()` for calculating the neighborhoods (but not creating graphs).

Time complexity: O(n*(|V|+|E|)), where n is the number vertices for which the calculation is performed, |V| and |E| are the number of vertices and edges in the original input graph.

Local scan statistics

The scan statistic is a summary of the locality statistics that is computed from the local neighborhood of each vertex. For details, see Priebe, C. E., Conroy, J. M., Marchette, D. J., Park, Y. (2005). Scan Statistics on Enron Graphs. Computational and Mathematical Organization Theory.

"Us" statistics

`igraph_local_scan_0` — Local scan-statistics, k=0

```
igraph_error_t igraph_local_scan_0(const igraph_t *graph, igraph_vector_t *res,
                        const igraph_vector_t *weights,
                        igraph_neimode_t mode);
```

K=0 scan-statistics is arbitrarily defined as the vertex degree for unweighted, and the vertex strength for weighted graphs. See `igraph_degree()` and `igraph_strength()`.

Arguments:

`graph`: The input graph

`res`: An initialized vector, the results are stored here.

`weights`: Weight vector for weighted graphs, null pointer for unweighted graphs.

`mode`: Type of the neighborhood, `IGRAPH_OUT` means outgoing, `IGRAPH_IN` means incoming and `IGRAPH_ALL` means all edges.

Returns:

Error code.

`igraph_local_scan_1_ecount` — Local scan-statistics, k=1, edge count and sum of weights

```
igraph_error_t igraph_local_scan_1_ecount(const igraph_t *graph, igraph_vector_
                                const igraph_vector_t *weights,
                                igraph_neimode_t mode);
```

Count the number of edges or the sum the edge weights in the 1-neighborhood of vertices.

Arguments:

`graph`: The input graph

`res`: An initialized vector, the results are stored here.

`weights`: Weight vector for weighted graphs, null pointer for unweighted graphs.

`mode`: Type of the neighborhood, `IGRAPH_OUT` means outgoing, `IGRAPH_IN` means incoming and `IGRAPH_ALL` means all edges.

Returns:

Error code.

`igraph_local_scan_k_ecount` — Sum the number of edges or the weights in k-neighborhood of every vertex.

```
igraph_error_t igraph_local_scan_k_ecount(const igraph_t *graph, igraph_int_t k,
                                          igraph_vector_t *res,
                                          const igraph_vector_t *weights,
                                          igraph_neimode_t mode);
```

Arguments:

graph: The input graph.

k: The size of the neighborhood, non-negative integer. The k=0 case is special, see `igraph_local_scan_0()`.

res: An initialized vector, the results are stored here.

weights: Weight vector for weighted graphs, null pointer for unweighted graphs.

mode: Type of the neighborhood, `IGRAPH_OUT` means outgoing, `IGRAPH_IN` means incoming and `IGRAPH_ALL` means all edges.

Returns:

Error code.

"Them" statistics

`igraph_local_scan_0_them` — Local THEM scan-statistics, k=0

```
igraph_error_t igraph_local_scan_0_them(const igraph_t *us, const igraph_t *them,
                                        igraph_vector_t *res,
                                        const igraph_vector_t *weights_them,
                                        igraph_neimode_t mode);
```

K=0 scan-statistics is arbitrarily defined as the vertex degree for unweighted, and the vertex strength for weighted graphs. See `igraph_degree()` and `igraph_strength()`.

Arguments:

us: The input graph, to use to extract the neighborhoods.

them: The input graph to use for the actually counting.

res: An initialized vector, the results are stored here.

weights_them: Weight vector for weighted graphs, null pointer for unweighted graphs.

mode: Type of the neighborhood, `IGRAPH_OUT` means outgoing, `IGRAPH_IN` means incoming and `IGRAPH_ALL` means all edges.

Returns:

Error code.

`igraph_local_scan_1_ecount_them` — Local THEM scan-statistics, k=1, edge count and sum of weights

```
igraph_error_t igraph_local_scan_1_ecount_them(const igraph_t *us, const igraph
                                    igraph_vector_t *res,
                                    const igraph_vector_t *weights_them,
                                    igraph_neimode_t mode);
```

Count the number of edges or the sum the edge weights in the 1-neighborhood of vertices.

Arguments:

us: The input graph to extract the neighborhoods.

them: The input graph to perform the counting.

weights_them: Weight vector for weighted graphs, null pointer for unweighted graphs.

mode: Type of the neighborhood, `IGRAPH_OUT` means outgoing, `IGRAPH_IN` means incoming and `IGRAPH_ALL` means all edges.

Returns:

Error code.

See also:

`igraph_local_scan_1_ecount()` for the US statistics.

`igraph_local_scan_k_ecount_them` — Local THEM scan-statistics, edge count or sum of weights.

```
igraph_error_t igraph_local_scan_k_ecount_them(const igraph_t *us, const igraph
                                    igraph_int_t k, igraph_vector_t *res,
                                    const igraph_vector_t *weights_them,
                                    igraph_neimode_t mode);
```

Count the number of edges or the sum the edge weights in the k-neighborhood of vertices.

Arguments:

us: The input graph to extract the neighborhoods.

them: The input graph to perform the counting.

k: The size of the neighborhood, non-negative integer. The k=0 case is special, see `igraph_local_scan_0_them()`.

res: An initialized vector, the results are stored here.

weights_them: Weight vector for weighted graphs, null pointer for unweighted graphs.

mode: Type of the neighborhood, `IGRAPH_OUT` means outgoing, `IGRAPH_IN` means incoming and `IGRAPH_ALL` means all edges.

Returns:

Error code.

See also:

`igraph_local_scan_1_ecount()` for the US statistics.

Pre-calculated subsets

`igraph_local_scan_neighborhood_ecount` — Local scan-statistics with pre-calculated neighborhoods

```
igraph_error_t igraph_local_scan_neighborhood_ecount(const igraph_t *graph,
        igraph_vector_t *res,
        const igraph_vector_t *weights,
        const igraph_vector_int_list_t *neighborhoods);
```

Count the number of edges, or sum the edge weights in neighborhoods given as a parameter.

Warning

Deprecated since version 0.10.0. Please do not use this function in new code; use `igraph_local_scan_subset_ecount()` instead.

Arguments:

graph:	The graph to perform the counting/summing in.
res:	Initialized vector, the result is stored here.
weights:	Weight vector for weighted graphs, null pointer for unweighted graphs.
neighborhoods:	List of igraph_vector_int_t objects, the neighborhoods, one for each vertex in the graph.

Returns:

Error code.

`igraph_local_scan_subset_ecount` — Local scan-statistics of subgraphs induced by subsets of vertices.

```
igraph_error_t igraph_local_scan_subset_ecount(const igraph_t *graph,
        igraph_vector_t *res,
        const igraph_vector_t *weights,
        const igraph_vector_int_list_t *subsets);
```

Count the number of edges, or sum the edge weights in induced subgraphs from vertices given as a parameter.

Arguments:

graph: The graph to perform the counting/summing in.

res: Initialized vector, the result is stored here.

weights: Weight vector for weighted graphs, null pointer for unweighted graphs.

subsets: List of igraph_vector_int_t objects, the vertex subsets.

Returns:

Error code.

Graph components

`igraph_subcomponent` — The vertices reachable from a given vertex.

```
igraph_error_t igraph_subcomponent(
    const igraph_t *graph, igraph_vector_int_t *res, igraph_int_t vertex,
    igraph_neimode_t mode
);
```

This function returns the set of vertices reachable from a specified vertex. In undirected graphs, this is simple the set of vertices within the same component.

Arguments:

graph: The graph object.

res: The result, vector with the IDs of the vertices reachable from *vertex*.

vertex: The id of the vertex of which the component is searched.

mode: Type of the component for directed graphs, possible values:

`IGRAPH_OUT` the set of vertices reachable *from* the *vertex*,

`IGRAPH_IN` the set of vertices from which the *vertex* is reachable.

`IGRAPH_ALL` the graph is considered as an undirected graph. Note that this is *not* the same as the union of the previous two.

Returns:

Error code:

`IGRAPH_ENOMEM` not enough memory for temporary data.

`IGRAPH_EINVVID` *vertex* is an invalid vertex ID

`IGRAPH_EINVMODE` invalid mode argument passed.

Time complexity: O(|V|+|E|), |V| and |E| are the number of vertices and edges in the graph.

See also:

`igraph_induced_subgraph()` if you want a graph object consisting only a given set of vertices and the edges between them; `igraph_reachability()` to efficiently compute the reachable set from *all* vertices; `igraph_neighborhood()` to find vertices within a given distance.

`igraph_connected_components` — Calculates the (weakly or strongly) connected components in a graph.

```
igraph_error_t igraph_connected_components(
    const igraph_t *graph, igraph_vector_int_t *membership,
    igraph_vector_int_t *csize, igraph_int_t *no, igraph_connectedness_t mode
);
```

When computing strongly connected components, the components will be indexed in topological order. In other words, vertex `v` is reachable from vertex `u` precisely when `membership[u] <= membership[v]`.

Arguments:

`graph`: The graph object to analyze.

`membership`: For every vertex the ID of its component is given. The vector has to be preinitialized and will be resized as needed. Alternatively this argument can be `NULL`, in which case it is ignored.

`csize`: For every component it gives its size, the order being defined by the component IDs. The vector must be preinitialized and will be resized as needed. Alternatively this argument can be `NULL`, in which case it is ignored.

`no`: Pointer to an integer, if not `NULL` then the number of components will be stored here.

`mode`: For directed graph this specifies whether to calculate weakly or strongly connected components. Possible values:

- `IGRAPH_WEAK` Compute weakly connected components, i.e. ignore edge directions.
- `IGRAPH_STRONG` Compute strongly connnected components, i.e. consider edge directions.

This parameter is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(|V|+|E|), where |V| and |E| are the number of vertices and edges in the graph.

Example 17.13. File `examples/simple/igraph_contract_vertices.c`

`igraph_is_connected` — Decides whether the graph is (weakly or strongly) connected.

```
igraph_error_t igraph_is_connected(const igraph_t *graph, igraph_bool_t *res,
                        igraph_connectedness_t mode);
```

A graph is considered connected when any of its vertices is reachable from any other. A directed graph with this property is called *strongly* connected. A directed graph that would be connected when ignoring the directions of its edges is called *weakly* connected.

A graph with zero vertices (i.e. the null graph) is *not* connected by definition. This behaviour changed in igraph 0.9; earlier versions assumed that the null graph is connected. See the following issue on Github for the argument that led us to change the definition: https://github.com/igraph/igraph/issues/1539

The return value of this function is cached in the graph itself, separately for weak and strong connectivity. Calling the function multiple times with no modifications to the graph in between will return a cached value in O(1) time.

Arguments:

graph: The graph object to analyze.

res: Pointer to a Boolean variable, the result will be stored here.

mode: For a directed graph this specifies whether to calculate weak or strong connectedness. Possible values: `IGRAPH_WEAK`, `IGRAPH_STRONG`. This argument is ignored for undirected graphs.

Returns:

Error code: `IGRAPH_EINVAL`: invalid mode argument.

See also:

`igraph_connected_components()` to find the connected components, `igraph_is_biconnected()` to check if the graph is 2-vertex-connected.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges in the graph.

`igraph_decompose` — Decomposes a graph into connected components.

```
igraph_error_t igraph_decompose(const igraph_t *graph, igraph_graph_list_t *com
                     igraph_connectedness_t mode,
                     igraph_int_t maxcompno, igraph_int_t minelements);
```

Creates a separate graph for each component of a graph. Note that the vertex IDs in the new graphs will be different than in the original graph, except when there is only a single component in the original graph.

Arguments:

graph: The original graph.

components: This list of graphs will contain the individual components. It should be initialized before calling this function and will be resized to hold the graphs.

mode: Either `IGRAPH_WEAK` or `IGRAPH_STRONG` for weakly and strongly connected components respectively.

maxcompno: The maximum number of components to return. The first *maxcompno* components will be returned (which hold at least *minelements* vertices, see the next parameter), the others will be ignored. Supply `-1` here if you don't want to limit the number of components.

minelements: The minimum number of vertices a component should contain in order to place it in the *components* vector. For example, supplying 2 here ignores isolated vertices.

Returns:

Error code, `IGRAPH_ENOMEM` if there is not enough memory to perform the operation.

Added in version 0.2.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

Example 17.14. File `examples/simple/igraph_decompose.c`

`igraph_reachability` — Calculates which vertices are reachable from each vertex in the graph.

```
igraph_error_t igraph_reachability(
        const igraph_t *graph,
        igraph_vector_int_t *membership,
        igraph_vector_int_t *csize,
        igraph_int_t *no_of_components,
        igraph_bitset_list_t *reach,
        igraph_neimode_t mode);
```

The resulting list will contain one bitset for each strongly connected component. The bitset for component i will have its j-th bit set, if vertex j is reachable from some vertex in component i in 0 or more steps. In particular, a vertex is always reachable from itself.

Arguments:

graph: The graph object to analyze.

membership: Pointer to an integer vector. For every vertex, the ID of its component is given. The vector will be resized as needed. This parameter must not be `NULL`.

csize: Pointer to an integer vector or `NULL`. For every component, it gives its size (vertex count), the order being defined by the component IDs. The vector will be resized as needed.

no_of_components: Pointer to an integer or `NULL`. The number of components will be stored here.

reach: A list of bitsets representing the result. It will be resized as needed. `reach[membership[u]][v]` is set to `true` if vertex `v` is reachable from vertex `u`.

mode: In directed graphs, controls the treatment of edge directions. Ignored in undirected graphs. With `IGRAPH_OUT`, reachability is computed by traversing edges along their direction. With `IGRAPH_IN`, edges are tra-

versed opposite to their direction. With IGRAPH_ALL, edge directions are ignored and the graph is treated as undirected.

Returns:

Error code: IGRAPH_ENOMEM if there is not enough memory to perform the operation.

See also:

igraph_connected_components() to find the connnected components of a graph; igraph_count_reachable() to count how many vertices are reachable from each vertex; igraph_subcomponent() to find which vertices are rechable from a single vertex.

Time complexity: O(|C||V|/w + |V| + |E|), where |C| is the number of strongly connected components (at most |V|), |V| is the number of vertices, and |E| is the number of edges respectively, and w is the bit width of igraph_int_t, typically the word size of the machine (32 or 64).

igraph_count_reachable — The number of vertices reachable from each vertex in the graph.

```
igraph_error_t igraph_count_reachable(const igraph_t *graph,
                                      igraph_vector_int_t *counts,
                                      igraph_neimode_t mode);
```

Arguments:

graph: The graph object to analyze.

counts: Integer vector. counts[v] will store the number of vertices reachable from vertex v, including v itself.

mode: In directed graphs, controls the treatment of edge directions. Ignored in undirected graphs. With IGRAPH_OUT, reachability is computed by traversing edges along their direction. With IGRAPH_IN, edges are traversed opposite to their direction. With IGRAPH_ALL, edge directions are ignored and the graph is treated as undirected.

Returns:

Error code: IGRAPH_ENOMEM if there is not enough memory to perform the operation.

See also:

igraph_connected_components(), igraph_transitive_closure()

Time complexity: O(|C||V|/w + |V| + |E|), where |C| is the number of strongly connected components (at most |V|), |V| is the number of vertices, and |E| is the number of edges respectively, and w is the bit width of igraph_int_t, typically the word size of the machine (32 or 64).

igraph_transitive_closure — Computes the transitive closure of a graph.

```
igraph_error_t igraph_transitive_closure(const igraph_t *graph, igraph_t *closu
```

The resulting graph will have an edge from vertex `i` to vertex `j` if `j` is reachable from `i`.

Arguments:

graph: The graph object to analyze.

closure: The resulting graph representing the transitive closure.

Returns:

Error code: `IGRAPH_ENOMEM` if there is not enough memory to perform the operation.

See also:

`igraph_connected_components()`, `igraph_count_reachable()`

Time complexity: O(|V|^2 + |E|), where |V| is the number of vertices, and |E| is the number of edges, respectively.

`igraph_biconnected_components` — Calculates biconnected components.

```
igraph_error_t igraph_biconnected_components(const igraph_t *graph,
                                  igraph_int_t *no,
                                  igraph_vector_int_list_t *tree_edges,
                                  igraph_vector_int_list_t *component_edges,
                                  igraph_vector_int_list_t *components,
                                  igraph_vector_int_t *articulation_points);
```

A graph is biconnected if the removal of any single vertex (and its incident 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 a 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, while isolated vertices are part of none at all.

Note that some authors do not consider the graph consisting of two connected vertices as biconnected, however, igraph does.

igraph does not consider components containing a single vertex only as being biconnected. Isolated vertices will not be part of any of the biconnected components. This means that checking whether there is a single biconnected component is not sufficient for determining if a graph is biconnected. Use `igraph_is_biconnected()` for this purpose.

Arguments:

graph: The input graph. It will be treated as undirected.

no: If not a `NULL` pointer, the number of biconnected components will be stored here.

tree_edges: If not a `NULL` pointer, then the found components are stored here, in a list of vectors. Every vector in the list is a biconnected component, represented by its edges. More precisely, a spanning tree of the biconnected component is returned.

component_edges: If not a NULL pointer, then the edges of the biconnected components are stored here, in the same form as for tree_edges.

components: If not a NULL pointer, then the vertices of the biconnected components are stored here, in the same format as for the previous two arguments.

articulation_points: If not a NULL pointer, then the articulation points of the graph are stored in this vector. A vertex is an articulation point if its removal increases the number of (weakly) connected components in the graph.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges, but only if you do not calculate *components* and *component_edges*. If you calculate *components*, then it is quadratic in the number of vertices. If you calculate *component_edges* as well, then it is cubic in the number of vertices.

See also:

igraph_articulation_points(), igraph_is_biconnected(), igraph_connected_components().

Example 17.15. File examples/simple/igraph_biconnected_components.c

igraph_articulation_points — Finds the articulation points in a graph.

```
igraph_error_t igraph_articulation_points(const igraph_t *graph, igraph_vector_
```

A vertex is an articulation point if its removal increases the number of (weakly) connected components in the graph.

Note that a graph without any articulation points is not necessarily biconnected. Counterexamples are the two-vertex complete graph as well as empty graphs. Use igraph_is_biconnected() to check whether a graph is biconnected.

Arguments:

graph: The input graph. It will be treated as undirected.

res: Pointer to an initialized vector, the articulation points will be stored here.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

See also:

igraph_biconnected_components(), igraph_is_bipartite(), igraph_connected_components(), igraph_bridges()

igraph_bridges — Finds all bridges in a graph.

```
igraph_error_t igraph_bridges(const igraph_t *graph, igraph_vector_int_t *bridg
```

An edge is a bridge if its removal increases the number of (weakly) connected components in the graph.

Arguments:

graph: The input graph. It will be treated as undirected.

bridges: Pointer to an initialized vector, the bridges will be stored here as edge indices.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

See also:

igraph_articulation_points(), igraph_biconnected_components(), igraph_connected_components()

igraph_is_biconnected — Checks whether a graph is biconnected.

```
igraph_error_t igraph_is_biconnected(const igraph_t *graph, igraph_bool_t *res)
```

A graph is biconnected if the removal of any single vertex (and its incident 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.

Arguments:

graph: The input graph. It will be treated as undirected.

res: If not a NULL pointer, the result will be returned here.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

See also:

igraph_articulation_points(), igraph_biconnected_components().

Example 17.16. File examples/simple/igraph_is_biconnected.c

Percolation

igraph_site_percolation — The size of the largest component as vertices are added to a graph.

```
igraph_error_t igraph_site_percolation(
        const igraph_t *graph,
        igraph_vector_int_t *giant_size,
        igraph_vector_int_t *edge_count,
        const igraph_vector_int_t *vertex_order);
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

Calculates the site percolation curve, i.e. the size of the largest connected component as vertices are added in the given order. If both *giant_size* and *vertex_order* are reversed, it is the size of the largest component as vertices are removed from the graph. If no vertex order is given, a random one will be used.

Arguments:

graph: The graph that vertices are assumed to be in. Edge directions are ignored.

giant_size: `giant_size[i]` will contain the size of the largest component after having added the vertex with index `vertex_order[i]`.

edge_count: `edge_count[i]` will contain the numer of edges in the graph having added the vertex with index `vertex_order[i]`.

vertex_order: The order the vertices are added in. Must not contain duplicates. If `NULL`, a random order will be used.

Returns:

Error code.

See also:

`igraph_bond_percolation()` to compute the edge percolation curve; `igraph_connected_components()` to find the size of connected components.

Time complexity: O(|V| + |E| a(|E|)) where a is the inverse Ackermann function, for all practical purposes it is not above 5.

igraph_bond_percolation — The size of the largest component as edges are added to a graph.

```
igraph_error_t igraph_bond_percolation(
        const igraph_t *graph,
```

```
        igraph_vector_int_t *giant_size,
        igraph_vector_int_t *vertex_count,
        const igraph_vector_int_t *edge_order);
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

Calculates the bond percolation curve, i.e. the size of the largest connected component as edges are added to the graph in the order given. If both *giant_size* and *edge_order* are reversed, it is the size of the largest component as edges are removed from the graph. If no edge order is given, a random one will be used.

Arguments:

graph: The graph that edges are assumed to be in. Edge directions are ignored.

giant_size: giant_size[i] will contain the size of the largest component after having added the edge with index edge_order[i].

vertex_count: vertex_count[i] will contain the number of vertices that have at least one incident edge after adding the edge with index edge_order[i].

edge_order: The order the edges are added in. Must not contain duplicates. If NULL, a random order will be used.

Returns:

Error code.

See also:

igraph_edgelist_percolation() to specify the edges to be added by their endpoints; igraph_site_percolation() to compute the vertex percolation curve; igraph_connected_components() to find the size of connected components.

Time complexity: O(|V| + |E| a(|E|)) where a is the inverse Ackermann function, for all practical purposes it is not above 5.

igraph_edgelist_percolation — The size of the largest component as vertex pairs are connected.

```
igraph_error_t igraph_edgelist_percolation(
        const igraph_vector_int_t *edges,
        igraph_vector_int_t *giant_size,
        igraph_vector_int_t *vertex_count);
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

Calculates the size of the largest connected component as edges are added to a graph in the given order. This function differs from `igraph_bond_percolation()` in that it take a list of vertex pairs as input.

Arguments:

`edges`: Vector of edges, where the i-th edge has endpoints `edges[2i]` and `edges[2i+1]`.

`giant_size`: `giant_size[i]` will contain the size of the largest connected component after edge `i` is added.

`vertex_count`: `vertex_count[i]` will contain the number of vertices with at least one edge after edge `i` is added.

Returns:

Error code.

See also:

`igraph_bond_percolation()` to specify edges by their ID in a graph object.

Time complexity: O(|E| a(|E|)) where a is the inverse Ackermann function, for all practical purposes it is not above 5.

Degree sequences

`igraph_is_graphical` — Is there a graph with the given degree sequence?

```
igraph_error_t igraph_is_graphical(const igraph_vector_int_t *out_degrees,
                        const igraph_vector_int_t *in_degrees,
                        const igraph_edge_type_sw_t allowed_edge_types,
                        igraph_bool_t *res);
```

Determines whether a sequence of integers can be the degree sequence of some graph. The classical concept of graphicality assumes simple graphs. This function can perform the check also when either self-loops, multi-edge, or both are allowed in the graph.

For simple undirected graphs, the Erd#s-Gallai conditions are checked using the linear-time algorithm of Cloteaux. If both self-loops and multi-edges are allowed, it is sufficient to chek that that sum of degrees is even. If only multi-edges are allowed, but not self-loops, there is an additional condition that the sum of degrees be no smaller than twice the maximum degree. If at most one self-loop is allowed per vertex, but no multi-edges, a modified version of the Erd#s-Gallai conditions are used (see Cairns & Mendan).

For simple directed graphs, the Fulkerson-Chen-Anstee theorem is used with the relaxation by Berger. If both self-loops and multi-edges are allowed, then it is sufficient to check that the sum of in- and out-degrees is the same. If only multi-edges are allowed, but not self loops, there is an additional condition that the sum of out-degrees (or equivalently, in-degrees) is no smaller than the maximum total degree. If single self-loops are allowed, but not multi-edges, the problem is equivalent to realizability as a simple bipartite graph, thus the Gale-Ryser theorem can be used; see `igraph_is_bigraphical()` for more information.

References:

P. Erd#s and T. Gallai, Gráfok el#írt fokú pontokkal, Matematikai Lapok 11, pp. 264–274 (1960). https://users.renyi.hu/~p_erdos/1961-05.pdf

Z. Király, Recognizing graphic degree sequences and generating all realizations. TR-2011-11, Egerváry Research Group, H-1117, Budapest, Hungary. ISSN 1587-4451 (2012). https://egres.elte.hu/tr/egres-11-11.pdf

B. Cloteaux, Is This for Real? Fast Graphicality Testing, Comput. Sci. Eng. 17, 91 (2015). https://dx.doi.org/10.1109/MCSE.2015.125

A. Berger, A note on the characterization of digraphic sequences, Discrete Math. 314, 38 (2014). https://dx.doi.org/10.1016/j.disc.2013.09.010

G. Cairns and S. Mendan, Degree Sequence for Graphs with Loops (2013). https://arxiv.org/abs/1303.2145v1

Arguments:

out_degrees: A vector of integers specifying the degree sequence for undirected graphs or the out-degree sequence for directed graphs.

in_degrees: A vector of integers specifying the in-degree sequence for directed graphs. For undirected graphs, it must be `NULL`.

allowed_edge_types: The types of edges to allow in the graph. See `igraph_edge_type_sw_t` for details.

`IGRAPH_SIMPLE_SW`	simple graphs (i.e. no self-loops or multi-edges allowed).
`IGRAPH_LOOPS_SW`	single self-loops are allowed, but not multi-edges.
`IGRAPH_MULTI_SW`	multi-edges are allowed, but not self-loops.
`IGRAPH_LOOPS_SW | IGRAPH_MULTI_SW`	both self-loops and multi-edges are allowed.

res: Pointer to a Boolean. The result will be stored here.

Returns:

Error code.

See also:

`igraph_is_bigraphical()` to check if a bi-degree-sequence can be realized as a bipartite graph; `igraph_realize_degree_sequence()` to construct a graph with a given degree sequence.

Time complexity: O(n), where n is the length of the degree sequence(s).

`igraph_is_bigraphical` — Is there a bipartite graph with the given bi-degree-sequence?

```
igraph_error_t igraph_is_bigraphical(const igraph_vector_int_t *degrees1,
```

```
                        const igraph_vector_int_t *degrees2,
                        const igraph_edge_type_sw_t allowed_edge_types,
                        igraph_bool_t *res);
```

Determines whether two sequences of integers can be the degree sequences of a bipartite graph. Such a pair of degree sequence is called *bigraphical*.

When multi-edges are allowed, it is sufficient to check that the sum of degrees is the same in the two partitions. For simple graphs, the Gale-Ryser theorem is used with Berger's relaxation.

References:

H. J. Ryser, Combinatorial Properties of Matrices of Zeros and Ones, Can. J. Math. 9, 371 (1957). https://dx.doi.org/10.4153/cjm-1957-044-3

D. Gale, A theorem on flows in networks, Pacific J. Math. 7, 1073 (1957). https://dx.doi.org/10.2140/pjm.1957.7.1073

A. Berger, A note on the characterization of digraphic sequences, Discrete Math. 314, 38 (2014). https://dx.doi.org/10.1016/j.disc.2013.09.010

Arguments:

degrees1: A vector of integers specifying the degrees in the first partition

degrees2: A vector of integers specifying the degrees in the second partition

allowed_edge_types: The types of edges to allow in the graph:

`IGRAPH_SIMPLE_SW` simple graphs (i.e. no multi-edges allowed).

`IGRAPH_MULTI_SW` multi-edges are allowed.

res: Pointer to a Boolean. The result will be stored here.

Returns:

Error code.

See also:

`igraph_is_graphical()`

Time complexity: O(n), where n is the length of the larger degree sequence.

Centrality measures

`igraph_closeness` — Closeness centrality calculations for some vertices.

```
igraph_error_t igraph_closeness(const igraph_t *graph, igraph_vector_t *res,
                      igraph_vector_int_t *reachable_count, igraph_bool_t *all_r
                      const igraph_vs_t vids, igraph_neimode_t mode,
                      const igraph_vector_t *weights,
                      igraph_bool_t normalized);
```

The closeness centrality of a vertex measures how easily other vertices can be reached from it (or the other way: how easily it can be reached from the other vertices). It is defined as the inverse of the mean distance to (or from) all other vertices.

Closeness centrality is meaningful only for connected graphs. If the graph is not connected, igraph computes the inverse of the mean distance to (or from) all *reachable* vertices. In undirected graphs, this is equivalent to computing the closeness separately in each connected component. The optional *all_reachable* output parameter is provided to help detect when the graph is disconnected.

While there is no universally adopted definition of closeness centrality for disconnected graphs, there have been some attempts for generalizing the concept to the disconnected case. One type of approach considers the mean distance only to reachable vertices, then re-scales the obtained certrality score by a factor that depends on the number of reachable vertices (i.e. the size of the component in the undirected case). To facilitate computing these generalizations of closeness centrality, the number of reachable vertices (not including the starting vertex) is returned in *reachable_count*.

In disconnected graphs, consider using the harmonic centrality, computable using `igraph_harmonic_centrality()`.

For isolated vertices, i.e. those having no associated paths, NaN is returned.

Arguments:

graph: The graph object.

res: The result of the computation, a vector containing the closeness centrality scores for the given vertices.

reachable_count: If not `NULL`, this vector will contain the number of vertices reachable from each vertex for which the closeness is calculated (not including that vertex).

all_reachable: Pointer to a Boolean. If not `NULL`, it indicates if all vertices of the graph were reachable from each vertex in *vids*. If false, the graph is non-connected. If true, and the graph is undirected, or if the graph is directed and *vids* contains all vertices, then the graph is connected.

vids: The vertices for which the closeness centrality will be computed.

mode: The type of shortest paths to be used for the calculation in directed graphs. Possible values:

- `IGRAPH_OUT` the lengths of the outgoing paths are calculated.
- `IGRAPH_IN` the lengths of the incoming paths are calculated.
- `IGRAPH_ALL` the directed graph is considered as an undirected one for the computation.

weights: An optional vector containing edge weights for weighted closeness. NaN values re not allowed as weights. Supply a null pointer here for traditional, unweighted closeness.

normalized: If true, the inverse of the mean distance to reachable vertices is returned. If false, the inverse of the sum of distances is returned.

Returns:

Error code:

`IGRAPH_ENOMEM` not enough memory for temporary data.

`IGRAPH_EINVVID` invalid vertex ID passed.

`IGRAPH_EINVMODE` invalid mode argument.

Time complexity: O(n|E|) for the unweighted case and O(n|E|log|V|+|V|) for the weighted case, where n is the number of vertices for which the calculation is done, |V| is the number of vertices and |E| is the number of edges in the graph.

See also:

Other centrality types: `igraph_degree()`, `igraph_betweenness()`, `igraph_harmonic_centrality()`. See `igraph_closeness_cutoff()` for the range-limited closeness centrality.

`igraph_harmonic_centrality` — Harmonic centrality for some vertices.

```
igraph_error_t igraph_harmonic_centrality(const igraph_t *graph, igraph_vector_
                                const igraph_vs_t vids, igraph_neimode_t mode,
                                const igraph_vector_t *weights,
                                igraph_bool_t normalized);
```

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.

References:

M. Marchiori and V. Latora, Harmony in the small-world, Physica A 285, pp. 539-546 (2000). https://doi.org/10.1016/S0378-4371%2800%2900311-3

Y. Rochat, Closeness Centrality Extended to Unconnected Graphs: the Harmonic Centrality Index, ASNA 2009. https://infoscience.epfl.ch/record/200525

S. Vigna and P. Boldi, Axioms for Centrality, Internet Mathematics 10, (2014). https://doi.org/10.1080/15427951.2013.865686

Arguments:

`graph`: The graph object.

`res`: The result of the computation, a vector containing the harmonic centrality scores for the given vertices.

`vids`: The vertices for which the harmonic centrality will be computed.

`mode`: The type of shortest paths to be used for the calculation in directed graphs. Possible values:

- `IGRAPH_OUT` the lengths of the outgoing paths are calculated.
- `IGRAPH_IN` the lengths of the incoming paths are calculated.
- `IGRAPH_ALL` the directed graph is considered as an undirected one for the computation.

`weights`: An optional vector containing edge weights for weighted harmonic centrality. No edge weight may be NaN. If `NULL`, all weights are considered to be one.

normalized: Boolean, whether to normalize the result. 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.

Returns:

Error code:

`IGRAPH_ENOMEM` not enough memory for temporary data.

`IGRAPH_EINVVID` invalid vertex ID passed.

`IGRAPH_EINVMODE` invalid mode argument.

Time complexity: O(n|E|) for the unweighted case and O(n*|E|log|V|+|V|) for the weighted case, where n is the number of vertices for which the calculation is done, |V| is the number of vertices and |E| is the number of edges in the graph.

See also:

Other centrality types: `igraph_closeness()`, `igraph_degree()`, `igraph_betweenness()`.

`igraph_betweenness` — Betweenness centrality of some vertices.

```
igraph_error_t igraph_betweenness(
        const igraph_t *graph, const igraph_vector_t *weights,
        igraph_vector_t *res, igraph_vs_t vids,
        igraph_bool_t directed, igraph_bool_t normalized);
```

The betweenness centrality of a vertex `v` is the number of shortest paths passing through it. If there is more than one shortest path between two vertices, the fraction of these passing through `v` is counted.

Reference:

Ulrik Brandes: A faster algorithm for betweenness centrality. The Journal of Mathematical Sociology, 25(2), 163–177 (2001). https://doi.org/10.1080/0022250X.2001.9990249

Arguments:

graph: The graph object.

weights: An optional vector containing edge weights for calculating weighted betweenness. No edge weight may be NaN. Supply a null pointer here for unweighted betweenness.

res: The result of the computation, a vector containing the betweenness scores for the specified vertices.

vids: The vertices for which the range-limited betweenness centrality scores will be returned. This paramerer is for convenience only and does not affect performance. Internally, the betewenness of all vertices is calculated.

directed: If true directed paths will be considered for directed graphs. It is ignored for undirected graphs.

normalized: Whether to normalize betweenness scores by the number of vertex pairs. In directed graphs, the number of ordered vertex pairs, in undirected graphs the number of unordered vertex pairs is used.

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory for temporary data. `IGRAPH_EINVVID`, invalid vertex ID passed in *`vids`*.

Time complexity: O(|V||E|), |V| and |E| are the number of vertices and edges in the graph. Note that the time complexity is independent of the number of vertices for which the score is calculated.

See also:

`igraph_edge_betweenness()` for calculating the betweenness score of the edges in a graph; `igraph_betweenness_cutoff()` to calculate the range-limited betweenness of the vertices in a graph; `igraph_betweenness_subset()` to consider shortest paths only between two vertex subsets for calculating betweenness.

`igraph_edge_betweenness` — Betweenness centrality of the edges.

```
igraph_error_t igraph_edge_betweenness(
        const igraph_t *graph, const igraph_vector_t *weights,
        igraph_vector_t *res, igraph_es_t eids,
        igraph_bool_t directed, igraph_bool_t normalized);
```

The betweenness centrality of an edge `e` is the number of shortest paths passing through it. If there is more than one shortest path between two vertices, the fraction of these passing through `e` is counted.

Reference:

Ulrik Brandes: A faster algorithm for betweenness centrality. The Journal of Mathematical Sociology, 25(2), 163–177 (2001). https://doi.org/10.1080/0022250X.2001.9990249

Arguments:

graph: The graph object.

weights: An optional weight vector for weighted edge betweenness. No edge weight may be NaN. Supply a null pointer here for the unweighted version.

res: The result of the computation, vector containing the betweenness scores for the edges.

eids: The edges for which the betweenness centrality will be returned. This parameter is for convenience only, and does not affect performance. Internally, the betweenness is be calculated for all edges.

directed: If true directed paths will be considered for directed graphs. It is ignored for undirected graphs.

normalized: Whether to normalize betweenness scores by the number of vertex pairs. In directed graphs, the number of ordered vertex pairs, in undirected graphs the number of unordered vertex pairs is used.

Returns:

Error code: IGRAPH_ENOMEM, not enough memory for temporary data.

Time complexity: O(|V||E|), |V| and |E| are the number of vertices and edges in the graph.

See also:

igraph_betweenness() for calculating the betweenness score of the vertices in a graph; igraph_edge_betweenness_cutoff() to compute the range-limited betweenness score of the edges in a graph; igraph_edge_betweenness_subset() to consider shortest paths only between two vertex subsets for calculating betweenness.

igraph_pagerank_algo_t — PageRank algorithm implementation.

```
typedef enum {
    IGRAPH_PAGERANK_ALGO_ARPACK = 1,
    IGRAPH_PAGERANK_ALGO_PRPACK = 2
} igraph_pagerank_algo_t;
```

Algorithms to calculate PageRank.

Values:

IGRAPH_PAGERANK_ALGO_ARPACK:	Use the ARPACK library, this was the PageRank implementation in igraph from version 0.5, until version 0.7.
IGRAPH_PAGERANK_ALGO_PRPACK:	Use the PRPACK library. Currently this implementation is recommended.

igraph_pagerank — Calculates the Google PageRank for the specified vertices.

```
igraph_error_t igraph_pagerank(
        const igraph_t *graph, const igraph_vector_t *weights,
        igraph_vector_t *vector, igraph_real_t *value,
        igraph_real_t damping, igraph_bool_t directed,
        igraph_vs_t vids,
        igraph_pagerank_algo_t algo,
        igraph_arpack_options_t *options);
```

The PageRank centrality of a vertex is the fraction of time a random walker traversing the graph would spend on that vertex. The walker follows the out-edges with probabilities proportional to their weights. Additionally, in each step, it restarts the walk from a random vertex with probability 1 - damping. If the random walker gets stuck in a sink vertex, it will also restart from a random vertex.

The PageRank centrality is mainly useful for directed graphs. In undirected graphs it converges to trivial values proportional to degrees as the damping factor approaches 1.

Starting from version 0.9, igraph has two PageRank implementations, and the user can choose between them. The first implementation is IGRAPH_PAGERANK_ALGO_ARPACK, which phrases the PageRank calculation as an eigenvalue problem, which is then solved using the ARPACK library. This was the default before igraph version 0.7. The second and recommended implementation is

IGRAPH_PAGERANK_ALGO_PRPACK. This is using the PRPACK package, see https://github.com/dgleich/prpack. PRPACK uses an algebraic method, i.e. solves a linear system to obtain the PageRank scores.

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.

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. https://doi.org/10.1016/S0169-7552(98)00110-X

Arguments:

graph: The graph object.

weights: Optional edge weights. May be a NULL pointer, meaning unweighted edges, or a vector of non-negative values of the same length as the number of edges.

vector: Pointer to an initialized vector, the result is stored here. It is resized as needed.

value: Pointer to a real variable. When using IGRAPH_PAGERANK_ALGO_ARPACK, the eigenvalue corresponding to the PageRank vector is stored here. It is expected to be exactly one. Checking this value can be used to diagnose cases when ARPACK failed to converge to the leading eigenvector. When using IGRAPH_PAGERANK_ALGO_PRPACK, this is always set to 1.0.

damping: The damping factor ("d" in the original paper). Must be a probability in the range [0, 1]. A commonly used value is 0.85.

directed: Boolean, whether to consider the directedness of the edges. This is ignored for undirected graphs.

vids: The vertex IDs for which the PageRank is returned. This parameter is only for convenience. Computing PageRank for fewer than all vertices will not speed up the calculation.

algo: The PageRank implementation to use. Possible values: IGRAPH_PAGERANK_ALGO_ARPACK, IGRAPH_PAGERANK_ALGO_PRPACK.

options: Options for the ARPACK method. See igraph_arpack_options_t for details. Supply NULL here to use the defaults. Note that the function overwrites the n (number of vertices), nev (1), ncv (3) and which (LM) parameters and it always starts the calculation from a non-random vector calculated based on the degree of the vertices.

Returns:

Error code: IGRAPH_ENOMEM, not enough memory for temporary data. IGRAPH_EINVVID, invalid vertex ID in *vids*.

Time complexity: depends on the input graph, usually it is O(|E|), the number of edges.

See also:

igraph_personalized_pagerank() and igraph_personalized_pagerank_vs() for the personalized PageRank measure. See igraph_arpack_rssolve() and igraph_arpack_rnsolve() for the underlying machinery used by IGRAPH_PAGERANK_ALGO_ARPACK.

Example 17.17. File `examples/simple/igraph_pagerank.c`

`igraph_personalized_pagerank` — Calculates the personalized Google PageRank for the specified vertices.

```
igraph_error_t igraph_personalized_pagerank(
        const igraph_t *graph, const igraph_vector_t *weights,
        igraph_vector_t *vector, igraph_real_t *value,
        const igraph_vector_t *reset,
        igraph_real_t damping, igraph_bool_t directed,
        igraph_vs_t vids,
        igraph_pagerank_algo_t algo,
        igraph_arpack_options_t *options);
```

The personalized PageRank is similar to the original PageRank measure, but when the random walk is restarted, a new starting vertex is chosen non-uniformly, according to the distribution specified in *`reset`* (instead of the uniform distribution in the original PageRank measure). The *`reset`* distribution is used both when restarting randomly with probability `1 - damping`, and when the walker is forced to restart due to being stuck in a sink vertex (a vertex with no outgoing edges).

Note that the personalized PageRank of a given vertex depends on the personalized PageRank of all other vertices, so even if you want to calculate the personalized PageRank for only some of the vertices, all of them must be calculated. Requesting the personalized PageRank for only some of the vertices does not result in any performance increase at all.

Arguments:

`graph`: The graph object.

`weights`: Optional edge weights. May be a `NULL` pointer, meaning unweighted edges, or a vector of non-negative values of the same length as the number of edges.

`vector`: Pointer to an initialized vector, the result is stored here. It is resized as needed.

`value`: Pointer to a real variable. When using `IGRAPH_PAGERANK_ALGO_ARPACK`, the eigenvalue corresponding to the PageRank vector is stored here. It is expected to be exactly one. Checking this value can be used to diagnose cases when ARPACK failed to converge to the leading eigenvector. When using `IGRAPH_PAGERANK_ALGO_PRPACK`, this is always set to 1.0.

`reset`: The probability distribution over the vertices used when resetting the random walk. It is either a `NULL` pointer (denoting a uniform choice that results in the original PageRank measure) or a vector of the same length as the number of vertices.

`damping`: The damping factor ("d" in the original paper). Must be a probability in the range [0, 1]. A commonly used value is 0.85.

`directed`: Boolean, whether to consider the directedness of the edges. This is ignored for undirected graphs.

`vids`: The vertex IDs for which the PageRank is returned. This parameter is only for convenience. Computing PageRank for fewer than all vertices will not speed up the calculation.

`algo`: The PageRank implementation to use. Possible values: `IGRAPH_PAGERANK_ALGO_ARPACK`, `IGRAPH_PAGERANK_ALGO_PRPACK`.

options: Options for the ARPACK method. See `igraph_arpack_options_t` for details. Supply `NULL` here to use the defaults. Note that the function overwrites the `n` (number of vertices), `nev` (1), `ncv` (3) and `which` (LM) parameters and it always starts the calculation from a non-random vector calculated based on the degree of the vertices.

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory for temporary data. `IGRAPH_EINVVID`, invalid vertex ID in *`vids`* or an invalid reset vector in *`reset`*.

Time complexity: depends on the input graph, usually it is O(|E|), the number of edges.

See also:

`igraph_pagerank()` for the non-personalized implementation, `igraph_personalized_pagerank_vs()` for a personalized implementation with resetting to specific vertices.

`igraph_personalized_pagerank_vs` — Calculates the personalized Google PageRank for the specified vertices.

```
igraph_error_t igraph_personalized_pagerank_vs(
        const igraph_t *graph, const igraph_vector_t *weights,
        igraph_vector_t *vector, igraph_real_t *value,
        igraph_vs_t reset_vids,
        igraph_real_t damping, igraph_bool_t directed,
        igraph_vs_t vids,
        igraph_pagerank_algo_t algo,
        igraph_arpack_options_t *options);
```

The personalized PageRank is similar to the original PageRank measure, but when the random walk is restarted, a new starting vertex is chosen according to a specified distribution. This distribution is used both when restarting randomly with probability `1 - damping`, and when the walker is forced to restart due to being stuck in a sink vertex (a vertex with no outgoing edges).

This simplified interface takes a vertex sequence and resets the random walk to one of the vertices in the specified vertex sequence, chosen uniformly. A typical application of personalized PageRank is when the random walk is reset to the same vertex every time: this can easily be achieved using `igraph_vss_1()` which generates a vertex sequence containing only a single vertex.

Note that the personalized PageRank of a given vertex depends on the personalized PageRank of all other vertices, so even if you want to calculate the personalized PageRank for only some of the vertices, all of them must be calculated. Requesting the personalized PageRank for only some of the vertices does not result in any performance increase at all.

Arguments:

graph: The graph object.

weights: Optional edge weights, it is either a null pointer, then the edges are not weighted, or a vector of the same length as the number of edges.

vector: Pointer to an initialized vector, the result is stored here. It is resized as needed.

value: Pointer to a real variable. When using `IGRAPH_PAGERANK_ALGO_ARPACK`, the eigenvalue corresponding to the PageRank vector is stored here. It is expected to

be exactly one. Checking this value can be used to diagnose cases when ARPACK failed to converge to the leading eigenvector. When using `IGRAPH_PAGERANK_ALGO_PRPACK`, this is always set to 1.0.

`reset_vids`: IDs of the vertices used when resetting the random walk. The walk will be restarted from a vertex in this set, chosen uniformly at random. Duplicate vertices are allowed.

`damping`: The damping factor ("d" in the original paper). Must be a probability in the range [0, 1]. A commonly used value is 0.85.

`directed`: Boolean, whether to consider the directedness of the edges. This is ignored for undirected graphs.

`vids`: The vertex IDs for which the PageRank is returned. This parameter is only for convenience. Computing PageRank for fewer than all vertices will not speed up the calculation.

`algo`: The PageRank implementation to use. Possible values: `IGRAPH_PAGERANK_ALGO_ARPACK`, `IGRAPH_PAGERANK_ALGO_PRPACK`.

`options`: Options for the ARPACK method. See `igraph_arpack_options_t` for details. Supply `NULL` here to use the defaults. Note that the function overwrites the `n` (number of vertices), `nev` (1), `ncv` (3) and `which` (LM) parameters and it always starts the calculation from a non-random vector calculated based on the degree of the vertices.

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory for temporary data. `IGRAPH_EINVVID`, invalid vertex ID in *`vids`* or an empty reset vertex sequence in *`vids_reset`*.

Time complexity: depends on the input graph, usually it is O(|E|), the number of edges.

See also:

`igraph_pagerank()` for the non-personalized implementation.

`igraph_constraint` — Burt's constraint scores.

```
igraph_error_t igraph_constraint(const igraph_t *graph, igraph_vector_t *res,
                      igraph_vs_t vids, const igraph_vector_t *weights);
```

This function calculates Burt's constraint scores for the given vertices, also known as structural holes.

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(sum((p[i,q] p[q,j])^2, q in V[i], q != i,j), j in V[], j != i)

for a graph of order (i.e. number of vertices) N, where proportional tie strengths are defined as

p[i,j]=(a[i,j]+a[j,i]) / sum(a[i,k]+a[k,i], k in V[i], k != i),

a[i,j] are elements of A and the latter being the graph adjacency matrix. For isolated vertices, constraint is undefined.

Burt, R.S. (2004). Structural holes and good ideas. American Journal of Sociology 110, 349-399.

The first R version of this function was contributed by Jeroen Bruggeman.

Arguments:

graph: A graph object.

res: Pointer to an initialized vector, the result will be stored here. The vector will be resized to have the appropriate size for holding the result.

vids: Vertex selector containing the vertices for which the constraint should be calculated.

weights: Vector giving the weights of the edges. If it is NULL then each edge is supposed to have the same weight.

Returns:

Error code.

Time complexity: O(|V|+E|+n*d^2), n is the number of vertices for which the constraint is calculated and d is the average degree, |V| is the number of vertices, |E| the number of edges in the graph. If the weights argument is NULL then the time complexity is O(|V|+n*d^2).

igraph_maxdegree — The maximum degree in a graph (or set of vertices).

```
igraph_error_t igraph_maxdegree(
    const igraph_t *graph, igraph_int_t *res, igraph_vs_t vids,
    igraph_neimode_t mode, igraph_loops_t loops
);
```

The largest in-, out- or total degree of the specified vertices is calculated. If the graph has no vertices, or *vids* is empty, 0 is returned, as this is the smallest possible value for degrees.

Arguments:

graph: The input graph.

res: Pointer to an integer (igraph_int_t), the result will be stored here.

vids: Vector giving the vertex IDs for which the maximum degree will be calculated.

mode: Defines the type of the degree. IGRAPH_OUT, out-degree, IGRAPH_IN, in-degree, IGRAPH_ALL, total degree (sum of the in- and out-degree). This parameter is ignored for undirected graphs.

loops: Specifies how to treat loop edges when calculating the degree. IGRAPH_NO_LOOPS ignores loop edges; IGRAPH_LOOPS_ONCE counts each loop edge only once; IGRAPH_LOOPS_TWICE counts each loop edge twice in undirected graphs and once in directed graphs.

Returns:

Error code: IGRAPH_EINVVID: invalid vertex ID. IGRAPH_EINVMODE: invalid mode argument.

Time complexity: O(v) if *loops* is `true`, and O(v*d) otherwise. v is the number of vertices for which the degree will be calculated, and d is their (average) degree.

See also:

`igraph_degree()` to retrieve the degrees for several vertices.

`igraph_strength` — Strength of the vertices, also called weighted vertex degree.

```
igraph_error_t igraph_strength(
    const igraph_t *graph, igraph_vector_t *res, const igraph_vs_t vids,
    igraph_neimode_t mode, igraph_loops_t loops, const igraph_vector_t *weights
);
```

In a weighted network the strength of a vertex is the sum of the weights of all incident edges. In a non-weighted network this is exactly the vertex degree.

Arguments:

graph: The input graph.

res: Pointer to an initialized vector, the result is stored here. It will be resized as needed.

vids: The vertices for which the calculation is performed.

mode: Gives whether to count only outgoing (`IGRAPH_OUT`), incoming (`IGRAPH_IN`) edges or both (`IGRAPH_ALL`). This parameter is ignored for undirected graphs.

loops: Constant of type `igraph_loops_t`. Specifies how to treat loop edges when calculating the strength. `IGRAPH_NO_LOOPS` ignores loop edges; `IGRAPH_LOOPS_ONCE` counts each loop edge only once; `IGRAPH_LOOPS_TWICE` counts each loop edge twice in undirected graphs and once in directed graphs.

weights: A vector giving the edge weights. If this is a `NULL` pointer, then `igraph_degree()` is called to perform the calculation.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number vertices and edges.

See also:

`igraph_degree()` for the traditional, non-weighted version.

`igraph_eigenvector_centrality` — Eigenvector centrality of the vertices.

```
igraph_error_t igraph_eigenvector_centrality(const igraph_t *graph,
                                     igraph_vector_t *vector,
                                     igraph_real_t *value,
                                     igraph_neimode_t mode,
                                     const igraph_vector_t *weights,
```

```
                                        igraph_arpack_options_t *options);
```

Eigenvector centrality is a measure of the importance of a node in a network. It assigns relative scores to all nodes in the network based on the principle that connections from high-scoring nodes contribute more to the score of the node in question than equal connections from low-scoring nodes. Specifically, the eigenvector centrality of each vertex is proportional to the sum of eigenvector centralities of its neighbors. In practice, the centralities are determined by calculating the eigenvector corresponding to the largest positive eigenvalue of the adjacency matrix. This is motivated by the fact that the principal eigenvector is guaranteed to be non-negative, assuming that edge weights are also non-negative. In fact, in connected undirected graphs, this is the *only* non-negative eigenvector.

In the undirected case, this function considers the diagonal entries of the adjacency matrix to be *twice* the number of self-loops on the corresponding vertex.

In the weighted case, the eigenvector centrality of a vertex is proportional to the weighted sum of centralities of its neighbours, i.e. `c_j = sum_i w_ij c_i`, where `w_ij` is the weight of the edge connecting vertex `i` to `j`. The weights of parallel edges are added up.

The centrality scores returned by igraph are normalized such that the largest eigenvector centrality score is 1, unless all scores are zeros.

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 directly verify that the graph is connected. If it is not, in the undirected case the scores of all but one component will typically be zeros. When zeros are detected, a warning is issued.

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 meaningful for these graphs. igraph will return an eigenvalue of zero in such cases. The returned eigenvector centralities will all be equal for vertices with zero out-degree or zero in-degrees (depending on whether *`mode`* is `IGRAPH_OUT` or `IGRAPH_IN`) and zeros for other vertices. Such pathological cases can be detected by asking igraph to calculate the eigenvalue as well (using the *`value`* parameter, see below) and checking whether the eigenvalue is very close to zero.

Eigenvector centrality was developed for networks with non-negative edge weights. While igraph does not refuse to carry out the calculation with negative weights, it will issue a warning.

When working with directed graphs, consider using hub and authority scores instead, see `igraph_hub_and_authority_scores()`.

Arguments:

`graph`: The input graph. It may be directed.

`vector`: Pointer to an initialized vector, it will be resized as needed. The result of the computation is stored here. It can be a null pointer, then it is ignored.

`value`: If not a null pointer, then the eigenvalue corresponding to the found eigenvector is stored here.

`mode`: How to consider edge directions in directed graphs. It is ignored for undirected graphs. Possible values:

- `IGRAPH_OUT` the left eigenvector of the adjacency matrix is calculated, i.e. the centrality of a vertex is proportional to the sum of centralities of vertices pointing to it. This is the standard eigenvector centrality.
- `IGRAPH_IN` the right eigenvector of the adjacency matrix is calculated, i.e. the centrality of a vertex is proportional to the sum of centralities of vertices it points to.

`IGRAPH_ALL` edge directions are ignored, and the unweighted eigenvector centrality is calculated.

`weights`: A null pointer (indicating no edge weights), or a vector giving the weights of the edges. Weights should be positive to guarantee a meaningful result. The algorithm might produce complex numbers when some weights are negative and the graph is directed. In this case only the real part is reported.

`options`: Options to ARPACK. See `igraph_arpack_options_t` for details. Supply `NULL` here to use the defaults. Note that the function overwrites the `n` (number of vertices) parameter and it always starts the calculation from a non-random vector calculated based on the degree of the vertices.

Returns:

Error code.

Time complexity: depends on the input graph, usually it is O(|V|+|E|).

See also:

`igraph_pagerank` and `igraph_personalized_pagerank` for modifications of eigenvector centrality. `igraph_hub_and_authority_scores()` for a similar pair of measures intended for directed graphs.

Example 17.18. File `examples/simple/eigenvector_centrality.c`

`igraph_hub_and_authority_scores` — Kleinberg's hub and authority scores (HITS).

```
igraph_error_t igraph_hub_and_authority_scores(const igraph_t *graph,
        igraph_vector_t *hub_vector, igraph_vector_t *authority_vector,
        igraph_real_t *value,
        const igraph_vector_t *weights, igraph_arpack_options_t *options);
```

Hub and authority scores are a generalization of the ideas behind eigenvector centrality to directed graphs. The authority score of a vertex is proportional to the sum of the hub scores of vertices that point to it. Conversely, the hub score of a vertex is proportional to the sum of authority scores of vertices that it points to. These concepts are also known under the name Hyperlink-Induced Topic Search (HITS).

The hub and authority scores of the vertices are defined as the principal eigenvectors of `A A^T` and `A^T A`, respectively, where `A` is the adjacency matrix of the graph and `A^T` is its transpose. The motivation for choosing the principal eigenvector is that it is guaranteed to be non-negative when edge weights are also non-negative.

If vectors `h` and `a` contain hub and authority scores, then the two scores are related by `h = A a` and `a = A^T h`. When the principal eigenvalue of `A A^T` is degenerate, there is no unique solution to the hub- and authority-score problem. igraph guarantees that the scores that are returned are matching, i.e. are related by these formulas, even in this situation.

Note that hub and authority scores are not well behaved in extremely sparse graphs where no single connected component dominates the undirected graphs corresponding to `A A^T` and `A^T A`. In these cases, there are many different non-negative eigenvectors, all reasonable solutions to the HITS equations. The symptom of such a situation is that a large fraction of the scores are zeros. igraph issues a warning when this is detected.

Results are scaled so that the largest hub and authority scores are both 1.

The concept of hub and authority scores were developed for *directed* graphs. In undirected graphs, both the hub and authority scores are equal to the eigenvector centrality, which can be computed using `igraph_eigenvector_centrality()`.

HITS scores were developed for networks with non-negative edge weights. While igraph does not refuse to carry out the calculation with negative weights, it will issue a warning.

See the following reference on the meaning of this score: 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). https://doi.org/10.1145/324133.324140 Also appears as IBM Research Report RJ 10076, May 1997.

Arguments:

graph:	The input graph. Can be directed and undirected.
hub_vector:	Pointer to an initialized vector, the hub scores are stored here. If a null pointer then it is ignored.
authority_vector:	Pointer to an initialized vector, the authority scores are stored here. If a null pointer then it is ignored.
value:	If not a null pointer then the eigenvalue corresponding to the calculated eigenvectors is stored here.
weights:	A null pointer (meaning no edge weights), or a vector giving the weights of the edges.
options:	Options to ARPACK. See `igraph_arpack_options_t` for details. Supply `NULL` here to use the defaults. Note that the function overwrites the `n` (number of vertices) parameter and it always starts the calculation from a vector calculated based on the degree of the vertices.

Returns:

Error code.

Time complexity: depends on the input graph, usually it is O(|V|), the number of vertices.

See also:

`igraph_pagerank()`, `igraph_personalized_pagerank()`; `igraph_eigenvector_centrality()` for a similar measure intended for undirected graphs.

`igraph_convergence_degree` — Calculates the convergence degree of each edge in a graph.

```
igraph_error_t igraph_convergence_degree(const igraph_t *graph, igraph_vector_t
                                  igraph_vector_t *ins, igraph_vector_t *outs);
```

Let us define the input set of an edge (i, j) as the set of vertices where the shortest paths passing through (i, j) originate, and similarly, let us defined the output set of an edge (i, j) as the set of vertices where the shortest paths passing through (i, j) terminate. The convergence degree of an edge is defined as the normalized value of the difference between the size of the input set and the output set, i.e. the difference of them divided by the sum of them. Convergence degrees are in the range (-1, 1); a positive

value indicates that the edge is *convergent* since the shortest paths passing through it originate from a larger set and terminate in a smaller set, while a negative value indicates that the edge is *divergent* since the paths originate from a small set and terminate in a larger set.

Note that the convergence degree as defined above does not make sense in undirected graphs as there is no distinction between the input and output set. Therefore, for undirected graphs, the input and output sets of an edge are determined by orienting the edge arbitrarily while keeping the remaining edges undirected, and then taking the absolute value of the convergence degree.

Arguments:

graph: The input graph, it can be either directed or undirected.

result: Pointer to an initialized vector; the convergence degrees of each edge will be stored here. May be NULL if we are not interested in the exact convergence degrees.

ins: Pointer to an initialized vector; the size of the input set of each edge will be stored here. May be NULL if we are not interested in the sizes of the input sets.

outs: Pointer to an initialized vector; the size of the output set of each edge will be stored here. May be NULL if we are not interested in the sizes of the output sets.

Returns:

Error code.

Time complexity: O(|V||E|), the number of vertices times the number of edges.

Range-limited centrality measures

igraph_closeness_cutoff — Range limited closeness centrality.

```
igraph_error_t igraph_closeness_cutoff(const igraph_t *graph, igraph_vector_t *
                            igraph_vector_int_t *reachable_count, igraph_bool_t
                            const igraph_vs_t vids, igraph_neimode_t mode,
                            const igraph_vector_t *weights,
                            igraph_bool_t normalized,
                            igraph_real_t cutoff);
```

This function computes a range-limited version of closeness centrality by considering only those shortest paths whose length is no greater then the given cutoff value.

Arguments:

graph: The graph object.

res: The result of the computation, a vector containing the range-limited closeness centrality scores for the given vertices.

reachable_count: If not NULL, this vector will contain the number of vertices reachable within the cutoff distance from each vertex for which the range-limited closeness is calculated (not including that vertex).

all_reachable: Pointer to a Boolean. If not NULL, it indicates if all vertices of the graph were reachable from each vertex in *vids* within the given cutoff distance.

vids: The vertices for which the range limited closeness centrality will be computed.

mode: The type of shortest paths to be used for the calculation in directed graphs. Possible values:

- `IGRAPH_OUT` the lengths of the outgoing paths are calculated.
- `IGRAPH_IN` the lengths of the incoming paths are calculated.
- `IGRAPH_ALL` the directed graph is considered as an undirected one for the computation.

weights: An optional vector containing edge weights for weighted closeness. No edge weight may be NaN. Supply a null pointer here for traditional, unweighted closeness.

normalized: If true, the inverse of the mean distance to vertices reachable within the cutoff is returned. If false, the inverse of the sum of distances is returned.

cutoff: The maximal length of paths that will be considered. If negative or `IGRAPH_UNLIMITED`, the exact closeness will be calculated (no upper limit on path lengths).

Returns:

Error code:

- `IGRAPH_ENOMEM` not enough memory for temporary data.
- `IGRAPH_EINVVID` invalid vertex ID passed.
- `IGRAPH_EINVMODE` invalid mode argument.

Time complexity: At most O(n|E|) for the unweighted case and O(n|E|log|V|+|V|) for the weighted case, where n is the number of vertices for which the calculation is done, |V| is the number of vertices and |E| is the number of edges in the graph. The timing decreases with smaller cutoffs in a way that depends on the graph structure.

See also:

`igraph_closeness()` to calculate the exact closeness centrality.

`igraph_harmonic_centrality_cutoff` — Range limited harmonic centrality.

```
igraph_error_t igraph_harmonic_centrality_cutoff(const igraph_t *graph, igraph_v
                                      const igraph_vs_t vids, igraph_neimode_t m
                                      const igraph_vector_t *weights,
                                      igraph_bool_t normalized,
                                      igraph_real_t cutoff);
```

This function computes the range limited version of harmonic centrality: only those shortest paths are considered whose length is not above the given cutoff. The inverse distance to vertices not reachable within the cutoff is considered to be zero.

Arguments:

graph: The graph object.

res: The result of the computation, a vector containing the range limited harmonic centrality scores for the given vertices.

vids: The vertices for which the harmonic centrality will be computed.

mode: The type of shortest paths to be used for the calculation in directed graphs. Possible values:

- `IGRAPH_OUT` the lengths of the outgoing paths are calculated.
- `IGRAPH_IN` the lengths of the incoming paths are calculated.
- `IGRAPH_ALL` the directed graph is considered as an undirected one for the computation.

weights: An optional vector containing edge weights for weighted harmonic centrality. No edge weight may be NaN. If `NULL`, all weights are considered to be one.

normalized: Boolean, whether to normalize the result. 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 maximal length of paths that will be considered. The inverse distance to vertices that are not reachable within the cutoff path length is considered to be zero. Supply a negative value to compute the exact harmonic centrality, without any upper limit on the length of paths.

Returns:

Error code:

- `IGRAPH_ENOMEM` not enough memory for temporary data.
- `IGRAPH_EINVVID` invalid vertex ID passed.
- `IGRAPH_EINVMODE` invalid mode argument.

Time complexity: At most O(n|E|) for the unweighted case and O(n|E|log|V|+|V|) for the weighted case, where n is the number of vertices for which the calculation is done, |V| is the number of vertices and |E| is the number of edges in the graph. The timing decreases with smaller cutoffs in a way that depends on the graph structure.

See also:

`igraph_harmonic_centrality()` to calculate the exact harmonic centrality. Other centrality types: `igraph_closeness()`, `igraph_betweenness()`.

`igraph_betweenness_cutoff` — Range-limited betweenness centrality.

```
igraph_error_t igraph_betweenness_cutoff(
        const igraph_t *graph, const igraph_vector_t *weights,
        igraph_vector_t *res,
        igraph_vs_t vids,
        igraph_bool_t directed, igraph_bool_t normalized,
```

```
        igraph_real_t cutoff);
```

This function computes a range-limited version of betweenness centrality by considering only those shortest paths whose length is no greater then the given cutoff value.

Arguments:

graph: The graph object.

weights: An optional vector containing edge weights for calculating weighted betweenness. No edge weight may be NaN. Supply a null pointer here for unweighted betweenness.

res: The result of the computation, a vector containing the range-limited betweenness scores for the specified vertices.

vids: The vertices for which the range-limited betweenness centrality scores will be returned. This paramerer is for convenience only and does not affect performance. Internally, the betewenness of all vertices is calculated.

directed: If true directed paths will be considered for directed graphs. It is ignored for undirected graphs.

normalized: Whether to normalize betweenness scores by the number of vertex pairs. In directed graphs, the number of ordered vertex pairs, in undirected graphs the number of unordered vertex pairs is used.

cutoff: The maximal length of paths that will be considered. If negative or `IGRAPH_UNLIMITED`, the exact betweenness will be calculated, and there will be no upper limit on path lengths.

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory for temporary data. `IGRAPH_EINVVID`, invalid vertex ID passed in *vids*.

Time complexity: O(|V||E|), |V| and |E| are the number of vertices and edges in the graph. Note that the time complexity is independent of the number of vertices for which the score is calculated.

See also:

`igraph_betweenness()` to calculate the exact betweenness and `igraph_edge_betweenness_cutoff()` to calculate the range-limited edge betweenness.

`igraph_edge_betweenness_cutoff` — Range-limited betweenness centrality of the edges.

```
igraph_error_t igraph_edge_betweenness_cutoff(
        const igraph_t *graph, const igraph_vector_t *weights,
        igraph_vector_t *res,
        igraph_es_t eids,
        igraph_bool_t directed, igraph_bool_t normalized,
        igraph_real_t cutoff);
```

This function computes a range-limited version of edge betweenness centrality by considering only those shortest paths whose length is no greater then the given cutoff value.

Arguments:

graph: The graph object.

weights: An optional weight vector for weighted betweenness. No edge weight may be NaN. Supply a null pointer here for unweighted betweenness.

res: The result of the computation, vector containing the betweenness scores for the edges.

eids: The edges for which the betweenness centrality will be returned. This parameter is for convenience only, and does not affect performance. Internally, the betweenness is be calculated for all edges.

directed: If true directed paths will be considered for directed graphs. It is ignored for undirected graphs.

normalized: Whether to normalize betweenness scores by the number of vertex pairs. In directed graphs, the number of ordered vertex pairs, in undirected graphs the number of unordered vertex pairs is used.

cutoff: The maximal length of paths that will be considered. If negative of `IGRAPH_UNLIMITED`, the exact betweenness will be calculated (no upper limit on path lengths).

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory for temporary data.

Time complexity: O(|V||E|), |V| and |E| are the number of vertices and edges in the graph.

See also:

`igraph_edge_betweenness()` to compute the exact edge betweenness and `igraph_betweenness_cutoff()` to compute the range-limited vertex betweenness.

Subset-limited centrality measures

`igraph_betweenness_subset` — Betweenness centrality for a subset of source and target vertices.

```
igraph_error_t igraph_betweenness_subset(
        const igraph_t *graph, const igraph_vector_t *weights,
        igraph_vector_t *res,
        igraph_vs_t sources, igraph_vs_t targets,
        igraph_vs_t vids,
        igraph_bool_t directed, igraph_bool_t normalized);
```

This function computes the subset-limited version of betweenness centrality by considering only those shortest paths that lie between vertices in a given source and target subset.

Arguments:

graph: The graph object.

weights: An optional vector containing edge weights for calculating weighted betweenness. No edge weight may be NaN. Supply a null pointer here for unweighted betweenness.

res: The result of the computation, a vector containing the betweenness score for the subset of vertices.

sources: A vertex selector for the sources of the shortest paths taken into consideration in the betweenness calculation.

targets: A vertex selector for the targets of the shortest paths taken into consideration in the betweenness calculation.

vids: The vertices for which the subset-limited betweenness centrality scores will be computed.

directed: If true directed paths will be considered for directed graphs. It is ignored for undirected graphs.

normalized: Whether to normalize betweenness scores. Normalization is currently unimplemented, and setting this to `true` raises an error.

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory for temporary data. `IGRAPH_EINVVID`, invalid vertex ID passed in *vids*, *sources* or *targets*

Time complexity: O(|S||E|), where |S| is the number of vertices in the subset and |E| is the number of edges in the graph.

See also:

`igraph_betweenness()` to calculate the exact vertex betweenness and `igraph_betweenness_cutoff()` to calculate the range-limited vertex betweenness.

`igraph_edge_betweenness_subset` — Edge betweenness centrality for a subset of source and target vertices.

```
igraph_error_t igraph_edge_betweenness_subset(
        const igraph_t *graph, const igraph_vector_t *weights,
        igraph_vector_t *res,
        igraph_vs_t sources, igraph_vs_t targets,
        igraph_es_t eids,
        igraph_bool_t directed, igraph_bool_t normalized);
```

This function computes the subset-limited version of edge betweenness centrality by considering only those shortest paths that lie between vertices in a given source and target subset.

Arguments:

graph: The graph object.

weights: An optional weight vector for weighted betweenness. No edge weight may be NaN. Supply a null pointer here for unweighted betweenness.

res: The result of the computation, vector containing the betweenness scores for the edges.

sources: A vertex selector for the sources of the shortest paths taken into consideration in the betweenness calculation.

targets: A vertex selector for the targets of the shortest paths taken into consideration in the betweenness calculation.

eids: The edges for which the subset-limited betweenness centrality scores will be returned. This parameter is for convenience only. Internally, the betweenness will be calculated for all edges.

directed: If true directed paths will be considered for directed graphs. It is ignored for undirected graphs.

normalized: Whether to normalize betweenness scores. Normalization is currently unimplemented, and setting this to `true` raises an error.

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory for temporary data. `IGRAPH_EINVVID`, invalid vertex ID passed in *sources* or *targets*

Time complexity: O(|S||E|), where |S| is the number of vertices in the subset and |E| is the number of edges in the graph.

See also:

`igraph_edge_betweenness()` to compute the exact edge betweenness and `igraph_edge_betweenness_cutoff()` to compute the range-limited edge betweenness.

Centralization

`igraph_centralization` — Calculate the centralization score from the node level scores.

```
igraph_real_t igraph_centralization(const igraph_vector_t *scores,
                                    igraph_real_t theoretical_max,
                                    igraph_bool_t normalized);
```

For a centrality score defined on the vertices of a graph, it is possible to define a graph level centralization index, by calculating the sum of the deviations from the maximum centrality score. Consequently, the higher the centralization index of the graph, the more centralized the structure is.

In order to make graphs of different sizes comparable, the centralization index is usually normalized to a number between zero and one, by dividing the (unnormalized) centralization score of the most centralized structure with the same number of vertices.

For most centrality indices, the most centralized structure is the star graph, a single center connected to all other nodes in the network. There is some variation depending on whether the graph is directed or not, whether loop edges are allowed, etc.

This function simply calculates the graph level index, if the node level scores and the theoretical maximum are given. It is called by all the measure-specific centralization functions. It uses the calculation

```
C = sum_v ((max_u c_u) - c_v)
```

where `c` are the centrality scores passed in *scores*. If *normalized* is `true`, then `C/theoretical_max` is returned.

Arguments:

scores: A vector containing the node-level centrality scores.

theoretical_max: The graph level centrality score of the most centralized graph with the same number of vertices. Only used if `normalized` set to true.

normalized: Boolean, whether to normalize the centralization by dividing the supplied theoretical maximum.

Returns:

The graph level index.

See also:

`igraph_centralization_degree()`, `igraph_centralization_betweenness()`, `igraph_centralization_closeness()`, and `igraph_centralization_eigenvector_centrality()` for specific centralization functions.

Time complexity: O(n), the length of the score vector.

Example 17.19. File `examples/simple/centralization.c`

`igraph_centralization_degree` — Calculate vertex degree and graph centralization.

```
igraph_error_t igraph_centralization_degree(
    const igraph_t *graph, igraph_vector_t *res, igraph_neimode_t mode,
    igraph_loops_t loops, igraph_real_t *centralization,
    igraph_real_t *theoretical_max, igraph_bool_t normalized
);
```

This function calculates the degree of the vertices by passing its arguments to `igraph_degree()`; and it calculates the graph level centralization index based on the results by calling `igraph_centralization()`.

Arguments:

graph: The input graph.

res: A vector if you need the node-level degree scores, or a null pointer otherwise.

mode: Constant the specifies the type of degree for directed graphs. Possible values: `IGRAPH_IN`, `IGRAPH_OUT` and `IGRAPH_ALL`. This argument is ignored for undirected graphs.

loops: Specifies how to treat loop edges when calculating the degree (and the centralization). `IGRAPH_NO_LOOPS` ignores loop edges; `IGRAPH_LOOPS_ONCE` counts each loop edge only once; `IGRAPH_LOOPS_TWICE` counts each loop edge twice in undirected graphs and once in directed graphs.

centralization: Pointer to a real number, the centralization score is placed here.

theoretical_max: Pointer to real number or a null pointer. If not a null pointer, then the theoretical maximum graph centrality score for a graph with the same number vertices is stored here.

normalized: Boolean, whether to calculate a normalized centralization score. See `igraph_centralization()` for how the normalization is done.

Returns:

Error code.

See also:

`igraph_centralization()`, `igraph_degree()`.

Time complexity: the complexity of `igraph_degree()` plus O(n), the number of vertices queried, for calculating the centralization score.

`igraph_centralization_betweenness` — Calculate vertex betweenness and graph centralization.

```
igraph_error_t igraph_centralization_betweenness(const igraph_t *graph,
                                                 igraph_vector_t *res,
                                                 igraph_bool_t directed,
                                                 igraph_real_t *centralization,
                                                 igraph_real_t *theoretical_max,
                                                 igraph_bool_t normalized);
```

This function calculates the betweenness centrality of the vertices by passing its arguments to `igraph_betweenness()`; and it calculates the graph level centralization index based on the results by calling `igraph_centralization()`.

Arguments:

graph: The input graph.

res: A vector if you need the node-level betweenness scores, or a null pointer otherwise.

directed: Boolean, whether to consider directed paths when calculating betweenness.

centralization: Pointer to a real number, the centralization score is placed here.

theoretical_max: Pointer to real number or a null pointer. If not a null pointer, then the theoretical maximum graph centrality score for a graph with the same number vertices is stored here.

normalized: Boolean, whether to calculate a normalized centralization score. See `igraph_centralization()` for how the normalization is done.

Returns:

Error code.

See also:

`igraph_centralization()`, `igraph_betweenness()`.

Time complexity: the complexity of `igraph_betweenness()` plus O(n), the number of vertices queried, for calculating the centralization score.

igraph_centralization_closeness — Calculate vertex closeness and graph centralization.

```
igraph_error_t igraph_centralization_closeness(const igraph_t *graph,
                                               igraph_vector_t *res,
                                               igraph_neimode_t mode,
                                               igraph_real_t *centralization,
                                               igraph_real_t *theoretical_max,
                                               igraph_bool_t normalized);
```

This function calculates the closeness centrality of the vertices by passing its arguments to `igraph_closeness()`; and it calculates the graph level centralization index based on the results by calling `igraph_centralization()`.

Arguments:

graph: The input graph.

res: A vector if you need the node-level closeness scores, or a null pointer otherwise.

mode: Constant the specifies the type of closeness for directed graphs. Possible values: `IGRAPH_IN`, `IGRAPH_OUT` and `IGRAPH_ALL`. This argument is ignored for undirected graphs. See `igraph_closeness()` argument with the same name for more.

centralization: Pointer to a real number, the centralization score is placed here.

theoretical_max: Pointer to real number or a null pointer. If not a null pointer, then the theoretical maximum graph centrality score for a graph with the same number vertices is stored here.

normalized: Boolean, whether to calculate a normalized centralization score. See `igraph_centralization()` for how the normalization is done.

Returns:

Error code.

See also:

`igraph_centralization()`, `igraph_closeness()`.

Time complexity: the complexity of `igraph_closeness()` plus O(n), the number of vertices queried, for calculating the centralization score.

igraph_centralization_eigenvector_centrality — Calculate eigenvector centrality scores and graph centralization.

```
igraph_error_t igraph_centralization_eigenvector_centrality(
    const igraph_t *graph,
    igraph_vector_t *vector,
    igraph_real_t *value,
```

```
    igraph_neimode_t mode,
    igraph_arpack_options_t *options,
    igraph_real_t *centralization,
    igraph_real_t *theoretical_max,
    igraph_bool_t normalized);
```

This function calculates the eigenvector centrality of the vertices by passing its arguments to `igraph_eigenvector_centrality()`; and it calculates the graph level centralization index based on the results by calling `igraph_centralization()`.

Note that vertex-level eigenvector centrality scores do not have a natural scale. As with any eigenvector, their interpretation is invariant to scaling by a constant factor. However, due to how graph-level *centralization* is defined, its value depends on the specific scale/normalization used for vertex-level scores. Which of two graphs will have a higher eigenvector *centralization* depends on the choice of normalization for centralities. This function makes the specific choice of scaling vertex-level centrality scores by their maximum (i.e. it uses the $_{\infty}$-norm). Other normalization choices, such as the 1-norm or 2-norm are not currently implemented.

Arguments:

graph: The input graph.

vector: A vector if you need the node-level eigenvector centrality scores, or a null pointer otherwise.

value: If not a null pointer, then the leading eigenvalue is stored here.

mode: How to consider edge directions in directed graphs. See `igraph_eigenvector_centrality()` for details. Ignored for directed graphs.

options: Options to ARPACK. See `igraph_arpack_options_t` for details. Note that the function overwrites the `n` (number of vertices) parameter and it always starts the calculation from a non-random vector calculated based on the degree of the vertices.

centralization: Pointer to a real number, the centralization score is placed here.

theoretical_max: Pointer to real number or a null pointer. If not a null pointer, then the theoretical maximum graph centrality score for a graph with the same number vertices is stored here.

normalized: Boolean, whether to calculate a normalized centralization score. See `igraph_centralization()` for how the normalization is done.

Returns:

Error code.

See also:

`igraph_centralization()`, `igraph_eigenvector_centrality()`.

Time complexity: the complexity of `igraph_eigenvector_centrality()` plus O(|V|), the number of vertices for the calculating the centralization.

`igraph_centralization_degree_tmax` — Theoretical maximum for graph centralization based on degree.

```
igraph_error_t igraph_centralization_degree_tmax(
    const igraph_t *graph, igraph_int_t nodes, igraph_neimode_t mode,
    igraph_loops_t loops, igraph_real_t *res
);
```

This function returns the theoretical maximum graph centrality based on vertex degree.

There are two ways to call this function, the first is to supply a graph as the *graph* argument, and then the number of vertices is taken from this object, and its directedness is considered as well. The *nodes* argument is ignored in this case. The *mode* argument is also ignored if the supplied graph is undirected.

The other way is to supply a null pointer as the *graph* argument. In this case the *nodes* and *mode* arguments are considered.

The most centralized structure is the star. More specifically, for undirected graphs it is the star, for directed graphs it is the in-star or the out-star.

Arguments:

graph: A graph object or a null pointer, see the description above.

nodes: The number of nodes. This is ignored if the *graph* argument is not a null pointer.

mode: Constant, whether the calculation is based on in-degree (IGRAPH_IN), out-degree (IGRAPH_OUT) or total degree (IGRAPH_ALL). This is ignored if the *graph* argument is not a null pointer and the given graph is undirected.

loops: Specifies how to treat loop edges when calculating the degree (and the centralization). IGRAPH_NO_LOOPS ignores loop edges; IGRAPH_LOOPS_ONCE counts each loop edge only once; IGRAPH_LOOPS_TWICE counts each loop edge twice in undirected graphs and once in directed graphs.

res: Pointer to a real variable, the result is stored here.

Returns:

Error code.

Time complexity: O(1).

See also:

igraph_centralization_degree() and igraph_centralization().

igraph_centralization_betweenness_tmax — Theoretical maximum for graph centralization based on betweenness.

```
igraph_error_t igraph_centralization_betweenness_tmax(const igraph_t *graph,
        igraph_int_t nodes,
        igraph_bool_t directed,
        igraph_real_t *res);
```

This function returns the theoretical maximum graph centrality based on vertex betweenness.

There are two ways to call this function, the first is to supply a graph as the *graph* argument, and then the number of vertices is taken from this object, and its directedness is considered as well. The *nodes* argument is ignored in this case. The *directed* argument is also ignored if the supplied graph is undirected.

The other way is to supply a null pointer as the *graph* argument. In this case the *nodes* and *directed* arguments are considered.

The most centralized structure is the star.

Arguments:

graph: A graph object or a null pointer, see the description above.

nodes: The number of nodes. This is ignored if the *graph* argument is not a null pointer.

directed: Boolean, whether to use directed paths in the betweenness calculation. This argument is ignored if *graph* is not a null pointer and it is undirected.

res: Pointer to a real variable, the result is stored here.

Returns:

Error code.

Time complexity: O(1).

See also:

`igraph_centralization_betweenness()` and `igraph_centralization()`.

`igraph_centralization_closeness_tmax` — Theoretical maximum for graph centralization based on closeness.

```
igraph_error_t igraph_centralization_closeness_tmax(const igraph_t *graph,
        igraph_int_t nodes,
        igraph_neimode_t mode,
        igraph_real_t *res);
```

This function returns the theoretical maximum graph centrality based on vertex closeness.

There are two ways to call this function, the first is to supply a graph as the *graph* argument, and then the number of vertices is taken from this object, and its directedness is considered as well. The *nodes* argument is ignored in this case. The *mode* argument is also ignored if the supplied graph is undirected.

The other way is to supply a null pointer as the *graph* argument. In this case the *nodes* and *mode* arguments are considered.

The most centralized structure is the star.

Arguments:

graph: A graph object or a null pointer, see the description above.

nodes: The number of nodes. This is ignored if the *graph* argument is not a null pointer.

mode: Constant, specifies what kind of distances to consider to calculate closeness. See the *mode* argument of `igraph_closeness()` for details. This argument is ignored if *graph* is not a null pointer and it is undirected.

res: Pointer to a real variable, the result is stored here.

Returns:

Error code.

Time complexity: O(1).

See also:

`igraph_centralization_closeness()` and `igraph_centralization()`.

`igraph_centralization_eigenvector_centrality_tmax` — Theoretical maximum centralization for eigenvector centrality.

```
igraph_error_t igraph_centralization_eigenvector_centrality_tmax(
    const igraph_t *graph,
    igraph_int_t nodes,
    igraph_neimode_t mode,
    igraph_real_t *res);
```

This function returns the theoretical maximum graph centrality based on vertex eigenvector centrality.

There are two ways to call this function, the first is to supply a graph as the *graph* argument, and then the number of vertices is taken from this object, and its directedness is considered as well. The *nodes* argument is ignored in this case. The *mode* argument is also ignored if the supplied graph is undirected.

The other way is to supply a null pointer as the *graph*. argument. In this case the *nodes* and *mode* arguments are considered.

The most centralized directed structure is the in-star with *mode* set to `IGRAPH_OUT`, and the out-star with *mode* set to `IGRAPH_IN`. The most centralized undirected structure is the graph with a single edge.

Arguments:

graph: A graph object or a null pointer, see the description above.

nodes: The number of nodes. This is ignored if the *graph* argument is not a null pointer.

mode: How to consider edge directions in directed graphs. See `igraph_eigenvector_centrality()` for details. This argument is ignored if *graph* is not a null pointer and it is undirected.

res: Pointer to a real variable, the result is stored here.

Returns:

Error code.

Time complexity: O(1).

See also:

`igraph_centralization_closeness()` and `igraph_centralization()`.

Similarity measures

`igraph_bibcoupling` — Bibliographic coupling.

```
igraph_error_t igraph_bibcoupling(const igraph_t *graph, igraph_matrix_t *res,
                      const igraph_vs_t vids);
```

The bibliographic coupling of two vertices is the number of other vertices they both cite, `igraph_bibcoupling()` calculates this. The bibliographic coupling score for each given vertex and all other vertices in the graph will be calculated.

Arguments:

`graph`: The graph object to analyze.

`res`: Pointer to a matrix, the result of the calculation will be stored here. The number of its rows is the same as the number of vertex IDs in *`vids`*, the number of columns is the number of vertices in the graph.

`vids`: The vertex IDs of the vertices for which the calculation will be done.

Returns:

Error code: `IGRAPH_EINVVID`: invalid vertex ID.

Time complexity: O(|V|d^2), |V| is the number of vertices in the graph, d is the (maximum) degree of the vertices in the graph.

See also:

`igraph_cocitation()`

Example 17.20. File `examples/simple/igraph_cocitation.c`

`igraph_cocitation` — Cocitation coupling.

```
igraph_error_t igraph_cocitation(const igraph_t *graph, igraph_matrix_t *res,
                     const igraph_vs_t vids);
```

Two vertices are cocited if there is another vertex citing both of them. `igraph_cocitation()` simply counts how many times two vertices are cocited. The cocitation score for each given vertex and all other vertices in the graph will be calculated.

Arguments:

`graph`: The graph object to analyze.

`res`: Pointer to a matrix, the result of the calculation will be stored here. The number of its rows is the same as the number of vertex IDs in *`vids`*, the number of columns is the number of vertices in the graph.

vids: The vertex IDs of the vertices for which the calculation will be done.

Returns:

Error code: IGRAPH_EINVVID: invalid vertex ID.

Time complexity: O(|V|d^2), |V| is the number of vertices in the graph, d is the (maximum) degree of the vertices in the graph.

See also:

igraph_bibcoupling()

Example 17.21. File examples/simple/igraph_cocitation.c

igraph_similarity_jaccard — Jaccard similarity coefficient for the given vertices.

```
igraph_error_t igraph_similarity_jaccard(const igraph_t *graph, igraph_matrix_t
                                const igraph_vs_t from, const igraph_vs_t to, igra
```

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. This function calculates the pairwise Jaccard similarities for some (or all) of the vertices.

Arguments:

graph: The graph object to analyze

res: Pointer to a matrix, the result of the calculation will be stored here. The number of its rows and columns is the same as the number of vertex IDs in *vit_from* and *vit_to*, respectively.

from: The vertex IDs of the first set of vertices of the pairs for which the calculation will be done.

to: The vertex IDs of the second set of vertices of the pairs for which the calculation will be done.

mode: The type of neighbors to be used for the calculation in directed graphs. Possible values:

- IGRAPH_OUT — the outgoing edges will be considered for each node.
- IGRAPH_IN — the incoming edges will be considered for each node.
- IGRAPH_ALL — the directed graph is considered as an undirected one for the computation.

loops: Whether to include the vertices themselves in the neighbor sets.

Returns:

Error code:

- IGRAPH_ENOMEM — not enough memory for temporary data.
- IGRAPH_EINVVID — invalid vertex ID passed.
- IGRAPH_EINVMODE — invalid mode argument.

Time complexity: O(|V|^2 d), |V| is the number of vertices in the vertex iterator given, d is the (maximum) degree of the vertices in the graph.

See also:

`igraph_similarity_dice()`, a measure very similar to the Jaccard coefficient

Example 17.22. File `examples/simple/igraph_similarity.c`

`igraph_similarity_jaccard_pairs` — Jaccard similarity coefficient for given vertex pairs.

```
igraph_error_t igraph_similarity_jaccard_pairs(const igraph_t *graph, igraph_ve
                                      const igraph_vector_int_t *pairs, igraph_ne
```

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. This function calculates the pairwise Jaccard similarities for a list of vertex pairs.

Arguments:

`graph`: The graph object to analyze

`res`: Pointer to a vector, the result of the calculation will be stored here. The number of elements is the same as the number of pairs in *`pairs`*.

`pairs`: A vector that contains the pairs for which the similarity will be calculated. Each pair is defined by two consecutive elements, i.e. the first and second element of the vector specifies the first pair, the third and fourth element specifies the second pair and so on.

`mode`: The type of neighbors to be used for the calculation in directed graphs. Possible values:

- `IGRAPH_OUT` the outgoing edges will be considered for each node.
- `IGRAPH_IN` the incoming edges will be considered for each node.
- `IGRAPH_ALL` the directed graph is considered as an undirected one for the computation.

`loops`: Whether to include the vertices themselves in the neighbor sets.

Returns:

Error code:

- `IGRAPH_ENOMEM` not enough memory for temporary data.
- `IGRAPH_EINVVID` invalid vertex ID passed.
- `IGRAPH_EINVMODE` invalid mode argument.

Time complexity: O(nd), n is the number of pairs in the given vector, d is the (maximum) degree of the vertices in the graph.

See also:

`igraph_similarity_jaccard()` to calculate the Jaccard similarity between all pairs of a vertex set, or `igraph_similarity_dice()` and `igraph_similarity_dice_pairs()` for a measure very similar to the Jaccard coefficient

Example 17.23. File `examples/simple/igraph_similarity.c`

`igraph_similarity_jaccard_es` — Jaccard similarity coefficient for a given edge selector.

```
igraph_error_t igraph_similarity_jaccard_es(const igraph_t *graph, igraph_vecto
                                    const igraph_es_t es, igraph_neimode_t mode, ig
```

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. This function calculates the pairwise Jaccard similarities for the endpoints of edges in a given edge selector.

Arguments:

`graph`: The graph object to analyze

`res`: Pointer to a vector, the result of the calculation will be stored here. The number of elements is the same as the number of edges in *`es`*.

`es`: An edge selector that specifies the edges to be included in the result.

`mode`: The type of neighbors to be used for the calculation in directed graphs. Possible values:

- `IGRAPH_OUT` the outgoing edges will be considered for each node.
- `IGRAPH_IN` the incoming edges will be considered for each node.
- `IGRAPH_ALL` the directed graph is considered as an undirected one for the computation.

`loops`: Whether to include the vertices themselves in the neighbor sets.

Returns:

Error code:

- `IGRAPH_ENOMEM` not enough memory for temporary data.
- `IGRAPH_EINVVID` invalid vertex ID passed.
- `IGRAPH_EINVMODE` invalid mode argument.

Time complexity: O(nd), n is the number of edges in the edge selector, d is the (maximum) degree of the vertices in the graph.

See also:

`igraph_similarity_jaccard()` and `igraph_similarity_jaccard_pairs()` to calculate the Jaccard similarity between all pairs of a vertex set or some selected vertex pairs, or `igraph_similarity_dice()`, `igraph_similarity_dice_pairs()` and `igraph_similarity_dice_es()` for a measure very similar to the Jaccard coefficient

Example 17.24. File `examples/simple/igraph_similarity.c`

`igraph_similarity_dice` — Dice similarity coefficient.

```
igraph_error_t igraph_similarity_dice(const igraph_t *graph, igraph_matrix_t *r
                                      const igraph_vs_t vit_from, const igraph_v
                                      igraph_neimode_t mode, igraph_bool_t loops
```

The Dice similarity coefficient of two vertices is twice the number of common neighbors divided by the sum of the degrees of the vertices. This function calculates the pairwise Dice similarities for some (or all) of the vertices.

Arguments:

graph: The graph object to analyze.

res: Pointer to a matrix, the result of the calculation will be stored here. The number of its rows and columns is the same as the number of vertex IDs in *vit_from* and *vit_to*, respectively.

vit_from: The vertex IDs of the first vertices of the pairs for which the calculation will be done.

vit_to: The vertex IDs of the second vertices of the pairs for which the calculation will be done.

mode: The type of neighbors to be used for the calculation in directed graphs. Possible values:

- IGRAPH_OUT — the outgoing edges will be considered for each node.
- IGRAPH_IN — the incoming edges will be considered for each node.
- IGRAPH_ALL — the directed graph is considered as an undirected one for the computation.

loops: Whether to include the vertices themselves as their own neighbors.

Returns:

Error code:

- IGRAPH_ENOMEM — not enough memory for temporary data.
- IGRAPH_EINVVID — invalid vertex ID passed.
- IGRAPH_EINVMODE — invalid mode argument.

Time complexity: O(|V|^2 d), where |V| is the number of vertices in the vertex iterator given, and d is the (maximum) degree of the vertices in the graph.

See also:

igraph_similarity_jaccard(), a measure very similar to the Dice coefficient

Example 17.25. File examples/simple/igraph_similarity.c

igraph_similarity_dice_pairs — Dice similarity coefficient for given vertex pairs.

```
igraph_error_t igraph_similarity_dice_pairs(const igraph_t *graph, igraph_vecto
                                    const igraph_vector_int_t *pairs, igraph_neimo
```

The Dice similarity coefficient of two vertices is twice the number of common neighbors divided by the sum of the degrees of the vertices. This function calculates the pairwise Dice similarities for a list of vertex pairs.

Arguments:

graph: The graph object to analyze

res: Pointer to a vector, the result of the calculation will be stored here. The number of elements is the same as the number of pairs in *pairs*.

pairs: A vector that contains the pairs for which the similarity will be calculated. Each pair is defined by two consecutive elements, i.e. the first and second element of the vector specifies the first pair, the third and fourth element specifies the second pair and so on.

mode: The type of neighbors to be used for the calculation in directed graphs. Possible values:

`IGRAPH_OUT` the outgoing edges will be considered for each node.

`IGRAPH_IN` the incoming edges will be considered for each node.

`IGRAPH_ALL` the directed graph is considered as an undirected one for the computation.

loops: Whether to include the vertices themselves as their own neighbors.

Returns:

Error code:

`IGRAPH_ENOMEM` not enough memory for temporary data.

`IGRAPH_EINVVID` invalid vertex ID passed.

`IGRAPH_EINVMODE` invalid mode argument.

Time complexity: O(nd), n is the number of pairs in the given vector, d is the (maximum) degree of the vertices in the graph.

See also:

`igraph_similarity_dice()` to calculate the Dice similarity between all pairs of a vertex set, or `igraph_similarity_jaccard()`, `igraph_similarity_jaccard_pairs()` and `igraph_similarity_jaccard_es()` for a measure very similar to the Dice coefficient

Example 17.26. File `examples/simple/igraph_similarity.c`

`igraph_similarity_dice_es` — Dice similarity coefficient for a given edge selector.

```
igraph_error_t igraph_similarity_dice_es(const igraph_t *graph, igraph_vector_t
                                const igraph_es_t es, igraph_neimode_t mode, igra
```

The Dice similarity coefficient of two vertices is twice the number of common neighbors divided by the sum of the degrees of the vertices. This function calculates the pairwise Dice similarities for the endpoints of edges in a given edge selector.

Arguments:

graph: The graph object to analyze

res: Pointer to a vector, the result of the calculation will be stored here. The number of elements is the same as the number of edges in *es*.

es: An edge selector that specifies the edges to be included in the result.

mode: The type of neighbors to be used for the calculation in directed graphs. Possible values:

- `IGRAPH_OUT` the outgoing edges will be considered for each node.
- `IGRAPH_IN` the incoming edges will be considered for each node.
- `IGRAPH_ALL` the directed graph is considered as an undirected one for the computation.

loops: Whether to include the vertices themselves as their own neighbors.

Returns:

Error code:

- `IGRAPH_ENOMEM` not enough memory for temporary data.
- `IGRAPH_EINVVID` invalid vertex ID passed.
- `IGRAPH_EINVMODE` invalid mode argument.

Time complexity: O(nd), n is the number of pairs in the given vector, d is the (maximum) degree of the vertices in the graph.

See also:

`igraph_similarity_dice()` and `igraph_similarity_dice_pairs()` to calculate the Dice similarity between all pairs of a vertex set or some selected vertex pairs, or `igraph_similarity_jaccard()`, `igraph_similarity_jaccard_pairs()` and `igraph_similarity_jaccard_es()` for a measure very similar to the Dice coefficient

Example 17.27. File `examples/simple/igraph_similarity.c`

`igraph_similarity_inverse_log_weighted` — Vertex similarity based on the inverse logarithm of vertex degrees.

```
igraph_error_t igraph_similarity_inverse_log_weighted(const igraph_t *graph,
        igraph_matrix_t *res, const igraph_vs_t vids, igraph_neimode_t mode);
```

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.

Note that the presence of loop edges may yield counter-intuitive results. A node with a loop edge is considered to be a neighbor of itself *twice* (because there are two edge stems incident on the node). Adding a loop edge to a node may decrease its similarity to other nodes, but it may also *increase* it.

For instance, if nodes A and B are connected but share no common neighbors, their similarity is zero. However, if a loop edge is added to B, then B itself becomes a common neighbor of A and B and thus the similarity of A and B will be increased. Consider removing loop edges explicitly before invoking this function using `igraph_simplify()`.

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. https://doi.org/10.1016/S0378-8733(03)00009-1

Arguments:

graph: The graph object to analyze.

res: Pointer to a matrix, the result of the calculation will be stored here. The number of its rows is the same as the number of vertex IDs in *vids*, the number of columns is the number of vertices in the graph.

vids: The vertex IDs of the vertices for which the calculation will be done.

mode: The type of neighbors to be used for the calculation in directed graphs. Possible values:

- `IGRAPH_OUT` the outgoing edges will be considered for each node. Nodes will be weighted according to their in-degree.
- `IGRAPH_IN` the incoming edges will be considered for each node. Nodes will be weighted according to their out-degree.
- `IGRAPH_ALL` the directed graph is considered as an undirected one for the computation. Every node is weighted according to its undirected degree.

Returns:

Error code: `IGRAPH_EINVVID`: invalid vertex ID.

Time complexity: O(|V|d^2), |V| is the number of vertices in the graph, d is the (maximum) degree of the vertices in the graph.

Example 17.28. File `examples/simple/igraph_similarity.c`

Trees and forests

`igraph_minimum_spanning_tree` — Calculates a minimum spanning tree of a graph.

```
igraph_error_t igraph_minimum_spanning_tree(
    const igraph_t *graph, igraph_vector_int_t *res,
    const igraph_vector_t *weights, igraph_mst_algorithm_t method);
```

Finds a minimum weight spanning tree of the graph. If the graph is not connected then its minimum spanning forest is returned, i.e. the set of the minimum spanning trees of each component.

Directed graphs are treated as undirected for this computation.

This function is deterministic, i.e. it always returns the same spanning tree. See `igraph_random_spanning_tree()` for the uniform random sampling of spanning trees of a graph.

References:

Prim, R.C.: Shortest connection networks and some generalizations, Bell System Technical Journal, Vol. 36, 1957, 1389--1401. https://doi.org/10.1002/j.1538-7305.1957.tb01515.x

Kruskal, J. B.: On the shortest spanning subtree of a graph and the traveling salesman problem, Proc. Amer. Math. Soc. 7 (1956), 48-50 https://doi.org/10.1090%2FS0002-9939-1956-0078686-7

Arguments:

graph: The graph object. Edge directions will be ignored.

res: An initialized vector, the IDs of the edges that constitute a spanning tree will be returned here. Use `igraph_subgraph_from_edges()` to extract the spanning tree as a separate graph object.

weights: A vector containing the weights of the edges in the order of edge IDs. Weights must not be NaN. Supply `NULL` to treat all edges as having the same weight.

method: The type of the algorithm used.

`IGRAPH_MST_AUTOMATIC`	tries to select the best performing algorithm for the current graph.
`IGRAPH_MST_UNWEIGHTED`	ignores edge weights and produces an arbitrary spanning tree.
`IGRAPH_MST_PRIM`	uses Prim's algorithm.
`IGRAPH_MST_KRUSKAL`	uses Kruskal's algorithm.

Returns:

Error code.

Time complexity: See the functions implementing the specific algorithms.

See also:

`igraph_random_spanning_tree()` to compute a random spanning tree instead of a minimum one.

Example 17.29. File `examples/simple/igraph_minimum_spanning_tree.c`

`igraph_random_spanning_tree` — Uniformly samples the spanning trees of a graph.

```
igraph_error_t igraph_random_spanning_tree(const igraph_t *graph, igraph_vector
```

Performs a loop-erased random walk on the graph to uniformly sample its spanning trees. Edge directions are ignored.

Multi-graphs are supported, and edge multiplicities will affect the sampling frequency. For example, consider the 3-cycle graph `1=2-3-1`, with two edges between vertices 1 and 2. Due to these parallel edges, the trees `1-2-3` and `3-1-2` will be sampled with multiplicity 2, while the tree `2-3-1` will be sampled with multiplicity 1.

Arguments:

graph: The input graph. Edge directions are ignored.

res: An initialized vector, the IDs of the edges that constitute a spanning tree will be returned here. Use `igraph_subgraph_from_edges()` to extract the spanning tree as a separate graph object.

vid: This parameter is relevant if the graph is not connected. If negative, a random spanning forest of all components will be generated. Otherwise, it should be the ID of a vertex. A random spanning tree of the component containing the vertex will be generated.

Returns:

Error code.

See also:

`igraph_minimum_spanning_tree()`, `igraph_random_walk()`

`igraph_is_tree` — Decides whether the graph is a tree.

```
igraph_error_t igraph_is_tree(const igraph_t *graph, igraph_bool_t *res, igraph
```

An undirected graph is a tree if it is connected and has no cycles.

In the directed case, an 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 connected, and therefore not a tree.

Arguments:

graph: The graph object to analyze.

res: Pointer to a Boolean variable, the result will be stored here.

root: If not `NULL`, the root node will be stored here. When *mode* is `IGRAPH_ALL` or the graph is undirected, any vertex can be the root and *root* is set to 0 (the first vertex). When *mode* is `IGRAPH_OUT` or `IGRAPH_IN`, the root is set to the vertex with zero in- or out-degree, respectively.

mode: For a directed graph this specifies whether to test for an out-tree, an in-tree or ignore edge directions. The respective possible values are: `IGRAPH_OUT`, `IGRAPH_IN`, `IGRAPH_ALL`. This argument is ignored for undirected graphs.

Returns:

Error code: `IGRAPH_EINVAL`: invalid mode argument.

Time complexity: At most O(|V|+|E|), the number of vertices plus the number of edges in the graph.

See also:

`igraph_is_forest()` to check if all components are trees, which is equivalent to the graph lacking undirected cycles; `igraph_is_connected()`, `igraph_is_acyclic()`

Example 17.30. File `examples/simple/igraph_kary_tree.c`

`igraph_is_forest` — Decides whether the graph is a forest.

```
igraph_error_t igraph_is_forest(const igraph_t *graph, igraph_bool_t *res,
                                igraph_vector_int_t *roots, igraph_neimode_t mo
```

An undirected graph is a forest if it has no cycles. Equivalently, a graph is a forest if all connected components are trees.

In the directed case, an 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.

The *`res`* return value of this function is cached in the graph itself if *`mode`* is set to `IGRAPH_ALL` or if the graph is undirected. Calling the function multiple times with no modifications to the graph in between will return a cached value in O(1) time if the roots are not requested.

Arguments:

`graph`: The graph object to analyze.

`res`: Pointer to a Boolean variable. If not `NULL`, then the result will be stored here.

`roots`: If not `NULL`, the root nodes will be stored here. When *`mode`* is `IGRAPH_ALL` or the graph is undirected, any one vertex from each component can be the root. When *`mode`* is `IGRAPH_OUT` or `IGRAPH_IN`, all the vertices with zero in- or out-degree, respectively are considered as root nodes.

`mode`: For a directed graph this specifies whether to test for an out-forest, an in-forest or ignore edge directions. The respective possible values are: `IGRAPH_OUT`, `IGRAPH_IN`, `IGRAPH_ALL`. This argument is ignored for undirected graphs.

Returns:

Error code: `IGRAPH_EINVMODE`: invalid mode argument.

Time complexity: At most O(|V|+|E|), the number of vertices plus the number of edges in the graph.

See also:

`igraph_is_tree()` to check if a graph is a tree, i.e. a forest with a single component; `igraph_is_acyclic()` to check if a graph lacks (undirected or directed) cycles.

`igraph_to_prufer` — Converts a tree to its Prüfer sequence.

```
igraph_error_t igraph_to_prufer(const igraph_t *graph, igraph_vector_int_t* pru
```

A Prüfer sequence is a unique sequence of integers associated with a labelled tree. A tree on n >= 2 vertices can be represented by a sequence of n-2 integers, each between 0 and n-1 (inclusive).

Arguments:

graph: Pointer to an initialized graph object which must be a tree on n >= 2 vertices.

prufer: A pointer to the integer vector that should hold the Prüfer sequence; the vector must be initialized and will be resized to n - 2.

Returns:

Error code:

`IGRAPH_ENOMEM` there is not enough memory to perform the operation.

`IGRAPH_EINVAL` the graph is not a tree or it is has less than vertices

See also:

`igraph_from_prufer()`

Transitivity or clustering coefficient

`igraph_transitivity_undirected` — Calculates the transitivity (clustering coefficient) of a graph.

```
igraph_error_t igraph_transitivity_undirected(const igraph_t *graph,
                                    igraph_real_t *res,
                                    igraph_transitivity_mode_t mode);
```

The transitivity measures the probability that two neighbors of a vertex are connected. More precisely, this is the ratio of the triangles and connected triples in the graph, the result is a single real number. Directed graphs are considered as undirected ones and multi-edges are ignored.

Note that this measure is different from the local transitivity measure (see `igraph_transitivity_local_undirected()`) as it calculates a single value for the whole graph.

Clustering coefficient is an alternative name for transitivity.

References:

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

Arguments:

graph: The graph object. Edge directions and multiplicites are ignored.

res: Pointer to a real variable, the result will be stored here.

mode: Defines how to treat graphs with no connected triples. `IGRAPH_TRANSITIVITY_NAN` returns `NaN` in this case, `IGRAPH_TRANSITIVITY_ZERO` returns zero.

Returns:

Error code: `IGRAPH_ENOMEM`: not enough memory for temporary data.

See also:

`igraph_transitivity_local_undirected()`, `igraph_transitivity_avglocal_undirected()`.

Time complexity: O(|V|*d^2), |V| is the number of vertices in the graph, d is the average node degree.

Example 17.31. File `examples/simple/igraph_transitivity.c`

`igraph_transitivity_local_undirected` — The local transitivity (clustering coefficient) of some vertices.

```
igraph_error_t igraph_transitivity_local_undirected(const igraph_t *graph,
        igraph_vector_t *res,
        const igraph_vs_t vids,
        igraph_transitivity_mode_t mode);
```

The transitivity measures the probability that two neighbors of a vertex are connected. In case of the local transitivity, this probability is calculated separately for each vertex.

Note that this measure is different from the global transitivity measure (see `igraph_transitivity_undirected()`) as it calculates a transitivity value for each vertex individually.

Clustering coefficient is an alternative name for transitivity.

References:

D. J. Watts and S. Strogatz: Collective dynamics of small-world networks. Nature 393(6684):440-442 (1998).

Arguments:

`graph`: The input graph. Edge directions and multiplicities are ignored.

`res`: Pointer to an initialized vector, the result will be stored here. It will be resized as needed.

`vids`: Vertex set, the vertices for which the local transitivity will be calculated.

`mode`: Defines how to treat vertices with degree less than two. `IGRAPH_TRANSITIVITY_NAN` returns `NaN` for these vertices, `IGRAPH_TRANSITIVITY_ZERO` returns zero.

Returns:

Error code.

See also:

`igraph_transitivity_undirected()`, `igraph_transitivity_avglocal_undirected()`.

Time complexity: O(n*d^2), n is the number of vertices for which the transitivity is calculated, d is the average vertex degree.

igraph_transitivity_avglocal_undirected — Average local transitivity (clustering coefficient).

```
igraph_error_t igraph_transitivity_avglocal_undirected(const igraph_t *graph,
        igraph_real_t *res,
        igraph_transitivity_mode_t mode);
```

The transitivity measures the probability that two neighbors of a vertex are connected. In case of the average local transitivity, this probability is calculated for each vertex and then the average is taken. Vertices with less than two neighbors require special treatment, they will either be left out from the calculation or they will be considered as having zero transitivity, depending on the `mode` argument. Edge directions and edge multiplicities are ignored.

Note that this measure is different from the global transitivity measure (see `igraph_transitivity_undirected()`) as it simply takes the average local transitivity across the whole network.

Clustering coefficient is an alternative name for transitivity.

References:

D. J. Watts and S. Strogatz: Collective dynamics of small-world networks. Nature 393(6684):440-442 (1998).

Arguments:

graph: The input graph. Edge directions and multiplicites are ignored.

res: Pointer to a real variable, the result will be stored here.

mode: Defines how to treat vertices with degree less than two. `IGRAPH_TRANSITIVITY_NAN` leaves them out from averaging, `IGRAPH_TRANSITIVITY_ZERO` includes them with zero transitivity. The result will be `NaN` if the mode is `IGRAPH_TRANSITIVITY_NAN` and there are no vertices with more than one neighbor.

Returns:

Error code.

See also:

`igraph_transitivity_undirected()`, `igraph_transitivity_local_undirected()`.

Time complexity: O(|V|*d^2), |V| is the number of vertices in the graph and d is the average degree.

igraph_transitivity_barrat — Weighted local transitivity of some vertices, as defined by A. Barrat.

```
igraph_error_t igraph_transitivity_barrat(const igraph_t *graph,
                                          igraph_vector_t *res,
                                          const igraph_vs_t vids,
                                          const igraph_vector_t *weights,
                                          igraph_transitivity_mode_t mode);
```

This is a local transitivity, i.e. a vertex-level index. For a given vertex `i`, from all triangles in which it participates we consider the weight of the edges incident on `i`. The transitivity is the sum of these

weights divided by twice the strength of the vertex (see `igraph_strength()`) and the degree of the vertex minus one. See equation (5) in Alain Barrat, Marc Barthelemy, Romualdo Pastor-Satorras, Alessandro Vespignani: The architecture of complex weighted networks, Proc. Natl. Acad. Sci. USA 101, 3747 (2004) at https://doi.org/10.1073/pnas.0400087101 for the exact formula.

Arguments:

graph: The input graph. Edge directions are ignored for directed graphs. Note that the function does *not* work for non-simple graphs.

res: Pointer to an initialized vector, the result will be stored here. It will be resized as needed.

vids: The vertices for which the calculation is performed.

weights: Edge weights. If this is a null pointer, then a warning is given and `igraph_transitivity_local_undirected()` is called.

mode: Defines how to treat vertices with zero strength. `IGRAPH_TRANSITIVITY_NAN` says that the transitivity of these vertices is `NaN`, `IGRAPH_TRANSITIVITY_ZERO` says it is zero.

Returns:

Error code.

Time complexity: O(|V|*d^2), |V| is the number of vertices in the graph, d is the average node degree.

See also:

`igraph_transitivity_undirected()`, `igraph_transitivity_local_undirected()` and `igraph_transitivity_avglocal_undirected()` for other kinds of (non-weighted) transitivity.

`igraph_ecc` — Edge clustering coefficient of some edges.

```
igraph_error_t igraph_ecc(const igraph_t *graph, igraph_vector_t *res,
                          const igraph_es_t eids, igraph_int_t k,
                          igraph_bool_t offset, igraph_bool_t normalize);
```

The edge clustering coefficient `C^(k)_ij` of an edge (i, j) is defined based on the number of k-cycles the edge participates in, `z^(k)_ij`, and the largest number of such cycles it could participate in given the degrees of its endpoints, `s^(k)_ij`. The original definition given in the reference below is:

```
C^(k)_ij = (z^(k)_ij + 1) / s^(k)_ij
```

For `k=3`, `s^(k)_ij = min(d_i - 1, d_j - 1)`, where `d_i` and `d_j` are the edge endpoint degrees. For `k=4`, `s^(k)_ij = (d_i - 1) (d_j - 1)`.

The *`normalize`* and *`offset`* parameters allow for skipping normalization by `s^(k)` and offsetting the cycle count `z^(k)` by one in the numerator of `C^(k)`. Set both to `true` to compute the original definition of this metric.

This function ignores edge multiplicities when listing k-cycles (i.e. `z^(k)`), but not when computing the maximum number of cycles an edge can participate in (`s^(k)`).

Reference:

F. Radicchi, C. Castellano, F. Cecconi, V. Loreto, and D. Parisi, PNAS 101, 2658 (2004). https://doi.org/10.1073/pnas.0400054101

Arguments:

graph: The input graph.

res: Initialized vector, the result will be stored here.

eids: The edges for which the edge clustering coefficient will be computed.

k: Size of cycles to use in calculation. Must be at least 3. Currently only values of 3 and 4 are supported.

offset: Boolean, whether to add one to cycle counts. When `false`, `z^(k)` is used instead of `z^(k) + 1`. In this case the maximum value of the normalized metric is 1. For `k=3` this is achieved for all edges in a complete graph.

normalize: Boolean, whether to normalize cycle counts by the maximum possible count `s^(k)` given the degrees.

Returns:

Error code.

Time complexity: When *k* is 3, O(|V| d log d + |E| d). When *k* is 4, O(|V| d log d + |E| d^2). d denotes the degree of vertices.

Directedness conversion

`igraph_to_directed` — Convert an undirected graph to a directed one.

```
igraph_error_t igraph_to_directed(igraph_t *graph,
                      igraph_to_directed_t mode);
```

If the supplied graph is directed, this function does nothing.

Arguments:

graph: The graph object to convert.

mode: Constant, specifies the details of how exactly the conversion is done. Possible values:

`IGRAPH_TO_DIRECTED_ARBITRARY`	The number of edges in the graph stays the same, an arbitrarily directed edge is created for each undirected edge.
`IGRAPH_TO_DIRECTED_MUTUAL`	Two directed edges are created for each undirected edge, one in each direction.
`IGRAPH_TO_DIRECTED_RANDOM`	Each undirected edge is converted to a randomly oriented directed one.
`IGRAPH_TO_DIRECTED_ACYCLIC`	Each undirected edge is converted to a directed edge oriented from a lower index vertex to a higher index one. If no self-loops were present, then the result is a directed acyclic graph.

Returns:

Error code.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

`igraph_to_undirected` — Convert a directed graph to an undirected one.

```
igraph_error_t igraph_to_undirected(igraph_t *graph,
                          igraph_to_undirected_t mode,
                          const igraph_attribute_combination_t *edge_comb);
```

If the supplied graph is undirected, this function does nothing.

Arguments:

graph: The graph object to convert.

mode: Constant, specifies the details of how exactly the conversion is done. Possible values: `IGRAPH_TO_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; `IGRAPH_TO_UNDIRECTED_COLLAPSE`: one undirected edge will be created for each pair of vertices that are connected with at least one directed edge, no multiple edges will be created. `IGRAPH_TO_UNDIRECTED_MUTUAL` creates an undirected edge for each pair of mutual edges in the directed graph. Non-mutual edges are lost; loop edges are kept unconditionally. This mode might create multiple edges.

edge_comb: What to do with the edge attributes. See the igraph manual section about attributes for details. `NULL` means that the edge attributes are lost during the conversion, *except* when `mode` is `IGRAPH_TO_UNDIRECTED_EACH`, in which case the edge attributes are kept intact.

Returns:

Error code.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

Example 17.32. File `examples/simple/igraph_to_undirected.c`

Spectral properties

`igraph_get_laplacian` — Returns the Laplacian matrix of a graph.

```
igraph_error_t igraph_get_laplacian(
    const igraph_t *graph, igraph_matrix_t *res, igraph_neimode_t mode,
    igraph_laplacian_normalization_t normalization,
    const igraph_vector_t *weights
);
```

The Laplacian matrix `L` of a graph is defined as `L_ij = - A_ij` when `i != j` and `L_ii = d_i - A_ii`. Here `A` denotes the (possibly weighted) adjacency matrix and `d_i` is the degree (or strength, if weighted) of vertex `i`. In directed graphs, the *`mode`* parameter controls whether to use out- or in-degrees. Correspondingly, the rows or columns will sum to zero. In undirected graphs, `A_ii` is taken to be *twice* the number (or total weight) of self-loops, ensuring that `d_i = \sum_j A_ij`. Thus, the Laplacian of an undirected graph is the same as the Laplacian of a directed one obtained by replacing each undirected edge with two reciprocal directed ones.

More compactly, `L = D - A` where the `D` is a diagonal matrix containing the degrees. The Laplacian matrix can also be normalized, with several conventional normalization methods. See `igraph_laplacian_normalization_t` for the methods available in igraph.

The first version of this function was written by Vincent Matossian.

Arguments:

`graph`: Pointer to the graph to convert.

`res`: Pointer to an initialized matrix object, the result is stored here. It will be resized if needed.

`mode`: Controls whether to use out- or in-degrees in directed graphs. If set to `IGRAPH_ALL`, edge directions will be ignored.

`normalization`: The normalization method to use when calculating the Laplacian matrix. See `igraph_laplacian_normalization_t` for possible values.

`weights`: An optional vector containing non-negative edge weights, to calculate the weighted Laplacian matrix. Set it to a null pointer to calculate the unweighted Laplacian.

Returns:

Error code.

Time complexity: O(|V|^2), |V| is the number of vertices in the graph.

Example 17.33. File `examples/simple/igraph_get_laplacian.c`

`igraph_get_laplacian_sparse` — Returns the Laplacian of a graph in a sparse matrix format.

```
igraph_error_t igraph_get_laplacian_sparse(
    const igraph_t *graph, igraph_sparsemat_t *sparseres, igraph_neimode_t mode,
    igraph_laplacian_normalization_t normalization,
    const igraph_vector_t *weights
);
```

See `igraph_get_laplacian()` for the definition of the Laplacian matrix.

The first version of this function was written by Vincent Matossian.

Arguments:

`graph`: Pointer to the graph to convert.

`sparseres`: Pointer to an initialized sparse matrix object, the result is stored here.

mode: Controls whether to use out- or in-degrees in directed graphs. If set to IGRAPH_ALL, edge directions will be ignored.

normalization: The normalization method to use when calculating the Laplacian matrix. See igraph_laplacian_normalization_t for possible values.

weights: An optional vector containing non-negative edge weights, to calculate the weighted Laplacian matrix. Set it to a null pointer to calculate the unweighted Laplacian.

Returns:

Error code.

Time complexity: O(|E|), |E| is the number of edges in the graph.

Example 17.34. File examples/simple/igraph_get_laplacian_sparse.c

igraph_laplacian_normalization_t — Normalization methods for a Laplacian matrix.

```
typedef enum {
    IGRAPH_LAPLACIAN_UNNORMALIZED = 0,
    IGRAPH_LAPLACIAN_SYMMETRIC = 1,
    IGRAPH_LAPLACIAN_LEFT = 2,
    IGRAPH_LAPLACIAN_RIGHT = 3
} igraph_laplacian_normalization_t;
```

Normalization methods for igraph_get_laplacian() and igraph_get_laplacian_sparse(). In the following, A refers to the (possibly weighted) adjacency matrix and D is a diagonal matrix containing degrees (unweighted case) or strengths (weighted case). Out-, in- or total degrees are used according to the *mode* parameter.

Values:

IGRAPH_LAPLACIAN_UNNORMALIZED: Unnormalized Laplacian, L = D - A.

IGRAPH_LAPLACIAN_SYMMETRIC: Symmetrically normalized Laplacian, L = I - D^(-1/2) A D^(-1/2).

IGRAPH_LAPLACIAN_LEFT: Left-stochastic normalized Laplacian, L = I - D^-1 A.

IGRAPH_LAPLACIAN_RIGHT: Right-stochastic normalized Laplacian, L = I - A D^-1.

Non-simple graphs: Multiple and loop edges

igraph_is_simple — Decides whether the input graph is a simple graph.

```
igraph_error_t igraph_is_simple(const igraph_t *graph, igraph_bool_t *res, igra
```

A graph is a simple graph if it does not contain loop edges and multiple edges.

Arguments:

graph: The input graph.

res: Pointer to a boolean constant, the result is stored here.

directed: Whether to consider the directions of edges. `IGRAPH_UNDIRECTED` means that edge directions will be ignored and a directed graph with at least one mutual edge pair will be considered non-simple. `IGRAPH_DIRECTED` means that edge directions will be considered. Ignored for undirected graphs.

Returns:

Error code.

See also:

`igraph_is_loop()` and `igraph_is_multiple()` to find the loops and multiple edges, `igraph_simplify()` to get rid of them, or `igraph_has_multiple()` to decide whether there is at least one multiple edge.

Time complexity: O(|V|+|E|).

`igraph_is_loop` — Find the loop edges in a graph.

```
igraph_error_t igraph_is_loop(const igraph_t *graph, igraph_vector_bool_t *res,
                              igraph_es_t es);
```

A loop edge, also called a self-loop, is an edge from a vertex to itself.

Arguments:

graph: The input graph.

res: Pointer to an initialized boolean vector for storing the result, it will be resized as needed.

es: The edges to check, for all edges supply `igraph_ess_all()` here.

Returns:

Error code.

See also:

`igraph_simplify()` to get rid of loop edges.

Time complexity: O(e), the number of edges to check.

Example 17.35. File `examples/simple/igraph_is_loop.c`

`igraph_has_loop` — Returns whether the graph has at least one loop edge.

```
igraph_error_t igraph_has_loop(const igraph_t *graph, igraph_bool_t *res);
```

A loop edge is an edge from a vertex to itself.

The return value of this function is cached in the graph itself; calling the function multiple times with no modifications to the graph in between will return a cached value in O(1) time.

Arguments:

graph: The input graph.

res: Pointer to an initialized boolean vector for storing the result.

Returns:

Error code.

See also:

`igraph_simplify()` to get rid of loop edges.

Time complexity: O(e), the number of edges to check.

Example 17.36. File `examples/simple/igraph_is_loop.c`

`igraph_count_loops` — Counts the self-loops in the graph.

```
igraph_error_t igraph_count_loops(const igraph_t *graph, igraph_int_t *loop_cou
```

Counts loop edges, i.e. edges whose two endpoints coincide.

Arguments:

graph: The input graph.

loop_count: Pointer to an integer, the number of self-loops will be stored here.

Returns:

Error code.

Time complexity: O(|E|), linear in the number of edges.

Example 17.37. File `examples/simple/igraph_is_loop.c`

`igraph_is_multiple` — Find the multiple edges in a graph.

```
igraph_error_t igraph_is_multiple(const igraph_t *graph, igraph_vector_bool_t *
```

```
                                    igraph_es_t es);
```

An edge is a multiple edge if there is another edge with the same head and tail vertices in the graph.

Note that this function returns true only for the second or more appearances of the multiple edges.

Arguments:

graph: The input graph.

res: Pointer to a boolean vector, the result will be stored here. It will be resized as needed.

es: The edges to check. Supply `igraph_ess_all()` if you want to check all edges.

Returns:

Error code.

See also:

`igraph_count_multiple()`, `igraph_count_multiple_1()`, `igraph_has_multiple()` and `igraph_simplify()`.

Time complexity: O(e*d), e is the number of edges to check and d is the average degree (out-degree in directed graphs) of the vertices at the tail of the edges.

Example 17.38. File `examples/simple/igraph_is_multiple.c`

`igraph_has_multiple` — Check whether the graph has at least one multiple edge.

```
igraph_error_t igraph_has_multiple(const igraph_t *graph, igraph_bool_t *res);
```

An edge is a multiple edge if there is another edge with the same head and tail vertices in the graph.

The return value of this function is cached in the graph itself; calling the function multiple times with no modifications to the graph in between will return a cached value in O(1) time.

Arguments:

graph: The input graph.

res: Pointer to a boolean variable, the result will be stored here.

Returns:

Error code.

See also:

`igraph_count_multiple()`, `igraph_is_multiple()` and `igraph_simplify()`.

Time complexity: O(e*d), e is the number of edges to check and d is the average degree (out-degree in directed graphs) of the vertices at the tail of the edges.

Example 17.39. File `examples/simple/igraph_has_multiple.c`

`igraph_count_multiple` — The multiplicity of some edges in a graph.

```
igraph_error_t igraph_count_multiple(const igraph_t *graph, igraph_vector_int_t
                                     igraph_es_t es);
```

An edge is called a multiple edge when there is one or more other edge between the same two vertices. The multiplicity of an edge is the number of edges between its endpoints.

Arguments:

`graph`: The input graph.

`res`: Pointer to a vector, the result will be stored here. It will be resized as needed.

`es`: The edges to check. Supply `igraph_ess_all()` if you want to check all edges.

Returns:

Error code.

See also:

`igraph_count_multiple_1()` if you only need the multiplicity of a single edge; `igraph_is_multiple()` if you are only interested in whether the graph has at least one edge with multiplicity greater than one; `igraph_simplify()` to ensure that the graph has no multiple edges.

Time complexity: O(E d), E is the number of edges to check and d is the average degree (out-degree in directed graphs) of the vertices at the tail of the edges.

`igraph_count_multiple_1` — The multiplicity of a single edge in a graph.

```
igraph_error_t igraph_count_multiple_1(const igraph_t *graph, igraph_int_t *res,
                                       igraph_int_t eid);
```

Arguments:

`graph`: The input graph.

`res`: Pointer to an iteger, the result will be stored here.

`eid`: The ID of the edge to check.

Returns:

Error code.

See also:

`igraph_count_multiple()` if you need the multiplicity of multiple edges; `igraph_is_multiple()` if you are only interested in whether the graph has at least one edge with multiplicity greater than one; `igraph_simplify()` to ensure that the graph has no multiple edges.

Time complexity: O(d), where d is the out-degree of the tail of the edge.

Mixing patterns and degree correlations

`igraph_assortativity_nominal` — Assortativity of a graph based on vertex categories.

```
igraph_error_t igraph_assortativity_nominal(
        const igraph_t *graph, const igraph_vector_t *weights,
        const igraph_vector_int_t *types,
        igraph_real_t *res,
        igraph_bool_t directed, igraph_bool_t normalized);
```

Assuming the vertices of the input graph belong to different categories, this function calculates the assortativity coefficient of the graph. The assortativity coefficient is between minus one and one and it is one if all connections stay within categories, it is minus one, if the network is perfectly disassortative. For a randomly connected network it is (asymptotically) zero.

The unnormalized version, computed when *`normalized`* is set to false, is identical to the modularity, and is defined as follows for directed networks:

`1/m sum_ij (A_ij - k^out_i k^in_j / m) d(i,j)`,

where `m` denotes the number of edges, `A_ij` is the adjacency matrix, `k^out` and `k^in` are the out- and in-degrees, and `d(i,j)` is one if vertices `i` and `j` are in the same category and zero otherwise.

The normalized assortativity coefficient is obtained by dividing the previous expression by

`1/m sum_ij (m - k^out_i k^in_j d(i,j) / m)`.

It can take any value within the interval [-1, 1].

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

References:

M. E. J. Newman: Mixing patterns in networks, Phys. Rev. E 67, 026126 (2003) https://doi.org/10.1103/PhysRevE.67.026126. See section II and equation (2) for the definition of the concept.

For an educational overview of assortativity, see M. E. J. Newman, Networks: An Introduction, Oxford University Press (2010). https://doi.org/10.1093/acprof%3Aoso/9780199206650.001.0001.

Arguments:

`graph`: The input graph, it can be directed or undirected.

`weights`: Weighted nominal assortativity is not currently implemented. Pass `NULL` to ignore.

`types`: Integer vector giving the vertex categories. The types are represented by integers starting at zero.

`res`: Pointer to a real variable, the result is stored here.

directed: Boolean, it gives whether to consider edge directions in a directed graph. It is ignored for undirected graphs.

normalized: Boolean, whether to compute the usual normalized assortativity. The unnormalized version is identical to modularity. Supply true here to compute the standard assortativity.

Returns:

Error code.

Time complexity: O(|E|+t), |E| is the number of edges, t is the number of vertex types.

See also:

`igraph_assortativity()` for computing the assortativity based on continuous vertex values instead of discrete categories. `igraph_modularity()` to compute generalized modularity. `igraph_joint_type_distribution()` to obtain the mixing matrix.

Example 17.40. File `examples/simple/igraph_assortativity_nominal.c`

`igraph_assortativity` — Assortativity based on numeric properties of vertices.

```
igraph_error_t igraph_assortativity(
        const igraph_t *graph, const igraph_vector_t *weights,
        const igraph_vector_t *values, const igraph_vector_t *values_in,
        igraph_real_t *res,
        igraph_bool_t directed, igraph_bool_t normalized);
```

This function calculates the assortativity coefficient of a graph based on given values `x_i` for each vertex `i`. This type of assortativity coefficient equals the Pearson correlation of the values at the two ends of the edges.

The unnormalized covariance of values, computed when *normalized* is set to false, is defined as follows in a directed graph:

```
cov(x_out, x_in) = 1/m sum_ij (A_ij - k^out_i k^in_j / m) x_i x_j,
```

where `m` denotes the number of edges, `A_ij` is the adjacency matrix, and `k^out` and `k^in` are the out- and in-degrees. `x_out` and `x_in` refer to the sets of vertex values at the start and end of the directed edges.

The normalized covariance, i.e. Pearson correlation, is obtained by dividing the previous expression by `sqrt(var(x_out)) sqrt(var(x_in))`, where

```
var(x_out) = 1/m sum_i k^out_i x_i^2 - (1/m sum_i k^out_i x_i^2)^2
```

```
var(x_in) = 1/m sum_j k^in_j x_j^2 - (1/m sum_j k^in_j x_j^2)^2
```

Undirected graphs are effectively treated as directed graphs where all edges are reciprocal. Therefore, self-loops are effectively considered twice in undirected graphs.

When edge weights are given, they are effectively treated as edge multiplicities. The above formulas are valid for weighted graph as well when `m` is interpreted as the total edge weight (instead of the edge count) and `k` as vertex strengths (instead of degrees).

References:

M. E. J. Newman: Mixing patterns in networks, Phys. Rev. E 67, 026126 (2003) https://doi.org/10.1103/PhysRevE.67.026126. See section III and equation (21) for the definition, and equation (26) for performing the calculation in directed graphs with the degrees as values.

M. E. J. Newman: Assortative mixing in networks, Phys. Rev. Lett. 89, 208701 (2002) https://doi.org/10.1103/PhysRevLett.89.208701. See equation (4) for performing the calculation in undirected graphs with the degrees as values.

For an educational overview of the concept of assortativity, see M. E. J. Newman, Networks: An Introduction, Oxford University Press (2010). https://doi.org/10.1093/acprof%3Aoso/9780199206650.001.0001.

Arguments:

graph: The input graph, it can be directed or undirected.

weights: The edge weights. Pass `NULL` to compute unweighed assortativity, which in effect assumes all weights to be 1.

values: The vertex values, these can be arbitrary numeric values.

values_in: A second value vector to be used 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 a null pointer and the undirected assortativity coefficient is being calculated.

res: Pointer to a real variable, the result is stored here.

directed: Boolean, whether to consider edge directions for directed graphs. It is ignored for undirected graphs.

normalized: Boolean, whether to compute the normalized covariance, i.e. Pearson correlation. Supply true here to compute the standard assortativity.

Returns:

Error code.

Time complexity: O(|E|), linear in the number of edges of the graph.

See also:

`igraph_assortativity_nominal()` if you have discrete vertex categories instead of numeric labels, and `igraph_assortativity_degree()` for the special case of assortativity based on vertex degrees.

`igraph_assortativity_degree` — Assortativity of a graph based on vertex degree.

```
igraph_error_t igraph_assortativity_degree(const igraph_t *graph,
                                           igraph_real_t *res,
                                           igraph_bool_t directed);
```

Assortativity based on vertex degree, please see the discussion at the documentation of `igraph_assortativity()` for details. This function simply calls `igraph_assortativity()` with the

degrees as the vertex values and normalization enabled. In the directed case, it uses out-degrees as out-values and in-degrees as in-values.

For regular graphs, i.e. graphs in which all vertices have the same degree, computing degree correlations is not meaningful, and this function returns NaN.

Arguments:

graph: The input graph, it can be directed or undirected.

res: Pointer to a real variable, the result is stored here.

directed: Boolean, 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.

Returns:

Error code.

Time complexity: O(|E|+|V|), |E| is the number of edges, |V| is the number of vertices.

See also:

`igraph_assortativity()` for the general function calculating assortativity for any kind of numeric vertex values, and `igraph_joint_degree_distribution()` to get the complete joint degree distribution.

Example 17.41. File `examples/simple/igraph_assortativity_degree.c`

`igraph_avg_nearest_neighbor_degree` — Average neighbor degree.

```
igraph_error_t igraph_avg_nearest_neighbor_degree(const igraph_t *graph,
                                        igraph_vs_t vids,
                                        igraph_neimode_t mode,
                                        igraph_neimode_t neighbor_degree_mode,
                                        igraph_vector_t *knn,
                                        igraph_vector_t *knnk,
                                        const igraph_vector_t *weights);
```

Calculates the average degree of the neighbors for each vertex (*knn*), and optionally, the same quantity as a function of the vertex degree (*knnk*).

For isolated vertices *knn* is set to NaN. The same is done in *knnk* for vertex degrees that don't appear in the graph.

The weighted version computes a weighted average of the neighbor degrees as

`k_nn_u = 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*. This is equation (6) in the reference below.

When only the k_nn(k) degree correlation function is needed, igraph_degree_correlation_vector() can be used as well. This function provides more flexible control over how degree at each end of directed edges are computed.

Reference:

A. Barrat, M. Barthélemy, R. Pastor-Satorras, and A. Vespignani, The architecture of complex weighted networks, Proc. Natl. Acad. Sci. USA 101, 3747 (2004). https://dx.doi.org/10.1073/pnas.0400087101

Arguments:

graph:	The input graph. It may be directed.
vids:	The vertices for which the calculation is performed.
mode:	The type of neighbors to consider in directed graphs. IGRAPH_OUT considers out-neighbors, IGRAPH_IN in-neighbors and IGRAPH_ALL ignores edge directions.
neighbor_degree_mode:	The type of degree to average in directed graphs. IGRAPH_OUT averages out-degrees, IGRAPH_IN averages in-degrees and IGRAPH_ALL ignores edge directions for the degree calculation.
knn:	Pointer to an initialized vector, the result will be stored here. It will be resized as needed. Supply a NULL pointer here if you only want to calculate knnk.
knnk:	Pointer to an initialized vector, the average neighbor degree as a function of the vertex degree is stored here. This is sometimes referred to as the k_nn(k) degree correlation function. The first (zeroth) element is for degree one vertices, etc. The calculation is done based only on the vertices *vids*. Supply a NULL pointer here if you don't want to calculate this.
weights:	Optional edge weights. Supply a null pointer here for the non-weighted version.

Returns:

Error code.

See also:

igraph_degree_correlation_vector() for computing only the degree correlation function, with more flexible control over degree computations.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

Example 17.42. File examples/simple/igraph_avg_nearest_neighbor_degree.c

igraph_degree_correlation_vector — Degree correlation function.

```
igraph_error_t igraph_degree_correlation_vector(
        const igraph_t *graph, const igraph_vector_t *weights,
        igraph_vector_t *knnk,
        igraph_neimode_t from_mode, igraph_neimode_t to_mode,
        igraph_bool_t directed_neighbors);
```

Computes the degree correlation function `k_nn(k)`, defined as the mean degree of the targets of directed edges whose source has degree `k`. The averaging is done over all directed edges. The *`from_mode`* and *`to_mode`* parameters control how the source and target vertex degrees are computed. This way the out-in, out-out, in-in and in-out degree correlation functions can all be computed.

In undirected graphs, edges are treated as if they were a pair of reciprocal directed ones.

If P_ij is the joint degree distribution of the graph, computable with `igraph_joint_degree_distribution()`, then `k_nn(k) = (sum_j j P_kj) / (sum_j P_kj)`.

The function `igraph_avg_nearest_neighbor_degree()`, whose main purpose is to calculate the average neighbor degree for each vertex separately, can also compute `k_nn(k)`. It differs from this function in that it can take a subset of vertices to base the calculation on, but it does not allow the same fine-grained control over how degrees are computed.

References:

R. Pastor-Satorras, A. Vazquez, A. Vespignani: Dynamical and Correlation Properties of the Internet, Phys. Rev. Lett., vol. 87, pp. 258701 (2001). https://doi.org/10.1103/PhysRevLett.87.258701

A. Vazquez, R. Pastor-Satorras, A. Vespignani: Large-scale topological and dynamical properties of the Internet, Phys. Rev. E, vol. 65, pp. 066130 (2002). https://doi.org/10.1103/PhysRevE.65.066130

A. Barrat, M. Barthélemy, R. Pastor-Satorras, and A. Vespignani, The architecture of complex weighted networks, Proc. Natl. Acad. Sci. USA 101, 3747 (2004). https://dx.doi.org/10.1073/pnas.0400087101

Arguments:

`graph`: The input graph.

`weights`: An optional weight vector. If not `NULL`, weighted averages will be computed.

`knnk`: An initialized vector, the result will be written here. `knnk[d]` will contain the mean degree of vertices connected to by vertices of degree `d`. Note that in contrast to `igraph_avg_nearest_neighbor_degree()`, `d=0` is also included.

`from_mode`: How to compute the degree of sources? Can be `IGRAPH_OUT` for out-degree, `IGRAPH_IN` for in-degree, or `IGRAPH_ALL` for total degree. Ignored in undirected graphs.

`to_mode`: How to compute the degree of sources? Can be `IGRAPH_OUT` for out-degree, `IGRAPH_IN` for in-degree, or `IGRAPH_ALL` for total degree. Ignored in undirected graphs.

`directed_neighbors`: Whether to consider `u -> v` connections to be directed. Undirected connections are treated as reciprocal directed ones, i.e. both `u -> v` and `v -> u` will be considered. Ignored in undirected graphs.

Returns:

Error code.

See also:

`igraph_avg_nearest_neighbor_degree()` for computing the average neighbour degree of a set of vertices, `igraph_joint_degree_distribution()` to get the complete joint degree distribution, and `igraph_assortativity_degree()` to compute the degree assortativity.

Time complexity: O(|E| + |V|)

`igraph_joint_type_distribution` — Mixing matrix for vertex categories.

```
igraph_error_t igraph_joint_type_distribution(
        const igraph_t *graph, const igraph_vector_t *weights,
        igraph_matrix_t *p,
        const igraph_vector_int_t *from_types, const igraph_vector_int_t *to_ty
        igraph_bool_t directed, igraph_bool_t normalized);
```

Computes the mixing matrix M_ij, i.e. the joint distribution of vertex types at the endpoints directed of edges. Categories are represented by non-negative integer indices, passed in *`from_types`* and *`to_types`*. The row and column counts of *`m`* will be one larger than the largest source and target type, respectively. Re-index type vectors using `igraph_reindex_membership()` if they are not contiguous integers, to avoid producing a very large matrix.

M_ij is proportional to the probability that a randomly chosen ordered pair of vertices have types `i` and `j`.

When there is a single categorization of vertices, i.e. *`from_types`* and *`to_types`* are the same, M_ij is related to the modularity (`igraph_modularity()`) and nominal assortativity (`igraph_assortativity_nominal()`). Let `a_i = sum_j M_ij` and `b_j = sum_i M_ij`. If M_ij is normalized, i.e. `sum_ij M_ij = 1`, and the types represent membership in vertex partitions, then the modularity of the partitioning can be computed as

```
Q = sum_ii M_ii - sum_i a_i b_i
```

The normalized nominal assortativity is

```
Q / (1 - sum_i a_i b_i)
```

`igraph_joint_degree_distribution()` is a special case of this function, with categories consisting vertices of the same degree.

References:

M. E. J. Newman: Mixing patterns in networks, Phys. Rev. E 67, 026126 (2003) https://doi.org/10.1103/PhysRevE.67.026126.

Arguments:

`graph`: The input graph.

`weights`: A vector containing the weights of the edges. If passing a `NULL` pointer, edges will be assumed to have unit weights.

`p`: The mixing matrix `M_ij` will be stored here.

`from_types`: Vertex types for source vertices. These must be non-negative integers.

to_types: Vertex types for target vertices. These must be non-negative integers. If NULL, it is assumed to be the same as *from_types*.

directed: Whether to treat edges are directed. Ignored for undirected graphs.

normalized: Whether to normalize the matrix so that entries sum to 1.0. If false, matrix entries will be connection counts. Normalization is not meaningful if some edge weights are negative.

Returns:

Error code.

See also:

igraph_joint_degree_distribution() to compute the joint distribution of vertex degrees; igraph_modularity() to compute the modularity of a vertex partitioning; igraph_assortativity_nominal() to compute assortativity based on vertex categories.

Time complexity: O(E), where E is the number of edges in the input graph.

igraph_joint_degree_distribution — The joint degree distribution of a graph.

```
igraph_error_t igraph_joint_degree_distribution(
        const igraph_t *graph, const igraph_vector_t *weights,
        igraph_matrix_t *p,
        igraph_neimode_t from_mode, igraph_neimode_t to_mode,
        igraph_bool_t directed_neighbors,
        igraph_bool_t normalized,
        igraph_int_t max_from_degree, igraph_int_t max_to_degree);
```

Computes the joint degree distribution P_ij of a graph, used in the study of degree correlations. P_ij is the probability that a randomly chosen ordered pair of *connected* vertices have degrees i and j.

In directed graphs, directionally connected u -> v pairs are considered. The joint degree distribution of an undirected graph is the same as that of the corresponding directed graph in which all connection are bidirectional, assuming that *from_mode* is IGRAPH_OUT, *to_mode* is IGRAPH_IN and *directed_neighbors* is true.

When *normalized* is false, sum_ij P_ij gives the total number of connections in a directed graph, or twice that value in an undirected graph. The sum is taken over ordered (i,j) degree pairs.

The joint degree distribution relates to other concepts used in the study of degree correlations. If P_ij is normalized then the degree correlation function k_nn(k) is obtained as

k_nn(k) = (sum_j j P_kj) / (sum_j P_kj).

The non-normalized degree assortativity is obtained as

a = sum_ij i j (P_ij - q_i r_j),

where q_i = sum_k P_ik and r_j = sum_k P_kj.

Note that the joint degree distribution P_ij is similar, but not identical to the joint degree matrix J_ij computed by igraph_joint_degree_matrix(). If the graph is undirected, then the diagonal entries of an unnormalized P_ij are double that of J_ij, as any undirected connection between

same-degree vertices is counted in both directions. In contrast to `igraph_joint_degree_matrix()`, this function returns matrices which include the row and column corresponding to zero degrees. In directed graphs, this row and column is not necessarily zero when *`from_mode`* is different from `IGRAPH_OUT` or *`to_mode`* is different from `IGRAPH_IN`.

References:

M. E. J. Newman: Mixing patterns in networks, Phys. Rev. E 67, 026126 (2003) https://doi.org/10.1103/PhysRevE.67.026126.

Arguments:

`graph`:	A pointer to an initialized graph object.
`weights`:	A vector containing the weights of the edges. If passing a `NULL` pointer, edges will be assumed to have unit weights.
`p`:	A pointer to an initialized matrix that will be resized. The `P_ij` value will be written into `p[i,j]`.
`from_mode`:	How to compute the degree of sources? Can be `IGRAPH_OUT` for out-degree, `IGRAPH_IN` for in-degree, or `IGRAPH_ALL` for total degree. Ignored in undirected graphs.
`to_mode`:	How to compute the degree of targets? Can be `IGRAPH_OUT` for out-degree, `IGRAPH_IN` for in-degree, or `IGRAPH_ALL` for total degree. Ignored in undirected graphs.
`directed_neighbors`:	Whether to consider `u -> v` connections to be directed. Undirected connections are treated as reciprocal directed ones, i.e. both `u -> v` and `v -> u` will be considered. Ignored in undirected graphs.
`normalized`:	Whether to normalize the matrix so that entries sum to 1.0. If false, matrix entries will be connection counts. Normalization is not meaningful if some edge weights are negative.
`max_from_degree`:	The largest source vertex degree to consider. If negative or `IGRAPH_UNLIMITED`, the largest source degree will be used. The row count of the result matrix is one larger than this value.
`max_to_degree`:	The largest target vertex degree to consider. If negative or `IGRAPH_UNLIMITED`, the largest target degree will be used. The column count of the result matrix is one larger than this value.

Returns:

Error code.

See also:

`igraph_joint_degree_matrix()` for computing the joint degree matrix; `igraph_assortativity_degree()` and `igraph_assortativity()` for degree correlations coefficients, and `igraph_degree_correlation_vector()` for the degree correlation function.

Time complexity: O(E), where E is the number of edges in the input graph.

`igraph_joint_degree_matrix` — The joint degree matrix of a graph.

```
igraph_error_t igraph_joint_degree_matrix(
        const igraph_t *graph, const igraph_vector_t *weights,
        igraph_matrix_t *jdm,
        igraph_int_t max_out_degree, igraph_int_t max_in_degree);
```

In graph theory, the joint degree matrix `J_ij` of a graph gives the number of edges, or sum of edge weights, between vertices of degree `i` and degree `j`. This function stores `J_ij` into `jdm[i-1, j-1]`. Each edge, including self-loops, is counted precisely once, both in undirected and directed graphs.

`sum_(i,j) J_ij` is the total number of edges (or total edge weight) `m` in the graph, where `(i,j)` refers to ordered or unordered pairs in directed and undirected graphs, respectively. Thus `J_ij / m` is the probability that an edge chosen at random (with probability proportional to its weight) connects vertices with degrees `i` and `j`.

Note that `J_ij` is similar, but not identical to the joint degree *distribution*, computed by `igraph_joint_degree_distribution()`, which is defined for *ordered* `(i, j)` degree pairs even in the undirected case. When considering undirected graphs, the diagonal of the joint degree distribution is twice that of the joint degree matrix.

References:

Isabelle Stanton and Ali Pinar: Constructing and sampling graphs with a prescribed joint degree distribution. ACM J. Exp. Algorithmics 17, Article 3.5 (2012). https://doi.org/10.1145/2133803.2330086

Arguments:

`graph`: A pointer to an initialized graph object.

`weights`: A vector containing the weights of the edges. If passing a `NULL` pointer, edges will be assumed to have unit weights, i.e. the matrix entries will be connection counts.

`jdm`: A pointer to an initialized matrix that will be resized. The values will be written here.

`max_out_degree`: Number of rows in the result, i.e. the largest (out-)degree to consider. If negative or `IGRAPH_UNLIMITED`, the largest (out-)degree of the graph will be used.

`max_in_degree`: Number of columns in the result, i.e. the largest (in-)degree to consider. If negative or `IGRAPH_UNLIMITED`, the largest (in-)degree of the graph will be used.

Returns:

Error code.

See also:

`igraph_joint_degree_distribution()` to count ordered vertex pairs instead of edges, or to obtain a normalized matrix.

Time complexity: O(E), where E is the number of edges in input graph.

`igraph_rich_club_sequence` — Density sequence of subgraphs formed by sequential vertex removal.

```
igraph_error_t igraph_rich_club_sequence(
        const igraph_t *graph, const igraph_vector_t *weights,
        igraph_vector_t *res,
        const igraph_vector_int_t *vertex_order,
        igraph_bool_t normalized,
        igraph_bool_t loops, igraph_bool_t directed);
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

This function takes a graph and a vertex ordering as input, sequentially removes the vertices in the given order, and calculates the density of the remaining subgraph after each removal.

Density is calculated as the ratio of the number of edges (or total edge weight, if weighted) to the number of total possible edges in the graph. The latter is dependent on whether the graph is directed and whether self-loops are assumed to be possible: for undirected graphs without self-loops, this total is given by `n(n-1)/2`, and for directed graphs by `n(n-1)`. When self-loops are allowed, these are adjusted to `n(n+1)/2` for undirected and `n^2` for directed graphs.

Vertex order can be sorted by degree so that the resulting density sequence helps reveal how interconnected a graph is across different degree levels, or the presence of a "rich-club" effect.

Arguments:

graph: The graph object to analyze.

weights: Vector of edge weights. If `NULL` all weights are assumed to be 1.

res: Initialized vector, the result will be written here. `res[i]` contain the density of the remaining graph after `i` vertices have been removed. If *normalized* is set to `false`, it contains the remaining edge count (or remaining total edge weights if weights were given).

vertex_order: Vector giving the order in which vertices are removed.

normalized: If `false`, return edge counts (or total edge weights). If `true`, divide by the largest possible edge count to obtain densities.

loops: Whether self-loops are assumed to be possible. Ignored when normalized is not requested.

directed: If false, directed graphs will be treated as undirected. Ignored with undirected graphs.

Returns:

Error code: `IGRAPH_EINVAL`: invalid vertex_order vector and/or weight vector lengths

Time complexity: O(|V| + |E|) where |V| is the number of vertices and |E| the number of edges in the graph given.

See also:

`igraph_density()`, which uses the same calculation of total possible edges.

K-cores and k-trusses

`igraph_coreness` — The coreness of the vertices in a graph.

```
igraph_error_t igraph_coreness(const igraph_t *graph,
        igraph_vector_int_t *cores, igraph_neimode_t mode);
```

The k-core of a graph is a maximal subgraph in which each vertex has at least degree k. (Degree here means the degree in the subgraph of course.). The coreness of a vertex is the highest order of a k-core containing the vertex.

This function implements the algorithm presented in Vladimir Batagelj, Matjaz Zaversnik: An O(m) Algorithm for Cores Decomposition of Networks. https://arxiv.org/abs/cs/0310049

Arguments:

graph: The input graph.

cores: Pointer to an initialized vector, the result of the computation will be stored here. It will be resized as needed. For each vertex it contains the highest order of a core containing the vertex.

mode: For directed graph it specifies whether to calculate in-cores, out-cores or the undirected version. It is ignored for undirected graphs. Possible values: `IGRAPH_ALL` undirected version, `IGRAPH_IN` in-cores, `IGRAPH_OUT` out-cores.

Returns:

Error code.

Time complexity: O(|E|), the number of edges.

`igraph_trussness` — Finding the "trussness" of the edges in a network.

```
igraph_error_t igraph_trussness(const igraph_t* graph, igraph_vector_int_t* trus
```

A k-truss is a subgraph in which every edge occurs in at least `k-2` triangles in the subgraph. The trussness of an edge indicates the highest k-truss that the edge occurs in.

This function returns the highest `k` for each edge. If you are interested in a particular k-truss subgraph, you can subset the graph to those edges which are `>= k` because each k-truss is a subgraph of a `(k-1)`-truss Thus, to get all 4-trusses, take `k >= 4` because the 5-trusses, 6-trusses, etc. need to be included.

The current implementation of this function iteratively decrements support of each edge using O(|E|) space and O(|E|^1.5) time. The implementation does not support multigraphs; use `igraph_simplify()` to collapse edges before calling this function.

Reference:

See Algorithm 2 in: Wang, Jia, and James Cheng. "Truss decomposition in massive networks." Proceedings of the VLDB Endowment 5.9 (2012): 812-823. https://doi.org/10.14778/2311906.2311909

Arguments:

graph: The input graph. Loop edges are allowed; multigraphs are not.

truss: Pointer to initialized vector of truss values that will indicate the highest k-truss each edge occurs in. It will be resized as needed.

Returns:

Error code.

Time complexity: It should be O(|E|^1.5) according to the reference.

Maximum cardinality search and chordal graphs

`igraph_maximum_cardinality_search` — Maximum cardinality search.

```
igraph_error_t igraph_maximum_cardinality_search(const igraph_t *graph,
                                        igraph_vector_int_t *alpha,
                                        igraph_vector_int_t *alpham1);
```

This function implements the maximum cardinality search algorithm. It computes a rank *`alpha`* for each vertex, such that visiting vertices in decreasing rank order corresponds to always choosing the vertex with the most already visited neighbors as the next one to visit.

Maximum cardinality search is useful in deciding the chordality of a graph. A graph is chordal if and only if any two neighbors of a vertex which are higher in rank than it are connected to each other.

References:

Robert E Tarjan and Mihalis Yannakakis: 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, 1984. https://doi.org/10.1137/0213035

Arguments:

graph: The input graph. Edge directions will be ignored.

`alpha`: Pointer to an initialized vector, the result is stored here. It will be resized, as needed. Upon return it contains the rank of the each vertex in the range 0 to `n - 1`, where `n` is the number of vertices.

`alpham1`: Pointer to an initialized vector or a `NULL` pointer. If not `NULL`, then the inverse of *`alpha`* is stored here. In other words, the elements of *`alpham1`* are vertex IDs in reverse maximum cardinality search order.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in terms of the number of vertices and edges.

See also:

`igraph_is_chordal()`.

igraph_is_chordal — Decides whether a graph is chordal.

```
igraph_error_t igraph_is_chordal(const igraph_t *graph,
                      const igraph_vector_int_t *alpha,
                      const igraph_vector_int_t *alpham1,
                      igraph_bool_t *chordal,
                      igraph_vector_int_t *fill_in,
                      igraph_t *newgraph);
```

A graph is chordal if each of its cycles of four or more nodes has a chord, i.e. 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. If either *alpha* or *alpham1* is given, then the other is calculated by taking simply the inverse. If neither are given, then igraph_maximum_cardinality_search() is called to calculate them.

Arguments:

graph: The input graph. Edge directions will be ignored.

alpha: Either an alpha vector coming from igraph_maximum_cardinality_search() (on the same graph), or a NULL pointer.

alpham1: Either an inverse alpha vector coming from igraph_maximum_cardinality_search() (on the same graph) or a NULL pointer.

chordal: Pointer to a boolean. If not NULL the result is stored here.

fill_in: Pointer to an initialized vector, or a NULL pointer. If not a NULL pointer, then the fill-in, also called the chordal completion of the graph is stored here. The chordal completion is a set of edges that are needed to make the graph chordal. The vector is resized as needed. Note that the chordal completion returned by this function may not be minimal, i.e. some of the returned fill-in edges may not be needed to make the graph chordal.

newgraph: Pointer to an uninitialized graph, or a NULL pointer. If not a null pointer, then a new triangulated graph is created here. This essentially means adding the fill-in edges to the original graph.

Returns:

Error code.

Time complexity: O(n).

See also:

igraph_maximum_cardinality_search().

Matchings

igraph_is_matching — Checks whether the given matching is valid for the given graph.

```
igraph_error_t igraph_is_matching(const igraph_t *graph,
                        const igraph_vector_bool_t *types, const igraph_vector_i
                        igraph_bool_t *result);
```

This function checks a matching vector and verifies whether its length matches the number of vertices in the given graph, its values are between -1 (inclusive) and the number of vertices (exclusive), 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.

Arguments:

graph: The input graph. It can be directed but the edge directions will be ignored.

types: If the graph is bipartite and you are interested in bipartite matchings only, pass the vertex types here. If the graph is non-bipartite, simply pass `NULL`.

matching: The matching itself. It must be a vector where element i contains the ID of the vertex that vertex i is matched to, or -1 if vertex i is unmatched.

result: Pointer to a boolean variable, the result will be returned here.

See also:

`igraph_is_maximal_matching()` if you are also interested in whether the matching is maximal (i.e. non-extendable).

Time complexity: O(|V|+|E|) where |V| is the number of vertices and |E| is the number of edges.

Example 17.43. File `examples/simple/igraph_maximum_bipartite_matching.c`

`igraph_is_maximal_matching` — Checks whether a matching in a graph is maximal.

```
igraph_error_t igraph_is_maximal_matching(const igraph_t *graph,
                                const igraph_vector_bool_t *types, const igraph_v
                                igraph_bool_t *result);
```

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.

Arguments:

graph: The input graph. It can be directed but the edge directions will be ignored.

types: If the graph is bipartite and you are interested in bipartite matchings only, pass the vertex types here. If the graph is non-bipartite, simply pass `NULL`.

matching: The matching itself. It must be a vector where element i contains the ID of the vertex that vertex i is matched to, or -1 if vertex i is unmatched.

result: Pointer to a boolean variable, the result will be returned here.

Returns:

Error code.

See also:

`igraph_is_matching()` if you are only interested in whether a matching vector is valid for a given graph.

Time complexity: O(|V|+|E|) where |V| is the number of vertices and |E| is the number of edges.

Example 17.44. File `examples/simple/igraph_maximum_bipartite_matching.c`

`igraph_maximum_bipartite_matching` — Calculates a maximum matching in a bipartite graph.

```
igraph_error_t igraph_maximum_bipartite_matching(const igraph_t *graph,
                                        const igraph_vector_bool_t *types, igraph
                                        igraph_real_t *matching_weight, igraph_ve
                                        const igraph_vector_t *weights, igraph_re
```

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.

References: Cherkassky BV, Goldberg AV, Martin P, Setubal JC and Stolfi J: Augment or push: A computational study of bipartite matching and unit-capacity flow algorithms. ACM Journal of Experimental Algorithmics 3, 1998.

Kaya K, Langguth J, Manne F and Ucar B: Experiments on push-relabel-based maximum cardinality matching algorithms for bipartite graphs. Technical Report TR/PA/11/33 of the Centre Europeen de Recherche et de Formation Avancee en Calcul Scientifique, 2011.

Arguments:

`graph`:	The input graph. It can be directed but the edge directions will be ignored.
`types`:	Boolean vector giving the vertex types of the graph.
`matching_size`:	The size of the matching (i.e. the number of matched vertex pairs will be returned here). It may be `NULL` if you don't need this.
`matching_weight`:	The weight of the matching if the edges are weighted, or the size of the matching again if the edges are unweighted. It may be `NULL` if you don't need this.
`matching`:	The matching itself. It must be a vector where element i contains the ID of the vertex that vertex i is matched to, or -1 if vertex i is unmatched.
`weights`:	A null pointer (=no edge weights), or a vector giving the weights of the edges. Note that the algorithm is stable only for integer weights.
`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. It is advised to pass a value derived from the `DBL_EPSILON` constant in `float`.h here. If you are running the algorithm with no `weights` vector, this argument is ignored.

Returns:

Error code.

Time complexity: O(sqrt(|V|) |E|) for unweighted graphs (according to the technical report referenced above), O(|V||E|) for weighted graphs.

Example 17.45. File `examples/simple/igraph_maximum_bipartite_matching.c`

Unfolding a graph into a tree

`igraph_unfold_tree` — Unfolding a graph into a tree, by possibly multiplicating its vertices.

```
igraph_error_t igraph_unfold_tree(const igraph_t *graph, igraph_t *tree,
                      igraph_neimode_t mode, const igraph_vector_int_t *roots,
                      igraph_vector_int_t *vertex_index);
```

A graph is converted into a tree (or forest, if it is unconnected), by performing a breadth-first search on it, and replicating vertices that were found a second, third, etc. time.

Arguments:

`graph`: The input graph, it can be either directed or undirected.

`tree`: Pointer to an uninitialized graph object, the result is stored here.

`mode`: For directed graphs; whether to follow paths along edge directions (`IGRAPH_OUT`), or the opposite (`IGRAPH_IN`), or ignore edge directions completely (`IGRAPH_ALL`). It is ignored for undirected graphs.

`roots`: A numeric vector giving the root vertex, or vertices (if the graph is not connected), to start from.

`vertex_index`: Pointer to an initialized vector, or a null pointer. If not a null pointer, then a mapping from the vertices in the new graph to the ones in the original is created here.

Returns:

Error code.

Time complexity: O(n+m), linear in the number vertices and edges.

Other operations

`igraph_density` — Calculate the density of a graph.

```
igraph_error_t igraph_density(
        const igraph_t *graph,
        const igraph_vector_t *weights,
        igraph_real_t *res,
        igraph_bool_t loops);
```

The density of a graph is simply the ratio of the actual number of its edges and the largest possible number of edges it could have. The maximum number of edges depends on interpretation: are vertices allowed to have a connection to themselves? This is controlled by the *loops* parameter.

The classic definition of the density is formulated for unweighted graphs without multi-edges. This function allows multigraphs and weighted graphs as well. In this case, it computes the ratio of the total edge weight to the largest possible number of adjacent vertex pairs the graph could have. This value may be larger than 1.

If you need the density concept for simple graphs, make sure to eliminate any multi-edges appropriately. This can be done using `igraph_simplify()`.

Arguments:

graph: The input graph object. It must not have parallel edges.

res: Pointer to a real number, the result will be stored here.

weights: Vector of edge weights. Pass `NULL` to to perform an unweighted density calculation.

loops: Boolean constant, whether to include self-loops in the calculation. If this constant is `true` then loop edges are thought to be possible in the graph (this does not necessarily mean that the graph really contains any loops). If this is `false` then the result is only correct if the graph does not contain loops. This function does not check if loops are actually present.

Returns:

Error code.

Time complexity: O(1).

`igraph_mean_degree` — The mean degree of a graph.

```
igraph_error_t igraph_mean_degree(const igraph_t *graph, igraph_real_t *res,
                                  igraph_bool_t loops);
```

This is a convenience function that computes the average of all vertex degrees. In directed graphs, the average of out-degrees and in-degrees is the same; this is the number that is returned. For the null graph, which has no vertices, NaN is returned.

Arguments:

graph: The input graph object.

res: Pointer to a real number, the result will be stored here.

loops: Whether to consider self-loops during the calculation.

Returns:

Error code.

Time complexity: O(1) if self-loops are considered, O(|E|) where |E| is the number of edges if self-loops are ignored.

igraph_reciprocity — Calculates the reciprocity of a directed graph.

```
igraph_error_t igraph_reciprocity(const igraph_t *graph, igraph_real_t *res,
                      igraph_bool_t ignore_loops,
                      igraph_reciprocity_t mode);
```

In a directed graph, the measure of reciprocity defines the proportion of mutual connections. It is most commonly defined as the probability that the opposite counterpart of a randomly chosen directed edge is also included in the graph. In adjacency matrix notation: `1 - (sum_ij |A_ij - A_ji|) / (2 sum_ij A_ij)`. In multigraphs, each parallel edge between two vertices must have its own separate reciprocal edge, in accordance with the above formula. This measure is calculated if the *mode* argument is `IGRAPH_RECIPROCITY_DEFAULT`.

For directed graphs with no edges, NaN is returned. For undirected graphs, 1 is returned unconditionally.

Prior to igraph version 0.6, another measure was implemented, defined as the probability of having mutual connections 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) disconnected, (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 `IGRAPH_RECIPROCITY_RATIO`.

Arguments:

graph: The graph object.

res: Pointer to an `igraph_real_t` which will contain the result.

ignore_loops: Whether to ignore self-loops when counting edges. Self-loops are considered as a mutual connection.

mode: Type of reciprocity to calculate, possible values are `IGRAPH_RECIPROCITY_DEFAULT` and `IGRAPH_RECIPROCITY_RATIO`, please see their description above.

Returns:

Error code: `IGRAPH_EINVAL`: graph has no edges `IGRAPH_ENOMEM`: not enough memory for temporary data.

Time complexity: O(|V|+|E|), |V| is the number of vertices, |E| is the number of edges.

Example 17.46. File `examples/simple/igraph_reciprocity.c`

igraph_diversity — Structural diversity index of the vertices.

```
igraph_error_t igraph_diversity(const igraph_t *graph, const igraph_vector_t *w
                    igraph_vector_t *res, const igraph_vs_t vids);
```

This measure was defined in Nathan Eagle, Michael Macy and Rob Claxton: Network Diversity and Economic Development, Science 328, 1029--1031, 2010.

It is simply the (normalized) Shannon entropy of the incident edges' weights. `D(i) = H(i) / log(k[i])`, and `H(i) = -sum(p[i,j] log(p[i,j]), j=1..k[i])`, where `p[i,j] = w[i,j] / sum(w[i,l], l=1..k[i])`, `k[i]` is the (total) degree of vertex `i`, and `w[i,j]` is the weight of the edge(s) between vertex `i` and `j`. The diversity of isolated vertices will be NaN (not-a-number), while that of vertices with a single connection will be zero.

The measure works only if the graph is undirected and has no multiple edges. If the graph has multiple edges, simplify it first using `igraph_simplify()`. If the graph is directed, convert it into an undirected graph with `igraph_to_undirected()`.

Arguments:

graph: The undirected input graph.

weights: The edge weights, in the order of the edge IDs, must have appropriate length. Weights must be non-negative.

res: An initialized vector, the results are stored here.

vids: Vertex selector that specifies the vertices which to calculate the measure.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear.

`igraph_is_mutual` — Check whether some edges of a directed graph are mutual.

```
igraph_error_t igraph_is_mutual(const igraph_t *graph, igraph_vector_bool_t *res
                                igraph_es_t es, igraph_bool_t loops);
```

An (A,B) non-loop directed edge is mutual if the graph contains the (B,A) edge too. Whether directed self-loops are considered mutual is controlled by the *loops* parameter.

An undirected graph only has mutual edges, by definition.

Edge multiplicity is not considered here, e.g. if there are two (A,B) edges and one (B,A) edge, then all three are considered to be mutual.

Arguments:

graph: The input graph.

res: Pointer to an initialized vector, the result is stored here.

es: The sequence of edges to check. Supply `igraph_ess_all()` to check all edges.

loops: Boolean, whether to consider directed self-loops to be mutual.

Returns:

Error code.

Time complexity: O(n log(d)), n is the number of edges supplied, d is the maximum in-degree of the vertices that are targets of the supplied edges. An upper limit of the time complexity is O(n log(|E|)), |E| is the number of edges in the graph.

`igraph_has_mutual` — Check whether a directed graph has any mutual edges.

```
igraph_error_t igraph_has_mutual(const igraph_t *graph, igraph_bool_t *res,
                                 igraph_bool_t loops);
```

An (A,B) non-loop directed edge is mutual if the graph contains the (B,A) edge too. Whether directed self-loops are considered mutual is controlled by the *`loops`* parameter.

In undirected graphs, all edges are considered mutual by definition. Thus for undirected graph, this function returns false only when there are no edges.

To check whether a graph is an oriented graph, use this function in conjunction with `igraph_is_directed()`.

Arguments:

`graph`: The input graph.

`res`: Pointer to a boolean, the result will be stored here.

`loops`: Boolean, whether to consider directed self-loops to be mutual.

Returns:

Error code.

Time complexity: O(|E| log(d)) where d is the maximum in-degree.

`igraph_get_adjacency` — The adjacency matrix of a graph.

```
igraph_error_t igraph_get_adjacency(
    const igraph_t *graph, igraph_matrix_t *res, igraph_get_adjacency_t type,
    const igraph_vector_t *weights, igraph_loops_t loops
);
```

The result is an adjacency matrix. Entry i, j of the matrix contains the number of edges connecting vertex i to vertex j in the unweighted case, or the total weight of edges connecting vertex i to vertex j in the weighted case.

Arguments:

`graph`: Pointer to the graph to convert

`res`: Pointer to an initialized matrix object, it will be resized if needed.

`type`: Constant specifying the type of the adjacency matrix to create for undirected graphs. It is ignored for directed graphs. Possible values:

`IGRAPH_GET_ADJACENCY_UPPER` the upper right triangle of the matrix is used.

IGRAPH_GET_ADJACENCY_LOWER	the lower left triangle of the matrix is used.
IGRAPH_GET_ADJACENCY_BOTH	the whole matrix is used, a symmetric matrix is returned if the graph is undirected.

weights: An optional vector containing the weight of each edge in the graph. Supply a null pointer here to make all edges have the same weight of 1.

loops: Constant specifying how loop edges should be handled. Possible values:

IGRAPH_NO_LOOPS	loop edges are ignored and the diagonal of the matrix will contain zeros only
IGRAPH_LOOPS_ONCE	loop edges are counted once, i.e. a vertex with a single unweighted loop edge will have 1 in the corresponding diagonal entry
IGRAPH_LOOPS_TWICE	loop edges are counted twice in *undirected* graphs, i.e. a vertex with a single unweighted loop edge in an undirected graph will have 2 in the corresponding diagonal entry. Loop edges in directed graphs are still counted as 1. Essentially, this means that the function is counting the incident edge *stems* , which makes more sense when using the adjacency matrix in linear algebra.

Returns:

Error code: IGRAPH_EINVAL invalid type argument.

See also:

igraph_get_adjacency_sparse() if you want a sparse matrix representation

Time complexity: O(|V||V|), |V| is the number of vertices in the graph.

igraph_get_adjacency_sparse — Returns the adjacency matrix of a graph in a sparse matrix format.

```
igraph_error_t igraph_get_adjacency_sparse(
    const igraph_t *graph, igraph_sparsemat_t *res, igraph_get_adjacency_t type,
    const igraph_vector_t *weights, igraph_loops_t loops
);
```

Arguments:

graph: The input graph.

res: Pointer to an *initialized* sparse matrix. The result will be stored here. The matrix will be resized as needed.

type: Constant specifying the type of the adjacency matrix to create for undirected graphs. It is ignored for directed graphs. Possible values:

IGRAPH_GET_ADJACENCY_UPPER	the upper right triangle of the matrix is used.

`IGRAPH_GET_ADJACENCY_LOWER`	the lower left triangle of the matrix is used.
`IGRAPH_GET_ADJACENCY_BOTH`	the whole matrix is used, a symmetric matrix is returned if the graph is undirected.

`weights`: An optional vector containing the weight of each edge in the graph. Supply a null pointer here to make all edges have the same weight of 1.

`loops`: Constant specifying how loop edges should be handled. Possible values:

`IGRAPH_NO_LOOPS`	loop edges are ignored and the diagonal of the matrix will contain zeros only
`IGRAPH_LOOPS_ONCE`	loop edges are counted once, i.e. a vertex with a single unweighted loop edge will have 1 in the corresponding diagonal entry
`IGRAPH_LOOPS_TWICE`	loop edges are counted twice in *undirected* graphs, i.e. a vertex with a single unweighted loop edge in an undirected graph will have 2 in the corresponding diagonal entry. Loop edges in directed graphs are still counted as 1. Essentially, this means that the function is counting the incident edge *stems* , which makes more sense when using the adjacency matrix in linear algebra.

Returns:

Error code: `IGRAPH_EINVAL` invalid type argument.

See also:

`igraph_get_adjacency()`, the dense version of this function.

Time complexity: TODO.

`igraph_get_stochastic` — Stochastic adjacency matrix of a graph.

```
igraph_error_t igraph_get_stochastic(
    const igraph_t *graph, igraph_matrix_t *res, igraph_bool_t column_wise,
    const igraph_vector_t *weights
);
```

Stochastic matrix of a graph. The stochastic matrix of a graph is its adjacency matrix, normalized row-wise (or column-wise), such that the sum of each row (or column) is one. The row-wise normalized matrix is also called a *right-stochastic* and contains the transition probabilities of a random walk that follows edge directions in a directed graph. The column-wise normalized matrix is called *left-stochastic* and is related to random walks moving against edge directions.

When the out-degree (or in-degree) of a vertex is zero, the corresponding row (or column) of the row-wise (or column-wise) normalized stochastic matrix will be zero.

Arguments:

`graph`: The input graph.

res: Pointer to an initialized matrix, the result is stored here. It will be resized as needed.

column_wise: If `false`, row-wise normalization is used. If `true`, column-wise normalization is used.

weights: An optional vector containing the weight of each edge in the graph. Supply a null pointer here to make all edges have the same weight of 1.

Returns:

Error code.

Time complexity: O(|V|^2), |V| is the number of vertices in the graph.

See also:

`igraph_get_stochastic_sparse()`, the sparse version of this function.

`igraph_get_stochastic_sparse` — The stochastic adjacency matrix of a graph.

```
igraph_error_t igraph_get_stochastic_sparse(
    const igraph_t *graph, igraph_sparsemat_t *res, igraph_bool_t column_wise,
    const igraph_vector_t *weights
);
```

Stochastic matrix of a graph in sparse format. See `igraph_get_stochastic()` for the information on stochastic matrices.

Arguments:

graph: The input graph.

res: Pointer to an *initialized* sparse matrix, the result is stored here. The matrix will be resized as needed.

column_wise: If `false`, row-wise normalization is used. If `true`, column-wise normalization is used.

weights: An optional vector containing the weight of each edge in the graph. Supply a null pointer here to make all edges have the same weight of 1.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number of vertices and edges.

See also:

`igraph_get_stochastic()`, the dense version of this function.

`igraph_get_edgelist` — The list of edges in a graph.

```
igraph_error_t igraph_get_edgelist(const igraph_t *graph, igraph_vector_int_t *
```

The order of the edges is given by the edge IDs.

Arguments:

graph: Pointer to the graph object

res: Pointer to an initialized vector object, it will be resized.

bycol: Boolean constant. If true, the edges will be returned columnwise, e.g. the first edge is `res[0]->res[|E|]`, the second is `res[1]->res[|E|+1]`, etc. Supply false to get the edge list in a format compatible with `igraph_add_edges()`.

Returns:

Error code.

See also:

`igraph_edges()` to return the result only for some edge IDs.

Time complexity: O(|E|), the number of edges in the graph.

Common types and constants

`igraph_loops_t` — How to interpret self-loops in undirected graphs?

```
typedef enum {
    IGRAPH_NO_LOOPS = 0,
    IGRAPH_LOOPS_TWICE = 1,
    IGRAPH_LOOPS_ONCE = 2,
    IGRAPH_LOOPS = IGRAPH_LOOPS_TWICE
} igraph_loops_t;
```

Controls the interpretation of self-loops in undirected graphs, typically in the context of adjacency matrices or degrees.

These constants are also used to improve readability in boolean contexts, with `IGRAPH_NO_LOOPS`, equivalent to `false`, signifying that loops should be ignored and `IGRAPH_LOOPS`, equivalent to `true`, that loops should be considered.

Values:

`IGRAPH_NO_LOOPS`: Self-loops are ignored.

`IGRAPH_LOOPS_TWICE`: Self-loops are considered, and counted twice in undirected graphs. For example, a self-loop contributes two to the degree of a vertex and to diagonal entries of adjacency matrices. This is the standard interpretation in graph theory, thus `IGRAPH_LOOPS` serves as an alias for this option.

`IGRAPH_LOOPS_ONCE`: Self-loops are considered, and counted only once in undirected graphs.

`igraph_neimode_t` — How to interpret edge directions in directed graphs?

```
typedef enum {
    IGRAPH_OUT = 1,
    IGRAPH_IN = 2,
    IGRAPH_ALL = 3
} igraph_neimode_t;
```

These "neighbor mode" constants are typically used to specify the treatment of edge directions in directed graphs, or which vertices to consider as adjacent to (i.e. neighbor of) a vertex. It is typically ignored in undirected graphs.

Values:

`IGRAPH_OUT`: Follow edge directions in directed graphs, or consider out-neighbors of vertices.

`IGRAPH_IN`: Follow edges in the reverse direction in directed graphs, or consider in-neighbors of vertices.

`IGRAPH_ALL`: Ignore edge directions in directed graphs, or consider all neighbours (both out and in-neighbors) of vertices.

Chapter 18. Graph cycles

Finding cycles

`igraph_find_cycle` — Finds a single cycle in the graph.

```
igraph_error_t igraph_find_cycle(const igraph_t *graph,
                                 igraph_vector_int_t *vertices,
                                 igraph_vector_int_t *edges,
                                 igraph_neimode_t mode);
```

This function returns a cycle of the graph, if there is one. If the graph is acyclic, it returns empty vectors.

Arguments:

graph: The input graph.

vertices: Pointer to an integer vector. If a cycle is found, its vertices will be stored here. Otherwise the vector will be empty.

edges: Pointer to an integer vector. If a cycle is found, its edges will be stored here. Otherwise the vector will be empty.

mode: A constant specifying how edge directions are considered in directed graphs. Valid modes are: `IGRAPH_OUT`, follows edge directions; `IGRAPH_IN`, follows the opposite directions; and `IGRAPH_ALL`, ignores edge directions. This argument is ignored for undirected graphs.

Returns:

Error code.

Time complexity: O(|V|+|E|), where |V| and |E| are the number of vertices and edges in the original input graph.

See also:

`igraph_is_acyclic()` to determine if a graph is acyclic, without returning a specific cycle; `igraph_simple_cycles()` to list all cycles in a graph.

`igraph_simple_cycles` — Finds all simple cycles.

```
igraph_error_t igraph_simple_cycles(
        const igraph_t *graph,
        igraph_vector_int_list_t *vertices, igraph_vector_int_list_t *edges,
        igraph_neimode_t mode,
        igraph_int_t min_cycle_length, igraph_int_t max_cycle_length,
        igraph_int_t max_results);
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

This function searches for all simple cycles using Johnson's cycle detection algorithm, and stores them in the provided vector lists. A simple cycle is a cycle (i.e. closed path) without repeated vertices.

Reference:

Johnson DB: Finding all the elementary circuits of a directed graph. SIAM J. Comput. 4(1):77-84. https://doi.org/10.1137/0204007

Arguments:

graph:	The graph to search for cycles in.
vertices:	The vertex IDs of each cycle will be stored here.
edges:	The edge IDs of each cycle will be stored here.
mode:	A constant specifying how edge directions are considered in directed graphs. Valid modes are: `IGRAPH_OUT`, follows edge directions; `IGRAPH_IN`, follows the opposite directions; and `IGRAPH_ALL`, ignores edge directions. This argument is ignored for undirected graphs.
min_cycle_length:	Limit the minimum length of cycles to search for. Pass a negative value to search for all cycles.
max_cycle_length:	Limit the maximum length of cycles to search for. Pass a negative value to search for all cycles.
max_results:	At most this many cycles will be recorded. If negative, or `IGRAPH_UNLIMITED`, no limit is applied.

Returns:

Error code.

See also:

`igraph_simple_cycles_callback()` to call a function for each found cycle; `igraph_find_cycle()` to find a single cycle; `igraph_fundamental_cycles()` and `igraph_minimum_cycle_basis()` to find a cycle basis, a compact representation of the cycle structure of the graph.

`igraph_simple_cycles_callback` — Finds all simple cycles (callback version).

```
igraph_error_t igraph_simple_cycles_callback(
        const igraph_t *graph,
        igraph_neimode_t mode,
        igraph_int_t min_cycle_length,
        igraph_int_t max_cycle_length,
        igraph_cycle_handler_t *callback,
```

```
        void *arg);
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

This function searches for all simple cycles using Johnson's cycle detection algorithm, and calls a function for each. A simple cycle is a cycle (i.e. closed path) without repeated vertices.

Reference:

Johnson DB: Finding all the elementary circuits of a directed graph. SIAM J. Comput. 4(1):77-84. https://doi.org/10.1137/0204007

Arguments:

graph:	The graph to search for
mode:	A constant specifying how edge directions are considered in directed graphs. Valid modes are: `IGRAPH_OUT`, follows edge directions; `IGRAPH_IN`, follows the opposite directions; and `IGRAPH_ALL`, ignores edge directions. This argument is ignored for undirected graphs.
min_cycle_length:	Limit the minimum length of cycles to search for. Pass a negative value to search for all cycles.
max_cycle_length:	Limit the maximum length of cycles to search for. Pass a negative value to search for all cycles.
callback:	A function to call for each cycle that is found. See also `igraph_cycle_handler_t`
arg:	This parameter will be passed to *callback*.

Returns:

Error code.

See also:

`igraph_simple_cycles()` to store the found cycles; `igraph_find_cycle()` to find a single cycle; `igraph_fundamental_cycles()` and igraph_minimum_cycle_basis() to find a cycle basis, a compact representation of the cycle structure of the graph.

`igraph_cycle_handler_t` — Type of cycle handler functions.

```
typedef igraph_error_t igraph_cycle_handler_t(
        const igraph_vector_int_t *vertices,
        const igraph_vector_int_t *edges,
        void *arg);
```

Callback type, called by `igraph_simple_cycles_callback()` when a cycle is found.

Arguments:

vertices: The vertices of the current cycle. Must not be modified.

edges: The edges of the current cycle. Must not be modified.

arg: The extra parameter passed to `igraph_simple_cycles_callback()`

Returns:

Error code; `IGRAPH_SUCCESS` to continue the search or `IGRAPH_STOP` to stop the search without signaling an error.

Acyclic graphs and feedback sets

`igraph_is_dag` — Checks whether a graph is a directed acyclic graph (DAG).

```
igraph_error_t igraph_is_dag(const igraph_t* graph, igraph_bool_t *res);
```

A directed acyclic graph (DAG) is a directed graph with no cycles.

This function returns `false` for undirected graphs.

The return value of this function is cached in the graph itself; calling the function multiple times with no modifications to the graph in between will return a cached value in O(1) time.

Arguments:

graph: The input graph.

res: Pointer to a boolean constant, the result is stored here.

Returns:

Error code.

Time complexity: O(|V|+|E|), where |V| and |E| are the number of vertices and edges in the original input graph.

See also:

`igraph_topological_sorting()` to get a possible topological sorting of a DAG.

`igraph_is_acyclic` — Checks whether a graph is acyclic or not.

```
igraph_error_t igraph_is_acyclic(const igraph_t *graph, igraph_bool_t *res);
```

This function checks whether a graph has any cycles. Edge directions are considered, i.e. in directed graphs, only directed cycles are searched for.

The result of this function is cached in the graph itself; calling the function multiple times with no modifications to the graph in between will return a cached value in O(1) time.

Arguments:

graph: The input graph.

res: Pointer to a boolean constant, the result is stored here.

Returns:

Error code.

See also:

`igraph_find_cycle()` to find a cycle that demonstrates that the graph is not acyclic; `igraph_is_forest()` to look for undirected cycles even in directed graphs; `igraph_is_dag()` to test specifically for directed acyclic graphs.

Time complexity: O(|V|+|E|), where |V| and |E| are the number of vertices and edges in the original input graph.

`igraph_topological_sorting` — Calculate a possible topological sorting of the graph.

```
igraph_error_t igraph_topological_sorting(
        const igraph_t* graph, igraph_vector_int_t *res, igraph_neimode_t mode)
```

A topological sorting of a directed acyclic graph (DAG) is a linear ordering of its vertices where each vertex comes before all nodes to which it has edges. Every DAG has at least one topological sort, and may have many. This function returns one possible topological sort among them. If the graph contains any cycles that are not self-loops, an error is raised.

Arguments:

graph: The input graph.

res: Pointer to a vector, the result will be stored here. It will be resized if needed.

mode: Specifies how to use the direction of the edges. For `IGRAPH_OUT`, the sorting order ensures that each vertex comes before all vertices to which it has edges, so vertices with no incoming edges go first. For `IGRAPH_IN`, it is quite the opposite: each vertex comes before all vertices from which it receives edges. Vertices with no outgoing edges go first.

Returns:

Error code.

Time complexity: O(|V|+|E|), where |V| and |E| are the number of vertices and edges in the original input graph.

See also:

`igraph_is_dag()` if you are only interested in whether a given graph is a DAG or not, or `igraph_feedback_arc_set()` to find a set of edges whose removal makes the graph acyclic.

Example 18.1. File `examples/simple/igraph_topological_sorting.c`

`igraph_feedback_arc_set` — Feedback arc set of a graph using exact or heuristic methods.

```
igraph_error_t igraph_feedback_arc_set(
        const igraph_t *graph,
        igraph_vector_int_t *result,
        const igraph_vector_t *weights,
        igraph_fas_algorithm_t algo);
```

A feedback arc set is a set of edges whose removal makes the graph acyclic. We are usually interested in *minimum* feedback arc sets, i.e. sets of edges whose total weight is the smallest among all the feedback arc sets.

For undirected graphs, the solution is simple: one has to find a maximum weight spanning tree and then remove all the edges not in the spanning tree. For directed graphs, this is an NP-complete problem, and various heuristics are usually used to find an approximate solution to the problem. This function implements both exact methods and heuristics, selectable with the *`algo`* parameter.

References:

Eades P, Lin X and Smyth WF: A fast and effective heuristic for the feedback arc set problem. Information Processing Letters 47(6), pp 319-323 (1993). https://doi.org/10.1016/0020-0190(93)90079-O

Baharev A, Hermann S, Arnold N and Tobias A: An Exact Method for the Minimum Feedback Arc Set Problem. ACM Journal of Experimental Algorithmics 26, 1–28 (2021). https://doi.org/10.1145/3446429.

Arguments:

`graph`: The graph object.

`result`: An initialized vector, the result will be written here.

`weights`: Weight vector or `NULL` if no weights are specified.

`algo`: The algorithm to use to solve the problem if the graph is directed. Possible values:

`IGRAPH_FAS_EXACT_IP` Finds a *minimum* feedback arc set using integer programming (IP), automatically selecting the best method of this type (currently always `IGRAPH_FAS_EXACT_IP_CG`). The complexity is of course at least exponential.

`IGRAPH_FAS_EXACT_IP_CG` This is an integer programming approach based on a minimum set cover formulation and using incremental constraint generation (CG), added in igraph 0.10.14. We minimize `sum_e w_e b_e` subject to the constraints `sum_e c_e b_e >= 1` for all cycles `c`. Here `w_e` is the weight of edge `e`, `b_e` is a binary variable (0 or 1) indicating whether edge `e` is in the feedback set, and `c_e` is a binary coefficient indicating whether edge `e` is in cycle `c`. The constraint expresses the requirement that all cycles must intersect with (be

broken by) the edge set represented by `b`. Since there are a very large number of cycles in the graph, constraints are generated incrementally, iteratively adding some cycles that do not intersect with the current edge set `b`, then solving for `b` again, until finally no unbroken cycles remain. This approach is similar to that described by Baharev et al (though with a simpler cycle generation scheme), and to what is implemented by SageMath's. `feedback_edge_set` function.

`IGRAPH_FAS_EXACT_IP_TI` This is another integer programming approach based on finding a maximum (largest weight) edge set that adheres to a topological order. It uses the common formulation through triangle inequalities (TI), see Section 3.1 of Baharev et al (2021) for an overview. This method was used before igraph 0.10.14, and is typically much slower than `IGRAPH_FAS_EXACT_IP_CG`.

`IGRAPH_FAS_APPROX_EADES` Finds a feedback arc set using the heuristic of Eades, Lin and Smyth (1993). This is guaranteed to be smaller than |E|/2 - |V|/6, and it is linear in the number of edges (i.e. O(|E|)) to compute.

Returns:

Error code: `IGRAPH_EINVAL` if an unknown method was specified or the weight vector is invalid.

Example 18.2. File `examples/simple/igraph_feedback_arc_set.c`

Example 18.3. File `examples/simple/igraph_feedback_arc_set_ip.c`

Time complexity: depends on *`algo`*, see the time complexities there.

`igraph_feedback_vertex_set` — Feedback vertex set of a graph.

```
igraph_error_t igraph_feedback_vertex_set(
    const igraph_t *graph, igraph_vector_int_t *result,
    const igraph_vector_t *vertex_weights, igraph_fvs_algorithm_t algo);
```

A feedback vertex set is a set of vertices whose removal makes the graph acyclic. Finding a *minimum* feedback vertex set is an NP-complete problem, both on directed and undirected graphs.

Arguments:

`graph`: The graph.

`result`: An initialized vector, the result will be written here.

`vertex_weights`: Vertex weight vector or `NULL` if no weights are specified.

`algo`: Algorithm to use. Possible values:

`IGRAPH_FVS_EXACT_IP` Finds a *miniumum* feedback vertex set using integer programming (IP). The complexity is of course at least exponential. Currently this method uses an approach analogous to that of the `IGRAPH_FAS_EXACT_IP_CG` algorithm of `igraph_feedback_arc_set()`.

Returns:

Error code.

Time complexity: depends on *`algo`*, see the time complexities there.

Eulerian cycles and paths

These functions calculate whether an Eulerian path or cycle exists and if so, can find them.

`igraph_is_eulerian` — Checks whether an Eulerian path or cycle exists.

```
igraph_error_t igraph_is_eulerian(const igraph_t *graph, igraph_bool_t *has_pat
```

An Eulerian path traverses each edge of the graph precisely once. A closed Eulerian path is referred to as an Eulerian cycle.

Arguments:

`graph`: The graph object.

`has_path`: Pointer to a Boolean, will be set to true if an Eulerian path exists. Must not be `NULL`.

`has_cycle`: Pointer to a Boolean, will be set to true if an Eulerian cycle exists. Must not be `NULL`.

Returns:

Error code: `IGRAPH_ENOMEM`, not enough memory for temporary data.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

`igraph_eulerian_cycle` — Finds an Eulerian cycle.

```
igraph_error_t igraph_eulerian_cycle(
        const igraph_t *graph, igraph_vector_int_t *edge_res, igraph_vector_int
```

Finds an Eulerian cycle, if it exists. An Eulerian cycle is a closed path that traverses each edge precisely once.

If the graph has no edges, a zero-length cycle is returned.

This function uses Hierholzer's algorithm.

Arguments:

`graph`: The graph object.

edge_res: Pointer to an initialised vector. The indices of edges belonging to the cycle will be stored here. May be NULL if it is not needed by the caller.

vertex_res: Pointer to an initialised vector. The indices of vertices belonging to the cycle will be stored here. The first and last vertex in the vector will be the same. May be NULL if it is not needed by the caller.

Returns:

Error code:

IGRAPH_ENOMEM not enough memory for temporary data.

IGRAPH_ENOSOL graph does not have an Eulerian cycle.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

igraph_eulerian_path — Finds an Eulerian path.

```
igraph_error_t igraph_eulerian_path(
        const igraph_t *graph, igraph_vector_int_t *edge_res, igraph_vector_int
```

Finds an Eulerian path, if it exists. An Eulerian path traverses each edge precisely once.

If the graph has no edges, a zero-length path is returned.

This function uses Hierholzer's algorithm.

Arguments:

graph: The graph object.

edge_res: Pointer to an initialised vector. The indices of edges belonging to the path will be stored here. May be NULL if it is not needed by the caller.

vertex_res: Pointer to an initialised vector. The indices of vertices belonging to the path will be stored here. May be NULL if it is not needed by the caller.

Returns:

Error code:

IGRAPH_ENOMEM not enough memory for temporary data.

IGRAPH_ENOSOL graph does not have an Eulerian path.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

Cycle bases

igraph_fundamental_cycles — Finds a fundamental cycle basis.

```
igraph_error_t igraph_fundamental_cycles(
        const igraph_t *graph, const igraph_vector_t *weights,
```

```
        igraph_vector_int_list_t *result,
        igraph_int_t start_vid, igraph_real_t bfs_cutoff);
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

This function computes a fundamental cycle basis associated with a breadth-first search tree of the graph.

Edge directions are ignored. Multi-edges and self-loops are supported.

Arguments:

graph: The graph object.

weights: Currently unused.

result: An initialized integer vector list. The result will be stored here, each vector containing the edge IDs of a basis element.

start_vid: If negative, a complete fundamental cycle basis is returned. If a vertex ID, the fundamental cycles associated with the BFS tree rooted in that vertex will be returned, only for the weakly connected component containing that vertex.

bfs_cutoff: If negative, a complete cycle basis is returned. Otherwise, only cycles of length `2*bfs_cutoff + 1` or shorter are included. *bfs_cutoff* is used to limit the depth of the BFS tree when searching for cycle edges.

Returns:

Error code.

See also:

`igraph_minimum_cycle_basis()`

Time complexity: O(|V| + |E|).

`igraph_minimum_cycle_basis` — Computes a minimum weight cycle basis.

```
igraph_error_t igraph_minimum_cycle_basis(
        const igraph_t *graph, const igraph_vector_t *weights,
        igraph_vector_int_list_t *result,
        igraph_real_t bfs_cutoff,
        igraph_bool_t complete, igraph_bool_t use_cycle_order);
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

This function computes a minimum weight cycle basis of a graph. Currently, a modified version of Horton's algorithm is used that allows for cutoffs.

Edge directions are ignored. Multi-edges and self-loops are supported.

References:

Horton, J. D. (1987) A polynomial-time algorithm to find the shortest cycle basis of a graph, SIAM Journal on Computing, 16 (2): 358–366. https://doi.org/10.1137%2F0216026

Arguments:

graph: The graph object.

weights: Currently unused.

result: An initialized integer vector list, the elements of the cycle basis will be stored here as vectors of edge IDs.

bfs_cutoff: If negative, an exact minimum cycle basis is returned. Otherwise only those cycles in the result will be part of some minimum cycle basis which are of size `2*bfs_cutoff + 1` or smaller. Cycles longer than this limit may not be of the smallest possible size. *bfs_cutoff* is used to limit the depth of the BFS tree when computing candidate cycles. Specifying a bfs_cutoff can speed up the computation substantially.

complete: Boolean value. Used only when *bfs_cutoff* was given. If true, a complete basis is returned. If false, only cycles not greater than `2*bfs_cutoff + 1` are returned. This may save computation time, however, the result will not span the entire cycle space.

use_cycle_order: If true, each cycle is returned in natural order: the edge IDs will appear ordered along the cycle. This comes at a small performance cost. If false, no guarantees are given about the ordering of edge IDs within cycles. This parameter exists solely to control performance tradeoffs.

Returns:

Error code.

See also:

`igraph_fundamental_cycles()`

Time complexity: TODO.

Chapter 19. Cliques and independent vertex sets

These functions calculate various graph properties related to cliques and independent vertex sets.

Cliques

`igraph_is_complete` — Decides whether the graph is complete.

```
igraph_error_t igraph_is_complete(const igraph_t *graph, igraph_bool_t *res);
```

A graph is considered complete if all pairs of different vertices are adjacent.

The null graph and the singleton graph are considered complete.

Arguments:

`graph`: The graph object to analyze.

`res`: Pointer to a Boolean variable, the result will be stored here.

Returns:

Error code.

Time complexity: O(|V| + |E|) at worst.

`igraph_is_clique` — Does a set of vertices form a clique?

```
igraph_error_t igraph_is_clique(const igraph_t *graph, igraph_vs_t candidate,
                                igraph_bool_t directed, igraph_bool_t *res);
```

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.

Arguments:

`graph`: The input graph.

`candidate`: The vertex set to test for being a clique.

`directed`: Whether to take edge directions into account in directed graphs.

`res`: The result will be stored here.

Returns:

Error code.

See also:

`igraph_is_complete()` to test if a graph is complete; `igraph_is_independent_vertex_set()` to test for independent vertex sets; `igraph_cliques()`, `igraph_maximal_cliques()` and `igraph_largest_cliques()` to find cliques.

Time complexity: O(n^2 log(d)) where n is the number of vertices in the candidate set and d is the typical vertex degree.

`igraph_cliques` — Finds all or some cliques in a graph.

```
igraph_error_t igraph_cliques(
        const igraph_t *graph, igraph_vector_int_list_t *res,
        igraph_int_t min_size, igraph_int_t max_size,
        igraph_int_t max_results);
```

Cliques are fully connected subgraphs of a graph.

If you are only interested in the size of the largest clique in the graph, use `igraph_clique_number()` instead.

The current implementation of this function uses version 1.21 of the Cliquer library by Sampo Niskanen and Patric R. J. Östergård, http://users.aalto.fi/~pat/cliquer.html

Arguments:

`graph`: The input graph.

`res`: Pointer to an initialized list of integer vectors. The cliques will be stored here as vectors of vertex IDs.

`min_size`: Integer specifying the minimum size of the cliques to be returned. If negative or zero, no lower bound will be used.

`max_size`: Integer specifying the maximum size of the cliques to be returned. If negative or zero, no upper bound will be used.

`max_results`: At most this many cliques will be recorded. If negative, or `IGRAPH_UNLIMITED`, no limit is applied.

Returns:

Error code.

See also:

`igraph_largest_cliques()` and `igraph_clique_number()`.

Time complexity: Exponential

Example 19.1. File `examples/simple/igraph_cliques.c`

`igraph_clique_size_hist` — Counts cliques of each size in the graph.

```
igraph_error_t igraph_clique_size_hist(const igraph_t *graph, igraph_vector_t *
                            igraph_int_t min_size, igraph_int_t max_size);
```

Cliques are fully connected subgraphs of a graph.

The current implementation of this function uses version 1.21 of the Cliquer library by Sampo Niskanen and Patric R. J. Östergård, http://users.aalto.fi/~pat/cliquer.html

Arguments:

graph: The input graph.

hist: Pointer to an initialized vector. The result will be stored here. The first element will store the number of size-1 cliques, the second element the number of size-2 cliques, etc. For cliques smaller than *min_size*, zero counts will be returned.

min_size: Integer specifying the minimum size of the cliques to be returned. If negative or zero, no lower bound will be used.

max_size: Integer specifying the maximum size of the cliques to be returned. If negative or zero, no upper bound will be used.

Returns:

Error code.

See also:

`igraph_cliques()` and `igraph_cliques_callback()`

Time complexity: Exponential

`igraph_cliques_callback` — Calls a function for each clique in the graph.

```
igraph_error_t igraph_cliques_callback(const igraph_t *graph,
                            igraph_int_t min_size, igraph_int_t max_size,
                            igraph_clique_handler_t *cliquehandler_fn, void *ar
```

Cliques are fully connected subgraphs of a graph. This function enumerates all cliques within the given size range and calls *cliquehandler_fn* for each of them. The cliques are passed to the callback function as a pointer to an `igraph_vector_int_t`. Destroying and freeing this vector is left up to the user. Use `igraph_vector_int_destroy()` to destroy it first, then free it using `igraph_free()`.

The current implementation of this function uses version 1.21 of the Cliquer library by Sampo Niskanen and Patric R. J. Östergård, http://users.aalto.fi/~pat/cliquer.html

Arguments:

graph: The input graph.

min_size: Integer specifying the minimum size of the cliques to be returned. If negative or zero, no lower bound will be used.

max_size: Integer specifying the maximum size of the cliques to be returned. If negative or zero, no upper bound will be used.

cliquehandler_fn: Callback function to be called for each clique. See also `igraph_clique_handler_t`.

arg: Extra argument to supply to *cliquehandler_fn*.

Returns:

Error code.

See also:

`igraph_cliques()`

Time complexity: Exponential

`igraph_clique_handler_t` — Type of clique handler functions.

```
typedef igraph_error_t igraph_clique_handler_t(const igraph_vector_int_t *cliqu
```

Callback type, called when a clique was found. See the details at the documentation of `igraph_cliques_callback()`.

Arguments:

clique: The current clique. The clique is owned by the clique search routine. You do not need to destroy or free it if you do not want to store it; however, if you want to hold on to it for a longer period of time, you need to make a copy of it on your own and store the copy itself.

arg: This extra argument was passed to `igraph_cliques_callback()` when it was called.

Returns:

Error code; `IGRAPH_SUCCESS` to continue the search or `IGRAPH_STOP` to stop the search without signaling an error.

`igraph_largest_cliques` — Finds the largest clique(s) in a graph.

```
igraph_error_t igraph_largest_cliques(const igraph_t *graph, igraph_vector_int_
```

A clique is largest (quite intuitively) if there is no other clique in the graph which contains more vertices.

Note that this is not necessarily the same as a maximal clique, i.e. the largest cliques are always maximal but a maximal clique is not always largest.

The current implementation of this function searches for maximal cliques using `igraph_maximal_cliques_callback()` and drops those that are not the largest.

The implementation of this function changed between igraph 0.5 and 0.6, so the order of the cliques and the order of vertices within the cliques will almost surely be different between these two versions.

Arguments:

graph: The input graph.

res: Pointer to an initialized list of integer vectors. The cliques will be stored here as vectors of vertex IDs.

Returns:

Error code.

See also:

`igraph_cliques()`, `igraph_maximal_cliques()`

Time complexity: O(3^(|V|/3)) worst case.

`igraph_maximal_cliques` — Finds all maximal cliques in a graph.

```
igraph_error_t igraph_maximal_cliques(
        const igraph_t *graph,
        igraph_vector_int_list_t *res,
        igraph_int_t min_size, igraph_int_t max_size,
        igraph_int_t max_results);
```

This function lists maximal cliques within a size range, ignoring edge directions. A clique is a subset of vertices in which all vertex pairs are connected. A *maximal* clique is a clique which is not a strict subset of any larger clique.

No guarantees are given about the order in which cliques are returned.

The current implementation uses a modified Bron-Kerbosch algorithm due to Eppstein, Löffler and Strash.

Reference:

David Eppstein, Maarten Löffler, Darren Strash: Listing All Maximal Cliques in Sparse Graphs in Near-Optimal Time. Algorithms and Computation, Lecture Notes in Computer Science, volume 6506, pp 403-414 (2010). https://doi.org/10.1007/978-3-642-17517-6_36 https://arxiv.org/abs/1006.5440

Arguments:

graph: The input graph. Edge directions are ignored.

res: Pointer to list of integer vectors. The maximal cliques will be returned here as vectors of vertex IDs. Note that vertices of a clique may be returned in arbitrary order.

min_size: Integer giving the minimum size of the cliques to be returned. If negative or zero, no lower bound will be used.

max_size: Integer giving the maximum size of the cliques to be returned. If negative or zero, no upper bound will be used.

max_results: At most this many cliques will be recorded. If negative, or IGRAPH_UNLIMITED, no limit is applied.

Returns:

Error code.

See also:

igraph_maximal_independent_vertex_sets() to find maximal independent sets, which are cliques of the complement graph; igraph_clique_number() to find the size of the largest clique; igraph_cliques() to find all cliques.

Time complexity: O(d(n-d)3^(d/3)) worst case, d is the degeneracy of the graph, this is typically small for sparse graphs.

Example 19.2. File examples/simple/igraph_maximal_cliques.c

igraph_maximal_cliques_count — Count the number of maximal cliques in a graph.

```
igraph_error_t igraph_maximal_cliques_count(
        const igraph_t *graph,
        igraph_int_t *res,
        igraph_int_t min_size, igraph_int_t max_size);
```

See igraph_maximal_cliques() for details.

Arguments:

graph: The input graph. Edge directions are ignored.

res: Pointer to an igraph_int_t; the number of maximal cliques will be stored here.

min_size: Integer giving the minimum size of the cliques to be returned. If negative or zero, no lower bound will be used.

max_size: Integer giving the maximum size of the cliques to be returned. If negative or zero, no upper bound will be used.

Returns:

Error code.

See also:

igraph_maximal_cliques().

Time complexity: O(d(n-d)3^(d/3)) worst case, d is the degeneracy of the graph, this is typically small for sparse graphs.

Example 19.3. File examples/simple/igraph_maximal_cliques.c

`igraph_maximal_cliques_file` — Find maximal cliques and write them to a file.

```
igraph_error_t igraph_maximal_cliques_file(
        const igraph_t *graph,
        FILE *outfile,
        igraph_int_t min_size, igraph_int_t max_size,
        igraph_int_t max_results);
```

This function enumerates all maximal cliques within a size range and writes them to file. See `igraph_maximal_cliques()` for details

Arguments:

`graph`: The input graph. Edge directions are ignored.

`outfile`: Pointer to the output file, it should be writable.

`min_size`: Integer giving the minimum size of the cliques to be returned. If negative or zero, no lower bound will be used.

`max_size`: Integer giving the maximum size of the cliques to be returned. If negative or zero, no upper bound will be used.

`max_results`: At most this many cliques will be output. If negative, or `IGRAPH_UNLIMITED`, no limit is applied.

Returns:

Error code.

See also:

`igraph_maximal_cliques()`.

Time complexity: O(d(n-d)3^(d/3)) worst case, d is the degeneracy of the graph, this is typically small for sparse graphs.*

`igraph_maximal_cliques_subset` — Maximal cliques for a subset of initial vertices.

```
igraph_error_t igraph_maximal_cliques_subset(
        const igraph_t *graph, const igraph_vector_int_t *subset,
        igraph_vector_int_list_t *res, igraph_int_t *no, FILE *outfile,
        igraph_int_t min_size, igraph_int_t max_size,
        igraph_int_t max_results);
```

This function enumerates all maximal cliques for a subset of initial vertices and writes them to file. See `igraph_maximal_cliques()` for details.

Arguments:

`graph`: The input graph. Edge directions are ignored.

`subset`: Pointer to an `igraph_vector_int_t` containing the subset of initial vertices.

res: Pointer to a list of integer vectors; the cliques will be stored here.

no: Pointer to an `igraph_int_t`; the number of maximal cliques will be stored here.

outfile: Pointer to an output file or `NULL`. When not `NULL`, the file should be writable.

min_size: Integer giving the minimum size of the cliques to be returned. If negative or zero, no lower bound will be used.

max_size: Integer giving the maximum size of the cliques to be returned. If negative or zero, no upper bound will be used.

max_results: At most this many cliques will be recorded. If negative, or `IGRAPH_UNLIMITED`, no limit is applied.

Returns:

Error code.

See also:

`igraph_maximal_cliques()`.

Time complexity: O(d(n-d)3^(d/3)) worst case, d is the degeneracy of the graph, this is typically small for sparse graphs.

`igraph_maximal_cliques_hist` — Counts the number of maximal cliques of each size in a graph.

```
igraph_error_t igraph_maximal_cliques_hist(
        const igraph_t *graph,
        igraph_vector_t *hist,
        igraph_int_t min_size, igraph_int_t max_size);
```

This function counts how many maximal cliques of each size are present in the graph. Maximal cliques of size one are simply isolated vertices.

Arguments:

graph: The input graph. Edge directions are ignored.

hist: Pointer to an initialized vector. The result will be stored here. The first element will store the number of size-1 maximal cliques, the second element the number of size-2 maximal cliques, etc. For cliques smaller than *min_size*, zero counts will be returned.

min_size: Integer giving the minimum size of the cliques to be returned. If negative or zero, no lower bound will be used.

max_size: Integer giving the maximum size of the cliques to be returned. If negative or zero, no upper bound will be used.

Returns:

Error code.

See also:

`igraph_maximal_cliques()`, `igraph_clique_size_hist()`.

Time complexity: O(d(n-d)3^(d/3)) worst case, d is the degeneracy of the graph, this is typically small for sparse graphs.

`igraph_maximal_cliques_callback` — Finds maximal cliques in a graph and calls a function for each one.

```
igraph_error_t igraph_maximal_cliques_callback(
        const igraph_t *graph,
        igraph_int_t min_size, igraph_int_t max_size,
        igraph_clique_handler_t *cliquehandler_fn, void *arg);
```

This function enumerates all maximal cliques within the given size range and calls *`cliquehandler_fn`* for each of them. The cliques are passed to the callback function as a pointer to an `igraph_vector_int_t`. The vector is owned by the maximal clique search routine so users are expected to make a copy of the vector using `igraph_vector_int_init_copy()` if they want to hold on to it.

Arguments:

`graph`: The input graph. Edge directions are ignored.

`cliquehandler_fn`: Callback function to be called for each clique. See also `igraph_clique_handler_t`.

`arg`: Extra argument to supply to *`cliquehandler_fn`*.

`min_size`: Integer giving the minimum size of the cliques to be returned. If negative or zero, no lower bound will be used.

`max_size`: Integer giving the maximum size of the cliques to be returned. If negative or zero, no upper bound will be used.

Returns:

Error code.

See also:

`igraph_maximal_cliques()`, `igraph_cliques_callback()`.

Time complexity: O(d(n-d)3^(d/3)) worst case, d is the degeneracy of the graph, this is typically small for sparse graphs.

`igraph_clique_number` — Finds the clique number of the graph.

```
igraph_error_t igraph_clique_number(const igraph_t *graph, igraph_int_t *no);
```

The clique number of a graph is the size of the largest clique.

The current implementation of this function searches for maximal cliques using `igraph_maximal_cliques_callback()` and keeps track of the size of the largest clique that was found.

Arguments:

graph: The input graph.

no: The clique number will be returned to the `igraph_int_t` pointed by this variable.

Returns:

Error code.

See also:

`igraph_cliques()`, `igraph_largest_cliques()`.

Time complexity: O(3^(|V|/3)) worst case.

Weighted cliques

`igraph_weighted_cliques` — Finds all cliques in a given weight range in a vertex weighted graph.

```
igraph_error_t igraph_weighted_cliques(
        const igraph_t *graph, const igraph_vector_t *vertex_weights,
        igraph_vector_int_list_t *res,
        igraph_bool_t maximal,
        igraph_real_t min_weight, igraph_real_t max_weight,
        igraph_int_t max_results);
```

Cliques are fully connected subgraphs of a graph. The weight of a clique is the sum of the weights of individual vertices within the clique.

Only positive integer vertex weights are supported.

The current implementation of this function uses version 1.21 of the Cliquer library by Sampo Niskanen and Patric R. J. Östergård, http://users.aalto.fi/~pat/cliquer.html

Arguments:

graph: The input graph.

vertex_weights: A vector of vertex weights. The current implementation will truncate all weights to their integer parts. You may pass `NULL` here to make each vertex have a weight of 1.

res: Pointer to an initialized list of integer vectors. The cliques will be stored here as vectors of vertex IDs.

maximal: If true, only maximal cliques will be returned

min_weight: Integer specifying the minimum weight of the cliques to be returned. If negative or zero, no lower bound will be used.

max_weight: Integer specifying the maximum weight of the cliques to be returned. If negative or zero, no upper bound will be used.

max_results: At most this many cliques will be recorded. If negative, or `IGRAPH_UNLIMITED`, no limit is applied.

Returns:

Error code.

See also:

`igraph_cliques()`, `igraph_maximal_cliques()`

Time complexity: Exponential

`igraph_largest_weighted_cliques` — Finds the largest weight clique(s) in a graph.

```
igraph_error_t igraph_largest_weighted_cliques(const igraph_t *graph,
                                    const igraph_vector_t *vertex_weights, igra
```

The weight of a clique is the sum of the weights of its vertices. This function finds the clique(s) having the largest weight in the graph.

Only positive integer vertex weights are supported.

The current implementation of this function uses version 1.21 of the Cliquer library by Sampo Niskanen and Patric R. J. Östergård, http://users.aalto.fi/~pat/cliquer.html

Arguments:

graph: The input graph.

vertex_weights: A vector of vertex weights. The current implementation will truncate all weights to their integer parts. You may pass `NULL` here to make each vertex have a weight of 1.

res: Pointer to an initialized list of integer vectors. The cliques will be stored here as vectors of vertex IDs.

Returns:

Error code.

See also:

`igraph_weighted_cliques()`, `igraph_weighted_clique_number()`, `igraph_largest_cliques()`

Time complexity: TODO

`igraph_weighted_clique_number` — Finds the weight of the largest weight clique in the graph.

```
igraph_error_t igraph_weighted_clique_number(const igraph_t *graph,
                                     const igraph_vector_t *vertex_weights, igraph
```

The weight of a clique is the sum of the weights of its vertices. This function finds the weight of the largest weight clique.

Only positive integer vertex weights are supported.

The current implementation of this function uses version 1.21 of the Cliquer library by Sampo Niskanen and Patric R. J. Östergård, http://users.aalto.fi/~pat/cliquer.html

Arguments:

graph: The input graph.

vertex_weights: A vector of vertex weights. The current implementation will truncate all weights to their integer parts. You may pass `NULL` here to make each vertex have a weight of 1.

res: The largest weight will be returned to the `igraph_real_t` pointed to by this variable.

Returns:

Error code.

See also:

`igraph_weighted_cliques()`, `igraph_largest_weighted_cliques()`, `igraph_clique_number()`

Time complexity: TODO

Independent vertex sets

`igraph_is_independent_vertex_set` — Does a set of vertices form an independent set?

```
igraph_error_t igraph_is_independent_vertex_set(const igraph_t *graph, igraph_v
                                        igraph_bool_t *res);
```

Tests if no pairs within a set of vertices are adjacenct, i.e. whether they form an independent set. An empty set and singleton set are both considered to be an independent set.

Arguments:

graph: The input graph.

candidate: The vertex set to test for being an independent set.

res: The result will be stored here.

Returns:

Error code.

See also:

`igraph_is_clique()` to test for cliques; `igraph_independent_vertex_sets()`, `igraph_maximal_independent_vertex_sets()` and `igraph_largest_independent_vertex_sets()` to find independent vertex sets.

Time complexity: O(n^2 log(d)) where n is the number of vertices in the candidate set and d is the typical vertex degree.

`igraph_independent_vertex_sets` — Finds all independent vertex sets in a graph.

```
igraph_error_t igraph_independent_vertex_sets(
        const igraph_t *graph,
        igraph_vector_int_list_t *res,
        igraph_int_t min_size, igraph_int_t max_size,
        igraph_int_t max_results);
```

A vertex set is considered independent if there are no edges between them.

If you are interested in the size of the largest independent vertex set, use `igraph_independence_number()` instead.

The current implementation was ported to igraph from the Very Nauty Graph Library by Keith Briggs and uses the algorithm from the paper 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.

Arguments:

graph:	The input graph.
res:	Pointer to an initialized list of integer vectors. The cliques will be stored here as vectors of vertex IDs.
min_size:	Integer specifying the minimum size of the sets to be returned. If negative or zero, no lower bound will be used.
max_size:	Integer specifying the maximum size of the sets to be returned. If negative or zero, no upper bound will be used.
max_results:	At most this many independent vertex sets will be recorded. If negative, or `IGRAPH_UNLIMITED`, no limit is applied.

Returns:

Error code.

See also:

`igraph_largest_independent_vertex_sets()`, `igraph_independence_number()`.

Time complexity: TODO

Example 19.4. File `examples/simple/igraph_independent_sets.c`

igraph_largest_independent_vertex_sets — Finds the largest independent vertex set(s) in a graph.

```
igraph_error_t igraph_largest_independent_vertex_sets(const igraph_t *graph,
        igraph_vector_int_list_t *res);
```

An independent vertex set is largest if there is no other independent vertex set with more vertices in the graph.

The current implementation was ported to igraph from the Very Nauty Graph Library by Keith Briggs and uses the algorithm from the paper 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.

Arguments:

graph: The input graph.

res: Pointer to an initialized list of integer vectors. The cliques will be stored here as vectors of vertex IDs.

Returns:

Error code.

See also:

`igraph_independent_vertex_sets()`, `igraph_maximal_independent_vertex_sets()`.

Time complexity: TODO

igraph_maximal_independent_vertex_sets — Finds all maximal independent vertex sets of a graph.

```
igraph_error_t igraph_maximal_independent_vertex_sets(
        const igraph_t *graph,
        igraph_vector_int_list_t *res,
        igraph_int_t min_size, igraph_int_t max_size,
        igraph_int_t max_results);
```

A maximal independent vertex set is an independent vertex set which can't be extended any more by adding a new vertex to it.

The algorithm used here is based on the following paper: 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.

The implementation was originally written by Kevin O'Neill and modified by K M Briggs in the Very Nauty Graph Library. I simply re-wrote it to use igraph's data structures.

If you are interested in the size of the largest independent vertex set, use `igraph_independence_number()` instead.

Arguments:

graph: The input graph.

res: Pointer to an initialized list of integer vectors. The cliques will be stored here as vectors of vertex IDs.

min_size: Integer specifying the minimum size of the sets to be returned. If negative or zero, no lower bound will be used.

max_size: Integer specifying the maximum size of the sets to be returned. If negative or zero, no upper bound will be used.

max_results: At most this many independent vertex sets will be recorded. If negative, or `IGRAPH_UNLIMITED`, no limit is applied.

Returns:

Error code.

See also:

`igraph_maximal_cliques()`, `igraph_independence_number()`

Time complexity: TODO.

`igraph_independence_number` — Finds the independence number of the graph.

```
igraph_error_t igraph_independence_number(const igraph_t *graph, igraph_int_t *
```

The independence number of a graph is the cardinality of the largest independent vertex set.

The current implementation was ported to igraph from the Very Nauty Graph Library by Keith Briggs and uses the algorithm from the paper 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.

Arguments:

graph: The input graph.

no: The independence number will be returned to the `igraph_int_t` pointed by this variable.

Returns:

Error code.

See also:

`igraph_independent_vertex_sets()`.

Time complexity: TODO.

Chapter 20. Graph motifs, dyad census and triad census

This section deals with functions which find small induced subgraphs in a graph. These were first defined for subgraphs of two and three vertices by Holland and Leinhardt, and named dyad census and triad census.

`igraph_dyad_census` — Dyad census, as defined by Holland and Leinhardt.

```
igraph_error_t igraph_dyad_census(const igraph_t *graph, igraph_real_t *mut,
                      igraph_real_t *asym, igraph_real_t *null);
```

Dyad census means classifying each pair of vertices of a directed graph into three categories: mutual (there is at least one edge from a to b and also from b to a); asymmetric (there is at least one edge either from a to b or from b to a, but not the other way) and null (no edges between a and b in either direction).

Holland, P.W. and Leinhardt, S. (1970). A Method for Detecting Structure in Sociometric Data. American Journal of Sociology, 70, 492-513.

Arguments:

graph: The input graph. For an undirected graph, there are no asymmetric connections.

mut: Pointer to a real, the number of mutual dyads is stored here.

asym: Pointer to a real, the number of asymmetric dyads is stored here.

null: Pointer to a real, the number of null dyads is stored here.

Returns:

Error code.

See also:

`igraph_reciprocity()`, `igraph_triad_census()`.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

`igraph_triad_census` — Triad census, as defined by Davis and Leinhardt.

```
igraph_error_t igraph_triad_census(const igraph_t *graph, igraph_vector_t *res)
```

Calculating the triad census means classifying every triple of vertices in a directed graph based on the type of pairwise connections it contains, i.e. mutual, asymmetric or no connection. A triple can be in one of 16 states, commonly described using Davis and Leinhardt's "MAN labels". The *res* vector will contain the counts of these in the following order:

`0: 003` A, B, C, the empty graph.

`1: 012` A->B, C, a graph with a single directed edge.

`2: 102` A<->B, C, a graph with a mutual connection between two vertices.

`3: 021D` A<-B->C, the binary out-tree.

`4: 021U` A->B<-C, the binary in-tree.

`5: 021C` A->B->C, the directed line.

`6: 111D` A<->B<-C.

`7: 111U` A<->B->C.

`8: 030T` A->B<-C, A->C.

`9: 030C` A<-B<-C, A->C.

`10: 201` A<->B<->C.

`11: 120D` A<-B->C, A<->C.

`12: 120U` A->B<-C, A<->C.

`13: 120C` A->B->C, A<->C.

`14: 210` A->B<->C, A<->C.

`15: 300` A<->B<->C, A<->C, the complete graph.

This function is intended for directed graphs. If the input is undirected, a warning is shown, and undirected edges will be interpreted as mutual.

This function calls `igraph_motifs_randesu()` which is an implementation of the FANMOD motif finder tool, see `igraph_motifs_randesu()` for details. Note that the order of the triads is not the same for `igraph_triad_census()` and `igraph_motifs_randesu()`.

References:

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.

Arguments:

`graph`: The input graph.

`res`: Pointer to an initialized vector, the result is stored here in the same order as given in the list above. Note that this order is different than the one used by `igraph_motifs_randesu()`.

Returns:

Error code.

See also:

`igraph_motifs_randesu()`, `igraph_dyad_census()`.

Time complexity: TODO.

Finding triangles

`igraph_count_adjacent_triangles` — Count the number of triangles a vertex is part of.

```
igraph_error_t igraph_count_adjacent_triangles(const igraph_t *graph,
                              igraph_vector_t *res,
                              const igraph_vs_t vids);
```

Arguments:

graph: The input graph. Edge directions and multiplicities are ignored.

res: Initiliazed vector, the results are stored here.

vids: The vertices to perform the calculation for.

Returns:

Error mode.

See also:

`igraph_list_triangles()` to list triangles, `igraph_count_triangles()` to count all triangles at once.

Time complexity: O(d^2 n), d is the average vertex degree of the queried vertices, n is their number.

`igraph_count_triangles` — Counts triangles in a graph.

```
igraph_error_t igraph_count_triangles(const igraph_t *graph, igraph_real_t *res
```

This function computes the total number of triangles, i.e. fully connected vertex triples, in a graph. Edge directions, edge multiplicities, and self-loops are ignored.

Arguments:

graph: The graph object. Edge directions and multiplicites are ignored.

res: Pointer to a real variable, the result will be stored here.

Returns:

Error code: `IGRAPH_ENOMEM`: not enough memory for temporary data.

See also:

`igraph_list_triangles()`, `igraph_count_adjacent_triangles()`, `igraph_transitivity_undirected()`.

Time complexity: O(|V|*d^2), |V| is the number of vertices in the graph, d is the average node degree.

`igraph_list_triangles` — Find all triangles in a graph.

```
igraph_error_t igraph_list_triangles(const igraph_t *graph,
                           igraph_vector_int_t *res);
```

The triangles are reported as a long list of vertex ID triplets. Use the `int` variant of `igraph_matrix_view_from_vector()` to create a matrix view into the vector where each triangle is stored in a column of the matrix (see the example).

Arguments:

`graph`: The input graph, edge directions are ignored. Multiple edges are ignored.

`res`: Pointer to an initialized integer vector, the result is stored here, in a long list of triples of vertex IDs. Each triple is a triangle in the graph. Each triangle is listed exactly once.

Returns:

Error code.

See also:

`igraph_count_triangles()` to count the triangles, `igraph_count_adjacent_triangles()` to count the triangles a vertex participates in, `igraph_transitivity_undirected()` to compute the global clustering coefficient.

Time complexity: O(d^2 n), d is the average degree, n is the number of vertices.

Example 20.1. File `examples/simple/igraph_list_triangles.c`

Graph motifs

`igraph_motifs_randesu` — Count the number of motifs in a graph.

```
igraph_error_t igraph_motifs_randesu(const igraph_t *graph, igraph_vector_t *his
                           igraph_int_t size, const igraph_vector_t *cut_prob);
```

Motifs are small weakly connected induced subgraphs of a given structure in a graph. It is argued that the motif profile (i.e. the number of different motifs in the graph) is characteristic for different types of networks and network function is related to the motifs in the graph.

This function is able to find directed motifs of sizes three and four and undirected motifs of sizes three to six (i.e. the number of different subgraphs with three to six vertices in the network).

In a big network the total number of motifs can be very large, so it takes a lot of time to find all of them. In this case, a sampling method can be used. This function is capable of doing sampling via the *`cut_prob`* argument. This argument gives the probability that a branch of the motif search tree will not be explored. See S. Wernicke and F. Rasche: FANMOD: a tool for fast network motif detection, Bioinformatics 22(9), 1152--1153, 2006 for details. https://doi.org/10.1093/bioinformatics/btl038

Set the *cut_prob* argument to a zero vector for finding all motifs.

Directed motifs will be counted in directed graphs and undirected motifs in undirected graphs.

Arguments:

graph: The graph to find the motifs in.

hist: The result of the computation, it gives the number of motifs found for each isomorphism class. See `igraph_isoclass()` for help about isomorphism classes. Note that this function does *not* count isomorphism classes that are not connected and will report NaN (more precisely `IGRAPH_NAN`) for them.

size: The size of the motifs to search for. For directed graphs, only 3 and 4 are implemented, for undirected, 3 to 6. The limitation is not in the motif finding code, but the graph isomorphism code.

cut_prob: Vector of probabilities for cutting the search tree at a given level. The first element is the first level, etc. To perform a complete search and find all motifs, supply either an all-zero vector of length *size*, or (since igraph 0.10.14) a `NULL` pointer.

Returns:

Error code.

See also:

`igraph_motifs_randesu_estimate()` for estimating the number of motifs in a graph, this can help to set the *cut_prob* parameter; `igraph_motifs_randesu_no()` to calculate the total number of motifs of a given size in a graph; `igraph_motifs_randesu_callback()` for calling a callback function for every motif found; `igraph_subisomorphic_lad()` for finding subgraphs on more than 4 (directed) or 6 (undirected) vertices; `igraph_graph_count()` to find the number of graph on a given number of vertices, i.e. the length of the *hist* vector.

Time complexity: TODO.

Example 20.2. File `examples/simple/igraph_motifs_randesu.c`

`igraph_motifs_randesu_no` — Count the total number of motifs in a graph.

```
igraph_error_t igraph_motifs_randesu_no(
    const igraph_t *graph, igraph_real_t *no, igraph_int_t size,
    const igraph_vector_t *cut_prob
);
```

This function counts the total number of (weakly) connected induced subgraphs on *size* vertices, without assigning isomorphism classes to them. Arbitrarily large motif sizes are supported.

Arguments:

graph: The graph object to study.

no: Pointer to an `igraph_real_t`, the result will be stored here. Note that even though the result is an integer, we need to use `igraph_real_t` to avoid overflow when igraph is compiled with 32-bit integers.

size: The size of the motifs to count.

cut_prob: Vector of probabilities for cutting the search tree at a given level. The first element is the first level, etc. To perform a complete search and find all connected subgraphs, supply either an all-zero vector of length *size*, or (since igraph 0.10.14) a NULL pointer.

Returns:

Error code.

See also:

igraph_motifs_randesu(), igraph_motifs_randesu_estimate().

Time complexity: TODO.

igraph_motifs_randesu_estimate — Estimate the total number of motifs in a graph.

```
igraph_error_t igraph_motifs_randesu_estimate(const igraph_t *graph, igraph_rea
                                      igraph_int_t size, const igraph_vector_t *cu
                                      igraph_int_t sample_size,
                                      const igraph_vector_int_t *parsample);
```

This function estimates the total number of (weakly) connected induced subgraphs on *size* vertices. For example, an undirected complete graph on n vertices will have one motif of size n, and n motifs of *size* n - 1. As another example, one triangle and a separate vertex will have zero motifs of size four.

This function is useful for large graphs for which it is not feasible to count all connected subgraphs, as there are too many of them.

The estimate is made by taking a sample of vertices and counting all connected subgraphs in which these vertices are included. There is also a *cut_prob* parameter which gives the probabilities to cut a branch of the search tree.

Arguments:

graph: The graph object to study.

est: Pointer to an igraph_real_t, the result will be stored here. Note that even though the result is an integer, we need to use igraph_real_t to avoid overflow when igraph is compiled with 32-bit integers.

size: The size of the subgraphs to look for.

cut_prob: Vector of probabilities for cutting the search tree at a given level. The first element is the first level, etc. To perform a complete search and find all motifs, supply either an all-zero vector of length *size*, or (since igraph 0.10.14) a NULL pointer.

sample_size: The number of vertices to use as the sample. This parameter is only used if the *parsample* argument is a null pointer.

parsample: Either pointer to an initialized vector or a null pointer. If a vector then the vertex IDs in the vector are used as a sample. If a null pointer then the *sample_size* argument is used to create a sample of vertices drawn with uniform probability.

Returns:

Error code.

See also:

`igraph_motifs_randesu()`, `igraph_motifs_randesu_no()`.

Time complexity: TODO.

`igraph_motifs_randesu_callback` — Finds motifs in a graph and calls a function for each of them.

```
igraph_error_t igraph_motifs_randesu_callback(
        const igraph_t *graph,
        igraph_int_t size, const igraph_vector_t *cut_prob,
        igraph_motifs_handler_t *callback, void* extra);
```

Similarly to `igraph_motifs_randesu()`, this function is able to find directed motifs of sizes three and four and undirected motifs of sizes three to six (i.e. the number of different subgraphs with three to six vertices in the network). However, instead of counting them, the function will call a callback function for each motif found to allow further tests or post-processing.

The *`cut_prob`* argument also allows sampling the motifs, just like for `igraph_motifs_randesu()`. Set the *`cut_prob`* argument to a zero vector for finding all motifs.

Arguments:

`graph`: The graph to find the motifs in.

`size`: The size of the motifs to search for. Only three and four are implemented currently. The limitation is not in the motif finding code, but the graph isomorphism code.

`cut_prob`: Vector of probabilities for cutting the search tree at a given level. The first element is the first level, etc. To perform a complete search and find all motifs, supply either an all-zero vector of length *`size`*, or (since igraph 0.10.14) a `NULL` pointer.

`callback`: A pointer to a function of type `igraph_motifs_handler_t`. This function will be called whenever a new motif is found.

`extra`: Extra argument to pass to the callback function.

Returns:

Error code.

Time complexity: TODO.

Example 20.3. File `examples/simple/igraph_motifs_randesu.c`

`igraph_motifs_handler_t` — Callback type for `igraph_motifs_randesu_callback`.

```
typedef igraph_error_t igraph_motifs_handler_t(const igraph_t *graph,
        const igraph_vector_int_t *vids,
        igraph_int_t isoclass,
        void *extra);
```

`igraph_motifs_randesu_callback()` calls a specified callback function whenever a new motif is found during a motif search. This callback function must be of type `igraph_motifs_handler_t`. It has the following arguments:

Arguments:

graph: The graph that that algorithm is working on. Of course this must not be modified.

vids: The IDs of the vertices in the motif that has just been found. This vector is owned by the motif search algorithm, so do not modify or destroy it; make a copy of it if you need it later.

isoclass: The isomorphism class of the motif that has just been found. Use `igraph_graph_count()` to find the maximum possible isoclass for graphs of a given size. See `igraph_isoclass` and `igraph_isoclass_subgraph` for more information.

extra: The extra argument that was passed to `igraph_motifs_randesu_callback()`.

Returns:

`IGRAPH_SUCCESS` to continue the motif search, `IGRAPH_STOP` to stop the motif search and return to the caller normally. Any other return value is interpreted as an igraph error code, which will terminate the search and return the same error code to the caller.

See also:

`igraph_motifs_randesu_callback()`

Chapter 21. Graph isomorphism

The simple interface

igraph provides four set of functions to deal with graph isomorphism problems.

The `igraph_isomorphic()` and `igraph_subisomorphic()` functions make up the first set (in addition with the `igraph_permute_vertices()` function). These functions choose the algorithm which is best for the supplied input graph. (The choice is not very sophisticated though, see their documentation for details.)

The VF2 graph (and subgraph) isomorphism algorithm is implemented in igraph, these functions are the second set. See `igraph_isomorphic_vf2()` and `igraph_subisomorphic_vf2()` for starters.

Functions for the Bliss algorithm constitute the third set, see `igraph_isomorphic_bliss()`.

Finally, the isomorphism classes of all directed graphs with three and four vertices and all undirected graphs with 3-6 vertices are precomputed and stored in igraph, so for these small graphs there is a separate fast path in the code that does not use more complex, generic isomorphism algorithms.

`igraph_isomorphic` — Are two graphs isomorphic?

```
igraph_error_t igraph_isomorphic(const igraph_t *graph1, const igraph_t *graph2
                      igraph_bool_t *iso);
```

In simple terms, two graphs are isomorphic if they become indistinguishable from each other once their vertex labels are removed (rendering the vertices within each graph indistiguishable). More precisely, two graphs are isomorphic if there is a one-to-one mapping from the vertices of the first one to the vertices of the second such that it transforms the edge set of the first graph into the edge set of the second. This mapping is called an *isomorphism*.

This function decides which graph isomorphism algorithm to be used based on the input graphs. Right now it does the following:

1. If one graph is directed and the other undirected then an error is triggered.
2. If one of the graphs has multi-edges then both graphs are simplified and colorized using `igraph_simplify_and_colorize()` and sent to VF2.
3. If the two graphs does not have the same number of vertices and edges it returns with `false`.
4. Otherwise, if the `igraph_isoclass()` function supports both graphs (which is true for directed graphs with 3 and 4 vertices, and undirected graphs with 3-6 vertices), an O(1) algorithm is used with precomputed data.
5. Otherwise Bliss is used, see `igraph_isomorphic_bliss()`.

Please call the VF2 and Bliss functions directly if you need something more sophisticated, e.g. you need the isomorphic mapping.

Arguments:

graph1: The first graph.

graph2: The second graph.

iso: Pointer to a Boolean variable, will be set to `true` if the two graphs are isomorphic, and `false` otherwise.

Returns:

Error code.

See also:

`igraph_isoclass()`, `igraph_isoclass_subgraph()`, `igraph_isoclass_create()`.

Time complexity: exponential.

`igraph_subisomorphic` — Decide subgraph isomorphism.

```
igraph_error_t igraph_subisomorphic(const igraph_t *graph1, const igraph_t *gra
                          igraph_bool_t *iso);
```

Check whether *graph2* is isomorphic to a subgraph of *graph1*. Currently this function just calls `igraph_subisomorphic_vf2()` for all graphs.

Currently this function does not support non-simple graphs.

Arguments:

graph1: The first input graph, may be directed or undirected. This is supposed to be the bigger graph.

graph2: The second input graph, it must have the same directedness as *graph2*, or an error is triggered. This is supposed to be the smaller graph.

iso: Pointer to a boolean, the result is stored here.

Returns:

Error code.

Time complexity: exponential.

`igraph_count_automorphisms` — Number of automorphisms of a graph.

```
igraph_error_t igraph_count_automorphisms(
    const igraph_t *graph, const igraph_vector_int_t *colors,
    igraph_real_t *result
);
```

This function computes the number of automorphisms of a graph. Since the number of automorphisms may be very large, the result is returned as an `igraph_real_t` instead of an integer. If the number of automorphisms is larger than what can be represented in an `igraph_real_t` and you need the exact number, use `igraph_count_automorphisms_bliss()`, which can return the number as a string.

Arguments:

graph: The input graph. Multiple edges between the same nodes are not supported and will cause an incorrect result to be returned.

colors: An optional vertex color vector for the graph. Supply a null pointer is the graph is not colored.

result: Pointer to an `igraph_real_t`, the number of automorphisms will be returned here.

Returns:

Error code. `IGRAPH_EOVERFLOW` if the number of automorphisms is too large to be represented in an `igraph_real_t` .

Time complexity: exponential, in practice it is fast for many graphs.

`igraph_automorphism_group` — Automorphism group generators of a graph.

```
igraph_error_t igraph_automorphism_group(
    const igraph_t *graph, const igraph_vector_int_t *colors,
    igraph_vector_int_list_t *generators
);
```

This function computes the generators of the automorphism group of a graph. The generator set may not be minimal and may depend on the specific parameters of the algorithm under the hood. The generators are permutations represented using zero-based indexing.

The current implementation uses BLISS behind the scenes and the result may be dependent on the splitting heuristics. Use `igraph_automorphism_group_bliss()` if you want to fine-tune the splitting heuristics.

Arguments:

graph: The input graph. Multiple edges between the same nodes are not supported and will cause an incorrect result to be returned.

colors: An optional vertex color vector for the graph. Supply a null pointer is the graph is not colored.

generators: Must be an initialized interger vector list. The generators of the automorphism group will be stored here.

Returns:

Error code.

Time complexity: exponential, in practice it is fast for many graphs.

`igraph_canonical_permutation` — Canonical permutation of a graph.

```
igraph_error_t igraph_canonical_permutation(
    const igraph_t *graph, const igraph_vector_int_t *colors,
```

```
    igraph_vector_int_t *labeling
);
```

This function computes the vertex permutation which transforms the graph into a canonical form. Two graphs have the same canonical form if and only if they are isomorphic. Use `igraph_is_same_graph()` to compare two canonical forms.

The current implementation uses the BLISS isomorphism algorithms with sensible defaults. Use `igraph_canonical_permutation_bliss()` to fine-tune the parameters.

Arguments:

`graph`: The input graph. Multiple edges between the same nodes are not supported and will cause an incorrect result to be returned.

`colors`: An optional vertex color vector for the graph. Supply a null pointer is the graph is not colored.

`labeling`: Pointer to a vector, the result is stored here. The permutation takes vertex 0 to the first element of the vector, vertex 1 to the second, etc. The vector will be resized as needed.

Returns:

Error code.

See also:

`igraph_is_same_graph()`

Time complexity: exponential, in practice it is fast for many graphs.

The BLISS algorithm

Bliss is a successor of the famous NAUTY algorithm and implementation. While using the same ideas in general, with better heuristics and data structures Bliss outperforms NAUTY on most graphs.

Bliss was developed and implemented by Tommi Junttila and Petteri Kaski at Helsinki University of Technology, Finland. For more information, see the Bliss homepage at https://users.aalto.fi/~tjunttil/bliss/ and the following publication:

Tommi Junttila and Petteri Kaski: "Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs" In ALENEX 2007, pages 135–149, 2007 https://doi.org/10.1137/1.9781611972870.13

Tommi Junttila and Petteri Kaski: "Conflict Propagation and Component Recursion for Canonical Labeling" in TAPAS 2011, pages 151–162, 2011. https://doi.org/10.1007/978-3-642-19754-3_16

Bliss works with both directed graphs and undirected graphs. It supports graphs with self-loops, but not graphs with multi-edges.

Bliss version 0.75 is included in igraph.

`igraph_bliss_sh_t` — Splitting heuristics for Bliss.

```
typedef enum { IGRAPH_BLISS_F = 0, IGRAPH_BLISS_FL,
               IGRAPH_BLISS_FS, IGRAPH_BLISS_FM,
               IGRAPH_BLISS_FLM, IGRAPH_BLISS_FSM
```

```
} igraph_bliss_sh_t;
```

`IGRAPH_BLISS_FL` provides good performance for many graphs, and is a reasonable default choice. `IGRAPH_BLISS_FSM` is recommended for graphs that have some combinatorial structure, and is the default of the Bliss library's command line tool.

Values:

`IGRAPH_BLISS_F`: First non-singleton cell.

`IGRAPH_BLISS_FL`: First largest non-singleton cell.

`IGRAPH_BLISS_FS`: First smallest non-singleton cell.

`IGRAPH_BLISS_FM`: First maximally non-trivially connected non-singleton cell.

`IGRAPH_BLISS_FLM`: Largest maximally non-trivially connected non-singleton cell.

`IGRAPH_BLISS_FSM`: Smallest maximally non-trivially connected non-singleton cell.

`igraph_bliss_info_t` — Information about a Bliss run.

```
typedef struct igraph_bliss_info_t {
    unsigned long nof_nodes;
    unsigned long nof_leaf_nodes;
    unsigned long nof_bad_nodes;
    unsigned long nof_canupdates;
    unsigned long nof_generators;
    unsigned long max_level;
    char *group_size;
} igraph_bliss_info_t;
```

Some secondary information found by the Bliss algorithm is stored here. It is useful if you wany to study the internal working of the algorithm.

Values:

`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.

`nof_generators`: Number of generators of the automorphism group.

`max_level`: Maximum level.

`group_size`: The size of the automorphism group of the graph, given as a string. It should be deallocated via `igraph_free()` if not needed any more.

See https://users.aalto.fi/~tjunttil/bliss/ for details about the algorithm and these parameters.

`igraph_isomorphic_bliss` — Graph isomorphism via Bliss.

```
igraph_error_t igraph_isomorphic_bliss(const igraph_t *graph1, const igraph_t *
                            const igraph_vector_int_t *colors1, const igraph_ve
                            igraph_bool_t *iso, igraph_vector_int_t *map12,
                            igraph_vector_int_t *map21, igraph_bliss_sh_t sh,
                            igraph_bliss_info_t *info1, igraph_bliss_info_t *in
```

This function uses the Bliss graph isomorphism algorithm, a successor of the famous NAUTY algorithm and implementation. Bliss is open source and licensed according to the GNU LGPL. See https://users.aalto.fi/~tjunttil/bliss/ for details. Currently the 0.75 version of Bliss is included in igraph.

Isomorphism testing is implemented by producing the canonical form of both graphs using `igraph_canonical_permutation_bliss()` and comparing them.

Arguments:

graph1: The first input graph. Multiple edges between the same nodes are not supported and will cause an incorrect result to be returned.

graph2: The second input graph. Multiple edges between the same nodes are not supported and will cause an incorrect result to be returned.

colors1: An optional vertex color vector for the first graph. Supply a null pointer if your graph is not colored.

colors2: An optional vertex color vector for the second graph. Supply a null pointer if your graph is not colored.

iso: Pointer to a boolean, the result is stored here.

map12: A vector or `NULL` pointer. If not `NULL` then an isomorphic mapping from *graph1* to *graph2* is stored here. If the input graphs are not isomorphic then this vector is cleared, i.e. it will have length zero.

map21: Similar to *map12*, but for the mapping from *graph2* to *graph1*.

sh: Splitting heuristics to be used for the graphs. See `igraph_bliss_sh_t`.

info1: If not `NULL`, information about the canonization of the first input graph is stored here. Note that if the two graphs have different number of vertices or edges, then this is only partially filled. The memory used by this structure should be released when no longer needed, see `igraph_bliss_info_t` for details.

info2: Same as *info1*, but for the second graph.

Returns:

Error code.

Time complexity: exponential, but in practice it is quite fast.

`igraph_count_automorphisms_bliss` — Number of automorphisms using Bliss.

```
igraph_error_t igraph_count_automorphisms_bliss(
    const igraph_t *graph, const igraph_vector_int_t *colors,
    igraph_bliss_sh_t sh, igraph_bliss_info_t *info
```

```
);
```

The number of automorphisms of a graph is computed using Bliss. The result is returned as part of the *info* structure, in tag `group_size`. It is returned as a string, as it can be very high even for relatively small graphs. See also `igraph_bliss_info_t`.

Arguments:

graph: The input graph. Multiple edges between the same nodes are not supported and will cause an incorrect result to be returned.

colors: An optional vertex color vector for the graph. Supply a null pointer is the graph is not colored.

sh: The splitting heuristics to be used in Bliss. See `igraph_bliss_sh_t`.

info: The result is stored here, in particular in the `group_size` tag of *info*. The memory used by this structure must be released when no longer needed, see `igraph_bliss_info_t`.

Returns:

Error code.

Time complexity: exponential, in practice it is fast for many graphs.

`igraph_automorphism_group_bliss` — Automorphism group generators using Bliss.

```
igraph_error_t igraph_automorphism_group_bliss(
    const igraph_t *graph, const igraph_vector_int_t *colors,
    igraph_vector_int_list_t *generators, igraph_bliss_sh_t sh,
    igraph_bliss_info_t *info
);
```

The generators of the automorphism group of a graph are computed using Bliss. The generator set may not be minimal and may depend on the splitting heuristics. The generators are permutations represented using zero-based indexing.

Arguments:

graph: The input graph. Multiple edges between the same nodes are not supported and will cause an incorrect result to be returned.

colors: An optional vertex color vector for the graph. Supply a null pointer is the graph is not colored.

generators: Must be an initialized interger vector list. The generators of the automorphism group will be stored here.

sh: The splitting heuristics to be used in Bliss. See `igraph_bliss_sh_t`.

info: If not `NULL` then information on Bliss internals is stored here. The memory used by this structure must to be freed when no longer needed, see `igraph_bliss_info_t`.

Returns:

Error code.

Time complexity: exponential, in practice it is fast for many graphs.

`igraph_canonical_permutation_bliss` — Canonical permutation using Bliss.

```
igraph_error_t igraph_canonical_permutation_bliss(
    const igraph_t *graph, const igraph_vector_int_t *colors,
    igraph_vector_int_t *labeling, igraph_bliss_sh_t sh,
    igraph_bliss_info_t *info
);
```

This function computes the vertex permutation which transforms the graph into a canonical form, using the Bliss algorithm. Two graphs have the same canonical form if and only if they are isomorphic. Use `igraph_is_same_graph()` to compare two canonical forms.

Arguments:

graph: The input graph. Multiple edges between the same nodes are not supported and will cause an incorrect result to be returned.

colors: An optional vertex color vector for the graph. Supply a null pointer is the graph is not colored.

labeling: Pointer to a vector, the result is stored here. The permutation takes vertex 0 to the first element of the vector, vertex 1 to the second, etc. The vector will be resized as needed.

sh: The splitting heuristics to be used in Bliss. See `igraph_bliss_sh_t`.

info: If not `NULL` then information on Bliss internals is stored here. The memory used by this structure must to be freed when no longer needed, see `igraph_bliss_info_t`.

Returns:

Error code.

See also:

`igraph_is_same_graph()`

Time complexity: exponential, in practice it is fast for many graphs.

The VF2 algorithm

The VF2 algorithm can search for a subgraph in a larger graph, or check if two graphs are isomorphic. See P. Foggia, C. Sansone, M. Vento, An Improved algorithm for matching large graphs, Proc. of the 3rd IAPR-TC-15 International Workshop on Graph-based Representations, Italy, 2001.

VF2 supports both vertex and edge-colored graphs, as well as custom vertex or edge compatibility functions.

VF2 works with both directed and undirected graphs. Only simple graphs are supported. Self-loops or multi-edges must not be present in the graphs. Currently, the VF2 functions do not check that the input graph is simple: it is the responsibility of the user to pass in valid input.

igraph_isomorphic_vf2 — Isomorphism via VF2.

```
igraph_error_t igraph_isomorphic_vf2(const igraph_t *graph1, const igraph_t *gra
                          const igraph_vector_int_t *vertex_color1,
                          const igraph_vector_int_t *vertex_color2,
                          const igraph_vector_int_t *edge_color1,
                          const igraph_vector_int_t *edge_color2,
                          igraph_bool_t *iso, igraph_vector_int_t *map12,
                          igraph_vector_int_t *map21,
                          igraph_isocompat_t *node_compat_fn,
                          igraph_isocompat_t *edge_compat_fn,
                          void *arg);
```

This function performs the VF2 algorithm via calling `igraph_get_isomorphisms_vf2_callback()`.

Note that this function cannot be used for deciding subgraph isomorphism, use `igraph_subisomorphic_vf2()` for that.

Arguments:

graph1: The first graph, may be directed or undirected.

graph2: The second graph. It must have the same directedness as *graph1*, otherwise an error is reported.

vertex_color1: An optional color vector for the first graph. If color vectors are given for both graphs, then the isomorphism is calculated on the colored graphs; i.e. two vertices can match only if their color also matches. Supply a null pointer here if your graphs are not colored.

vertex_color2: An optional color vector for the second graph. See the previous argument for explanation.

edge_color1: An optional edge color vector for the first graph. The matching edges in the two graphs must have matching colors as well. Supply a null pointer here if your graphs are not edge-colored.

edge_color2: The edge color vector for the second graph.

iso: Pointer to a Boolean constant, the result of the algorithm will be placed here.

map12: Pointer to an initialized vector or a NULL pointer. If not a NULL pointer then the mapping from *graph1* to *graph2* is stored here. If the graphs are not isomorphic then the vector is cleared (i.e. has zero elements).

map21: Pointer to an initialized vector or a NULL pointer. If not a NULL pointer then the mapping from *graph2* to *graph1* is stored here. If the graphs are not isomorphic then the vector is cleared (i.e. has zero elements).

node_compat_fn: A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two nodes are compatible.

edge_compat_fn: A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two edges are compatible.

arg: Extra argument to supply to functions *node_compat_fn* and *edge_compat_fn*.

Returns:

Error code.

See also:

`igraph_subisomorphic_vf2()`, `igraph_count_isomorphisms_vf2()`, `igraph_get_isomorphisms_vf2()`,

Time complexity: exponential, what did you expect?

Example 21.1. File `examples/simple/igraph_isomorphic_vf2.c`

`igraph_count_isomorphisms_vf2` — Number of isomorphisms via VF2.

```
igraph_error_t igraph_count_isomorphisms_vf2(const igraph_t *graph1, const igra
                                  const igraph_vector_int_t *vertex_color1,
                                  const igraph_vector_int_t *vertex_color2,
                                  const igraph_vector_int_t *edge_color1,
                                  const igraph_vector_int_t *edge_color2,
                                  igraph_int_t *count,
                                  igraph_isocompat_t *node_compat_fn,
                                  igraph_isocompat_t *edge_compat_fn,
                                  void *arg);
```

This function counts the number of isomorphic mappings between two graphs. It uses the generic `igraph_get_isomorphisms_vf2_callback()` function.

Arguments:

`graph1`: The first input graph, may be directed or undirected.

`graph2`: The second input graph, it must have the same directedness as *`graph1`*, or an error will be reported.

`vertex_color1`: An optional color vector for the first graph. If color vectors are given for both graphs, then the isomorphism is calculated on the colored graphs; i.e. two vertices can match only if their color also matches. Supply a null pointer here if your graphs are not colored.

`vertex_color2`: An optional color vector for the second graph. See the previous argument for explanation.

`edge_color1`: An optional edge color vector for the first graph. The matching edges in the two graphs must have matching colors as well. Supply a null pointer here if your graphs are not edge-colored.

`edge_color2`: The edge color vector for the second graph.

`count`: Point to an integer, the result will be stored here.

`node_compat_fn`: A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two nodes are compatible.

`edge_compat_fn`: A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two edges are compatible.

arg: Extra argument to supply to functions *node_compat_fn* and *edge_compat_fn*.

Returns:

Error code.

See also:

igraph_count_automorphisms_bliss()

Time complexity: exponential.

igraph_get_isomorphisms_vf2 — Collect all isomorphic mappings of two graphs.

```
igraph_error_t igraph_get_isomorphisms_vf2(const igraph_t *graph1,
                                const igraph_t *graph2,
                                const igraph_vector_int_t *vertex_color1,
                                const igraph_vector_int_t *vertex_color2,
                                const igraph_vector_int_t *edge_color1,
                                const igraph_vector_int_t *edge_color2,
                                igraph_vector_int_list_t *maps,
                                igraph_isocompat_t *node_compat_fn,
                                igraph_isocompat_t *edge_compat_fn,
                                void *arg);
```

This function finds all the isomorphic mappings between two simple graphs. It uses the igraph_get_isomorphisms_vf2_callback() function. Call the function with the same graph as *graph1* and *graph2* to get automorphisms.

Arguments:

graph1: The first input graph, may be directed or undirected.

graph2: The second input graph, it must have the same directedness as *graph1*, or an error will be reported.

vertex_color1: An optional color vector for the first graph. If color vectors are given for both graphs, then the isomorphism is calculated on the colored graphs; i.e. two vertices can match only if their color also matches. Supply a null pointer here if your graphs are not colored.

vertex_color2: An optional color vector for the second graph. See the previous argument for explanation.

edge_color1: An optional edge color vector for the first graph. The matching edges in the two graphs must have matching colors as well. Supply a null pointer here if your graphs are not edge-colored.

edge_color2: The edge color vector for the second graph.

maps: Pointer to a list of integer vectors. On return it is empty if the input graphs are not isomorphic. Otherwise it contains pointers to igraph_vector_int_t objects, each vector is an isomorphic mapping of *graph2* to *graph1*.

node_compat_fn: A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two nodes are compatible.

edge_compat_fn: A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two edges are compatible.

arg: Extra argument to supply to functions *node_compat_fn* and *edge_compat_fn*.

Returns:

Error code.

Time complexity: exponential.

`igraph_get_isomorphisms_vf2_callback` — The generic VF2 interface

```
igraph_error_t igraph_get_isomorphisms_vf2_callback(
    const igraph_t *graph1, const igraph_t *graph2,
    const igraph_vector_int_t *vertex_color1, const igraph_vector_int_t *vertex
    const igraph_vector_int_t *edge_color1, const igraph_vector_int_t *edge_col
    igraph_vector_int_t *map12, igraph_vector_int_t *map21,
    igraph_isohandler_t *isohandler_fn, igraph_isocompat_t *node_compat_fn,
    igraph_isocompat_t *edge_compat_fn, void *arg
);
```

This function is an implementation of the VF2 isomorphism algorithm, see P. Foggia, C. Sansone, M. Vento, An Improved algorithm for matching large graphs, Proc. of the 3rd IAPR-TC-15 International Workshop on Graph-based Representations, Italy, 2001.

For using it you need to define a callback function of type `igraph_isohandler_t`. This function will be called whenever VF2 finds an isomorphism between the two graphs. The mapping between the two graphs will be also provided to this function. If the callback returns `IGRAPH_SUCCESS`, then the search is continued, otherwise it stops. `IGRAPH_STOP` as a return value can be used to indicate normal premature termination; any other return value will be treated as an igraph error code, making the caller function return the same error code as well. The callback function must not destroy the mapping vectors that are passed to it.

Arguments:

graph1: The first input graph.

graph2: The second input graph.

vertex_color1: An optional color vector for the first graph. If color vectors are given for both graphs, then the isomorphism is calculated on the colored graphs; i.e. two vertices can match only if their color also matches. Supply a null pointer here if your graphs are not colored.

vertex_color2: An optional color vector for the second graph. See the previous argument for explanation.

edge_color1: An optional edge color vector for the first graph. The matching edges in the two graphs must have matching colors as well. Supply a null pointer here if your graphs are not edge-colored.

edge_color2: The edge color vector for the second graph.

map12: Pointer to an initialized vector or NULL. If not NULL and the supplied graphs are isomorphic then the permutation taking *graph1* to *graph* is stored here. If not NULL and the graphs are not isomorphic then a zero-length vector is returned.

map21: This is the same as *map12*, but for the permutation taking *graph2* to *graph1*.

isohandler_fn: The callback function to be called if an isomorphism is found. See also igraph_isohandler_t.

node_compat_fn: A pointer to a function of type igraph_isocompat_t. This function will be called by the algorithm to determine whether two nodes are compatible.

edge_compat_fn: A pointer to a function of type igraph_isocompat_t. This function will be called by the algorithm to determine whether two edges are compatible.

arg: Extra argument to supply to functions *isohandler_fn*, *node_compat_fn* and *edge_compat_fn*.

Returns:

Error code.

Time complexity: exponential.

igraph_isohandler_t — Callback type, called when an isomorphism was found

```
typedef igraph_error_t igraph_isohandler_t(const igraph_vector_int_t *map12,
        const igraph_vector_int_t *map21, void *arg);
```

See the details at the documentation of igraph_get_isomorphisms_vf2_callback().

Arguments:

map12: The mapping from the first graph to the second.

map21: The mapping from the second graph to the first, the inverse of *map12* basically.

arg: This extra argument was passed to igraph_get_isomorphisms_vf2_callback() when it was called.

Returns:

IGRAPH_SUCCESS to continue the search, IGRAPH_STOP to terminate the search. Any other return value is interpreted as an igraph error code, which will then abort the search and return the same error code from the caller function.

igraph_isocompat_t — Callback type, called to check whether two vertices or edges are compatible

```
typedef igraph_bool_t igraph_isocompat_t(const igraph_t *graph1,
        const igraph_t *graph2,
        const igraph_int_t g1_num,
        const igraph_int_t g2_num,
        void *arg);
```

VF2 (subgraph) isomorphism functions can be restricted by defining relations on the vertices and/or edges of the graphs, and then checking whether the vertices (edges) match according to these relations.

This feature is implemented by two callbacks, one for vertices, one for edges. Every time igraph tries to match a vertex (edge) of the first (sub)graph to a vertex of the second graph, the vertex (edge) compatibility callback is called. The callback returns a logical value, giving whether the two vertices match.

Both callback functions are of type `igraph_isocompat_t`.

Arguments:

graph1: The first graph.

graph2: The second graph.

g1_num: The id of a vertex or edge in the first graph.

g2_num: The id of a vertex or edge in the second graph.

arg: Extra argument to pass to the callback functions.

Returns:

Logical scalar, whether vertex (or edge) *g1_num* in *graph1* is compatible with vertex (or edge) *g2_num* in *graph2*.

`igraph_subisomorphic_vf2` — Decide subgraph isomorphism using VF2

```
igraph_error_t igraph_subisomorphic_vf2(const igraph_t *graph1, const igraph_t
                            const igraph_vector_int_t *vertex_color1,
                            const igraph_vector_int_t *vertex_color2,
                            const igraph_vector_int_t *edge_color1,
                            const igraph_vector_int_t *edge_color2,
                            igraph_bool_t *iso, igraph_vector_int_t *map12,
                            igraph_vector_int_t *map21,
                            igraph_isocompat_t *node_compat_fn,
                            igraph_isocompat_t *edge_compat_fn,
                            void *arg);
```

Decides whether a subgraph of *graph1* is isomorphic to *graph2*. It uses `igraph_get_subisomorphisms_vf2_callback()`.

Arguments:

graph1: The first input graph, may be directed or undirected. This is supposed to be the larger graph.

graph2: The second input graph, it must have the same directedness as *graph1*. This is supposed to be the smaller graph.

vertex_color1: An optional color vector for the first graph. If color vectors are given for both graphs, then the subgraph isomorphism is calculated on the colored graphs; i.e. two vertices can match only if their color also matches. Supply a null pointer here if your graphs are not colored.

vertex_color2: An optional color vector for the second graph. See the previous argument for explanation.

edge_color1: An optional edge color vector for the first graph. The matching edges in the two graphs must have matching colors as well. Supply a null pointer here if your graphs are not edge-colored.

edge_color2: The edge color vector for the second graph.

iso: Pointer to a boolean. The result of the decision problem is stored here.

map12: Pointer to a vector or `NULL`. If not `NULL`, then an isomorphic mapping from *graph1* to *graph2* is stored here.

map21: Pointer to a vector ot `NULL`. If not `NULL`, then an isomorphic mapping from *graph2* to *graph1* is stored here.

node_compat_fn: A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two nodes are compatible.

edge_compat_fn: A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two edges are compatible.

arg: Extra argument to supply to functions *node_compat_fn* and *edge_compat_fn*.

Returns:

Error code.

Time complexity: exponential.

`igraph_count_subisomorphisms_vf2` — Number of subgraph isomorphisms using VF2

```
igraph_error_t igraph_count_subisomorphisms_vf2(const igraph_t *graph1, const ig
                                        const igraph_vector_int_t *vertex_color1,
                                        const igraph_vector_int_t *vertex_color2,
                                        const igraph_vector_int_t *edge_color1,
                                        const igraph_vector_int_t *edge_color2,
                                        igraph_int_t *count,
                                        igraph_isocompat_t *node_compat_fn,
                                        igraph_isocompat_t *edge_compat_fn,
                                        void *arg);
```

Count the number of isomorphisms between subgraphs of *graph1* and *graph2*. This function uses `igraph_get_subisomorphisms_vf2_callback()`.

Arguments:

graph1: The first input graph, may be directed or undirected. This is supposed to be the larger graph.

graph2: The second input graph, it must have the same directedness as *graph1*. This is supposed to be the smaller graph.

vertex_color1: An optional color vector for the first graph. If color vectors are given for both graphs, then the subgraph isomorphism is calculated on the colored graphs; i.e. two vertices can match only if their color also matches. Supply a null pointer here if your graphs are not colored.

vertex_color2: An optional color vector for the second graph. See the previous argument for explanation.

edge_color1: An optional edge color vector for the first graph. The matching edges in the two graphs must have matching colors as well. Supply a null pointer here if your graphs are not edge-colored.

edge_color2: The edge color vector for the second graph.

count: Pointer to an integer. The number of subgraph isomorphisms is stored here.

node_compat_fn: A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two nodes are compatible.

edge_compat_fn: A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two edges are compatible.

arg: Extra argument to supply to functions *node_compat_fn* and *edge_compat_fn*.

Returns:

Error code.

Time complexity: exponential.

`igraph_get_subisomorphisms_vf2` — Return all subgraph isomorphic mappings.

```
igraph_error_t igraph_get_subisomorphisms_vf2(const igraph_t *graph1,
                                    const igraph_t *graph2,
                                    const igraph_vector_int_t *vertex_color1,
                                    const igraph_vector_int_t *vertex_color2,
                                    const igraph_vector_int_t *edge_color1,
                                    const igraph_vector_int_t *edge_color2,
                                    igraph_vector_int_list_t *maps,
                                    igraph_isocompat_t *node_compat_fn,
                                    igraph_isocompat_t *edge_compat_fn,
                                    void *arg);
```

This function collects all isomorphic mappings of *graph2* to a subgraph of *graph1*. It uses the `igraph_get_subisomorphisms_vf2_callback()` function. The graphs should be simple.

Arguments:

graph1: The first input graph, may be directed or undirected. This is supposed to be the larger graph.

graph2: The second input graph, it must have the same directedness as *graph1*. This is supposed to be the smaller graph.

vertex_color1: An optional color vector for the first graph. If color vectors are given for both graphs, then the subgraph isomorphism is calculated on the colored graphs; i.e. two vertices can match only if their color also matches. Supply a null pointer here if your graphs are not colored.

vertex_color2: An optional color vector for the second graph. See the previous argument for explanation.

edge_color1: An optional edge color vector for the first graph. The matching edges in the two graphs must have matching colors as well. Supply a null pointer here if your graphs are not edge-colored.

edge_color2: The edge color vector for the second graph.

maps: Pointer to a list of integer vectors. On return it contains pointers to `igraph_vector_int_t` objects, each vector is an isomorphic mapping of *graph2* to a subgraph of *graph1*.

node_compat_fn: A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two nodes are compatible.

edge_compat_fn: A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two edges are compatible.

arg: Extra argument to supply to functions *node_compat_fn* and *edge_compat_fn*.

Returns:

Error code.

Time complexity: exponential.

`igraph_get_subisomorphisms_vf2_callback` — Generic VF2 function for subgraph isomorphism problems.

```
igraph_error_t igraph_get_subisomorphisms_vf2_callback(
    const igraph_t *graph1, const igraph_t *graph2,
    const igraph_vector_int_t *vertex_color1, const igraph_vector_int_t *vertex
    const igraph_vector_int_t *edge_color1, const igraph_vector_int_t *edge_col
    igraph_vector_int_t *map12, igraph_vector_int_t *map21,
    igraph_isohandler_t *isohandler_fn, igraph_isocompat_t *node_compat_fn,
    igraph_isocompat_t *edge_compat_fn, void *arg
);
```

This function is the pair of `igraph_get_isomorphisms_vf2_callback()`, for subgraph isomorphism problems. It searches for subgraphs of *graph1* which are isomorphic to *graph2*. When it founds an isomorphic mapping it calls the supplied callback *isohandler_fn*. The mapping (and its inverse) and the additional *arg* argument are supplied to the callback.

Arguments:

graph1: The first input graph, may be directed or undirected. This is supposed to be the larger graph.

graph2: The second input graph, it must have the same directedness as *graph1*. This is supposed to be the smaller graph.

vertex_color1: An optional color vector for the first graph. If color vectors are given for both graphs, then the subgraph isomorphism is calculated on the colored graphs; i.e. two vertices can match only if their color also matches. Supply a null pointer here if your graphs are not colored.

vertex_color2: An optional color vector for the second graph. See the previous argument for explanation.

edge_color1: An optional edge color vector for the first graph. The matching edges in the two graphs must have matching colors as well. Supply a null pointer here if your graphs are not edge-colored.

edge_color2: The edge color vector for the second graph.

map12: Pointer to a vector or `NULL`. If not `NULL`, then an isomorphic mapping from *graph1* to *graph2* is stored here.

map21: Pointer to a vector ot `NULL`. If not `NULL`, then an isomorphic mapping from *graph2* to *graph1* is stored here.

isohandler_fn: A pointer to a function of type `igraph_isohandler_t`. This will be called whenever a subgraph isomorphism is found. If the function returns `IGRAPH_SUCCESS`, then the search is continued. If the function returns `IGRAPH_STOP`, the search is terminated normally. Any other value is treated as an igraph error code.

node_compat_fn: A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two nodes are compatible.

edge_compat_fn: A pointer to a function of type `igraph_isocompat_t`. This function will be called by the algorithm to determine whether two edges are compatible.

arg: Extra argument to supply to functions *isohandler_fn*, *node_compat_fn* and *edge_compat_fn*.

Returns:

Error code.

Time complexity: exponential.

The LAD algorithm

The LAD algorithm can search for a subgraph in a larger graph, or check if two graphs are isomorphic. See Christine Solnon: AllDifferent-based Filtering for Subgraph Isomorphism. Artificial Intelligence, 174(12-13):850-864, 2010. https://doi.org/10.1016/j.artint.2010.05.002 as well as the homepage of the LAD library at http://liris.cnrs.fr/csolnon/LAD.html The implementation in igraph is based on LADv1, but it is modified to use igraph's own memory allocation and error handling.

LAD uses the concept of domains to indicate vertex compatibility when matching the pattern graph. Domains can be used to implement matching of colored vertices.

LAD works with both directed and undirected graphs. Graphs with multi-edges are not supported.

`igraph_subisomorphic_lad` — Check subgraph isomorphism with the LAD algorithm

```
igraph_error_t igraph_subisomorphic_lad(const igraph_t *pattern, const igraph_t
                              const igraph_vector_int_list_t *domains,
                              igraph_bool_t *iso, igraph_vector_int_t *map,
                              igraph_vector_int_list_t *maps,
                              igraph_bool_t induced);
```

Check whether *pattern* is isomorphic to a subgraph os *target*. The original LAD implementation by Christine Solnon was used as the basis of this code.

See more about LAD at http://liris.cnrs.fr/csolnon/LAD.html and in Christine Solnon: AllDifferent-based Filtering for Subgraph Isomorphism. Artificial Intelligence, 174(12-13):850-864, 2010. https://doi.org/10.1016/j.artint.2010.05.002

Arguments:

pattern: The smaller graph, it can be directed or undirected.

target: The bigger graph, it can be directed or undirected.

domains: An integer vector list of `NULL`. The length of each vector must match the number of vertices in the *pattern* graph. For each vertex, the IDs of the compatible vertices in the target graph are listed.

iso: Pointer to a boolean, or a null pointer. If not a null pointer, then the boolean is set to `true` if a subgraph isomorphism is found, and to `false` otherwise.

map: Pointer to a vector or a null pointer. If not a null pointer and a subgraph isomorphism is found, the matching vertices from the target graph are listed here, for each vertex (in vertex ID order) from the pattern graph.

maps: Pointer to a list of integer vectors or a null pointer. If not a null pointer, then all subgraph isomorphisms are stored in the vector list, in `igraph_vector_int_t` objects.

induced: Boolean, whether to search for induced matching subgraphs.

time_limit: Processor time limit in seconds. Supply zero here for no limit. If the time limit is over, then the function signals an error.

Returns:

Error code

See also:

`igraph_subisomorphic_vf2()` for the VF2 algorithm.

Time complexity: exponential.

Example 21.2. File `examples/simple/igraph_subisomorphic_lad.c`

Functions for small graphs

`igraph_isoclass` — Determine the isomorphism class of small graphs.

```
igraph_error_t igraph_isoclass(const igraph_t *graph, igraph_int_t *isoclass);
```

All graphs with a given number of vertices belong to a number of isomorphism classes, with every graph in a given class being isomorphic to each other.

This function gives the isomorphism class (a number) of a graph. Two graphs have the same isomorphism class if and only if they are isomorphic.

The first isomorphism class is numbered zero and it contains the edgeless graph. The last isomorphism class contains the full graph. The number of isomorphism classes for directed graphs with three vertices is 16 (between 0 and 15), for undirected graph it is only 4. For graphs with four vertices it is 218 (directed) and 11 (undirected). For 5 and 6 vertex undirected graphs, it is 34 and 156, respectively. These values can also be retrieved using `igraph_graph_count()`. For more information, see https://oeis.org/A000273 and https://oeis.org/A000088.

At the moment, 3- and 4-vertex directed graphs and 3 to 6 vertex undirected graphs are supported.

Multi-edges and self-loops are ignored by this function.

Arguments:

`graph`: The graph object.

`isoclass`: Pointer to an integer, the isomorphism class will be stored here.

Returns:

Error code.

See also:

`igraph_isomorphic()`, `igraph_isoclass_subgraph()`, `igraph_isoclass_create()`, `igraph_motifs_randesu()`.

Because of some limitations this function works only for graphs with three of four vertices.

Time complexity: O(|E|), the number of edges in the graph.

`igraph_isoclass_subgraph` — The isomorphism class of a subgraph of a graph.

```
igraph_error_t igraph_isoclass_subgraph(const igraph_t *graph, igraph_vs_t vids,
                                        igraph_int_t *isoclass);
```

This function identifies the isomorphism class of the subgraph induced the vertices specified in *`vids`*.

At the moment, 3- and 4-vertex directed graphs and 3 to 6 vertex undirected graphs are supported.

Multi-edges and self-loops are ignored by this function.

Arguments:

`graph`: The graph object.

`vids`: The vertices of the subgraph. Each vertex must be included at most once.

`isoclass`: Pointer to an integer, this will be set to the isomorphism class.

Returns:

Error code.

See also:

`igraph_isoclass()`, `igraph_isomorphic()`, `igraph_isoclass_create()`.

Time complexity: O((d+n)*n), d is the average degree in the network, and n is the number of vertices in `vids`.

`igraph_isoclass_create` — Creates a graph from the given isomorphism class.

```
igraph_error_t igraph_isoclass_create(igraph_t *graph, igraph_int_t size,
                            igraph_int_t number, igraph_bool_t directed);
```

This function creates the canonical representative graph of the given isomorphism class.

The isomorphism class is an integer between 0 and the number of unique unlabeled (i.e. non-isomorphic) graphs on the given number of vertices and give directedness. See https://oeis.org/A000273 and https://oeis.org/A000088 for the number of directed and undirected graphs on *`size`* nodes.

At the moment, 3- and 4-vertex directed graphs and 3 to 6 vertex undirected graphs are supported.

Arguments:

`graph`: Pointer to an uninitialized graph object.

`size`: The number of vertices to add to the graph.

`number`: The isomorphism class.

`directed`: Boolean constant, whether to create a directed graph.

Returns:

Error code.

See also:

`igraph_isoclass()`, `igraph_isoclass_subgraph()`, `igraph_isomorphic()`.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges in the graph to create.

`igraph_graph_count` — The number of unlabelled graphs on the given number of vertices.

```
igraph_error_t igraph_graph_count(igraph_int_t n, igraph_bool_t directed, igraph
```

Gives the number of unlabelled *simple* graphs on the specified number of vertices. The "isoclass" of a graph of this size is at most one less than this value.

This function is meant to be used in conjunction with isoclass and motif finder functions. It will only work for small *n* values for which the result is represetable in an igraph_int_t. For larger *n* values, an overflow error is raised.

Arguments:

n: The number of vertices.

directed: Boolean, whether to consider directed graphs.

count: Pointer to an integer, the result will be stored here.

Returns:

Error code.

See also:

`igraph_isoclass()`, `igraph_motifs_randesu_callback()`.

Time complexity: O(1).

Utility functions

`igraph_invert_permutation` — Inverts a permutation.

```
igraph_error_t igraph_invert_permutation(const igraph_vector_int_t *permutation
```

Produces the inverse of *permutation* into *inverse* and at the same time it checks that the permutation vector is valid, i.e. all indices are within range and there are no duplicate entries.

Arguments:

permutation: A permutation vector containing 0-based integer indices.

inverse: An initialized vector. The inverse of *permutation* will be stored here.

Returns:

Error code.

`igraph_permute_vertices` — Permute the vertices.

```
igraph_error_t igraph_permute_vertices(const igraph_t *graph, igraph_t *res,
                                       const igraph_vector_int_t *permutation);
```

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 `igraph_canonical_permutation()` to create the canonical form of a graph.

Arguments:

graph: The input graph.

res: Pointer to an uninitialized graph object. The new graph is created here.

permutation: The permutation to apply. The i-th element of the vector specifies the index of the vertex in the original graph that will become vertex i in the new graph.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in terms of the number of vertices and edges.

`igraph_simplify_and_colorize` — Simplify the graph and compute self-loop and edge multiplicities.

```
igraph_error_t igraph_simplify_and_colorize(
    const igraph_t *graph, igraph_t *res,
    igraph_vector_int_t *vertex_color, igraph_vector_int_t *edge_color);
```

This function creates a vertex and edge colored simple graph from the input graph. The vertex colors are computed as the number of incident self-loops to each vertex in the input graph. The edge colors are computed as the number of parallel edges in the input graph that were merged to create each edge in the simple graph.

The resulting colored simple graph is suitable for use by isomorphism checking algorithms such as VF2, which only support simple graphs, but can consider vertex and edge colors.

Arguments:

graph: The graph object, typically having self-loops or multi-edges.

res: An uninitialized graph object. The result will be stored here

vertex_color: Computed vertex colors corresponding to self-loop multiplicities.

edge_color: Computed edge colors corresponding to edge multiplicities

Returns:

Error code.

See also:

`igraph_simplify()`, `igraph_isomorphic_vf2()`, `igraph_subisomorphic_vf2()`

Chapter 22. Graph coloring

`igraph_vertex_coloring_greedy` — Computes a vertex coloring using a greedy algorithm.

```
igraph_error_t igraph_vertex_coloring_greedy(const igraph_t *graph, igraph_vecto
```

This function assigns a "color"—represented as a non-negative integer—to each vertex of the graph in such a way that neighboring vertices never have the same color. The obtained coloring is not necessarily minimal.

Vertices are colored greedily, one by one, always choosing the smallest color index that differs from that of already colored neighbors. Vertices are picked in an order determined by the speified heuristic. Colors are represented by non-negative integers 0, 1, 2, ...

Arguments:

`graph`: The input graph.

`colors`: Pointer to an initialized integer vector. The vertex colors will be stored here.

`heuristic`: The vertex ordering heuristic to use during greedy coloring. See `igraph_coloring_greedy_t` for more information.

Returns:

Error code.

See also:

igraph_is_vertex_coloring() to check if a coloring is valid, i.e. if all edges connect vertices of different colors.

Example 22.1. File `examples/simple/coloring.c`

`igraph_coloring_greedy_t` — Ordering heuristics for greedy graph coloring.

```
typedef enum {
    IGRAPH_COLORING_GREEDY_COLORED_NEIGHBORS = 0,
    IGRAPH_COLORING_GREEDY_DSATUR = 1
} igraph_coloring_greedy_t;
```

Ordering heuristics for `igraph_vertex_coloring_greedy()`.

Values:

`IGRAPH_COLORING_GREEDY_COLORED_NEIGHBORS`: Choose the vertex with largest number of already colored neighbors.

`IGRAPH_COLORING_GREEDY_DSATUR`: Choose the vertex with largest number of unique colors in its neighborhood, i.e. its "saturation degree". When multiple vertices have the same saturation degree, choose the one with the most not yet colored neighbors. Added in igraph 0.10.4. This heuristic is known as "DSatur", and was proposed in Daniel Brélaz: New methods to color the vertices of a graph, Commun. ACM 22, 4 (1979), 251–256. https://doi.org/10.1145/359094.359101

`igraph_is_vertex_coloring` — Checks whether a vertex coloring is valid.

```
igraph_error_t igraph_is_vertex_coloring(
        const igraph_t *graph,
        const igraph_vector_int_t *types,
        igraph_bool_t *res);
```

This function checks whether the given vertex type/color assignment is a valid vertex coloring, i.e., no two adjacent vertices have the same color. Self-loops are ignored.

Arguments:

`graph`: The input graph.

`types`: The vertex types/colors as an integer vector.

`res`: Pointer to a boolean, the result is stored here.

Returns:

Error code.

Time complexity: O(|E|), linear in the number of edges.

Example 22.2. File `examples/simple/coloring.c`

`igraph_is_bipartite_coloring` — Checks whether a bipartite vertex coloring is valid.

```
igraph_error_t igraph_is_bipartite_coloring(
        const igraph_t *graph,
        const igraph_vector_bool_t *types,
        igraph_bool_t *res,
        igraph_neimode_t *mode);
```

This function checks whether the given vertex type assignment is a valid bipartite coloring, i.e., no two adjacent vertices have the same type. Additionally, for directed graphs, it determines the mode of edge directions. Self-loops are ignored.

Arguments:

graph: The input graph.

types: The vertex types as a boolean vector.

res: Pointer to a boolean, the result is stored here.

mode: Pointer to store the edge direction mode. Can be `NULL` if not needed. If all edges go from false to true vertices, `IGRAPH_OUT` is returned. If all edges go from true to false vertices, `IGRAPH_IN` is returned. If edges go in both directions or graph is undirected, `IGRAPH_ALL` is returned.

Returns:

Error code.

Time complexity: O(|E|), linear in the number of edges.

See also:

igraph_is_bipartite() to determine whether a graph is bipartite, i.e. 2-colorable, and find such a coloring.

`igraph_is_edge_coloring` — Checks whether an edge coloring is valid.

```
igraph_error_t igraph_is_edge_coloring(
        const igraph_t *graph,
        const igraph_vector_int_t *types,
        igraph_bool_t *res);
```

This function checks whether the given edge color assignment is a valid edge coloring, i.e., no two adjacent edges have the same color. Note that this function does not consider self-edges (loops) as being adjacent to themselves, so graphs with self-loops may still be considered to have a valid edge coloring.

Arguments:

graph: The input graph.

types: The edge colors as an integer vector.

res: Pointer to a boolean, the result is stored here.

Returns:

Error code.

Time complexity: O(|V|*d*log(d)), where d is the maximum degree.

`igraph_is_perfect` — Checks if the graph is perfect.

```
igraph_error_t igraph_is_perfect(const igraph_t *graph, igraph_bool_t *perfect)
```

A perfect graph is an undirected graph in which the chromatic number of every induced subgraph equals the order of the largest clique of that subgraph. The chromatic number of a graph G is the smallest number of colors needed to color the vertices of G so that no two adjacent vertices share the same color.

Warning: This function may create the complement of the graph internally, which consumes a lot of memory. For moderately sized graphs, consider decomposing them into biconnected components and running the check separately on each component.

This implementation is based on the strong perfect graph theorem which was conjectured by Claude Berge and proved by Maria Chudnovsky, Neil Robertson, Paul Seymour, and Robin Thomas.

Arguments:

graph: The input graph. It is expected to be undirected and simple.

perfect: Pointer to an integer, the result will be stored here.

Returns:

Error code.

Time complexity: worst case exponenital, often faster in practice.

Chapter 23. Maximum flows, minimum cuts and related measures

Maximum flows

`igraph_maxflow` — Maximum network flow between a pair of vertices.

```
igraph_error_t igraph_maxflow(const igraph_t *graph, igraph_real_t *value,
                              igraph_vector_t *flow, igraph_vector_int_t *cut,
                              igraph_vector_int_t *partition, igraph_vector_int_
                              igraph_int_t source, igraph_int_t target,
                              const igraph_vector_t *capacity,
                              igraph_maxflow_stats_t *stats);
```

This function implements the Goldberg-Tarjan algorithm for calculating value of the maximum flow in a directed or undirected graph. The algorithm was given in Andrew V. Goldberg, Robert E. Tarjan: A New Approach to the Maximum-Flow Problem, Journal of the ACM, 35(4), 921-940, 1988 https://doi.org/10.1145/48014.61051.

The input of the function is a graph, a vector of real numbers giving the capacity of the edges and two vertices of the graph, the source and the target. A flow is a function assigning positive real numbers to the edges and satisfying two requirements: (1) the flow value is less than the capacity of the edge and (2) at each vertex except the source and the target, the incoming flow (i.e. the sum of the flow on the incoming edges) is the same as the outgoing flow (i.e. the sum of the flow on the outgoing edges). The value of the flow is the incoming flow at the target vertex. The maximum flow is the flow with the maximum value.

Arguments:

graph: The input graph, either directed or undirected.

value: Pointer to a real number, the value of the maximum will be placed here, unless it is a null pointer.

flow: If not a null pointer, then it must be a pointer to an initialized vector. The vector will be resized, and the flow on each edge will be placed in it, in the order of the edge IDs. For undirected graphs this argument is bit trickier, since for these the flow direction is not predetermined by the edge direction. For these graphs the elements of the *flow* 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 null pointer or a pointer to an initialized vector. If not a null pointer, then the minimum cut corresponding to the maximum flow is stored here, i.e. all edge IDs that are part of the minimum cut are stored in the vector.

partition: A null pointer or a pointer to an initialized vector. If not a null pointer, then the first partition of the minimum cut that corresponds to the maximum flow will be placed here. The first partition is always the one that contains the source vertex.

partition2: A null pointer or a pointer to an initialized vector. If not a null pointer, then the second partition of the minimum cut that corresponds to the maximum flow will be placed here. The second partition is always the one that contains the target vertex.

source: The id of the source vertex.

target: The id of the target vertex.

capacity: Vector containing the capacity of the edges. If NULL, then every edge is considered to have capacity 1.0.

stats: Counts of the number of different operations performed by the algorithm are stored here.

Returns:

Error code.

Time complexity: O(|V|^3). In practice it is much faster, but I cannot prove a better lower bound for the data structure I've used. In fact, this implementation runs much faster than the `hi_pr` implementation discussed in B. V. Cherkassky and A. V. Goldberg: On implementing the push-relabel method for the maximum flow problem, (Algorithmica, 19:390--410, 1997) on all the graph classes I've tried.

See also:

`igraph_mincut_value()`, `igraph_edge_connectivity()`, `igraph_vertex_connectivity()` for properties based on the maximum flow.

Example 23.1. File `examples/simple/flow.c`

Example 23.2. File `examples/simple/flow2.c`

`igraph_maxflow_value` — Maximum flow in a network with the push/relabel algorithm.

```
igraph_error_t igraph_maxflow_value(const igraph_t *graph, igraph_real_t *value,
                                    igraph_int_t source, igraph_int_t target,
                                    const igraph_vector_t *capacity,
                                    igraph_maxflow_stats_t *stats);
```

This function implements the Goldberg-Tarjan algorithm for calculating value of the maximum flow in a directed or undirected graph. The algorithm was given in Andrew V. Goldberg, Robert E. Tarjan: A New Approach to the Maximum-Flow Problem, Journal of the ACM, 35(4), 921-940, 1988 https://doi.org/10.1145/48014.61051.

The input of the function is a graph, a vector of real numbers giving the capacity of the edges and two vertices of the graph, the source and the target. A flow is a function assigning positive real numbers to the edges and satisfying two requirements: (1) the flow value is less than the capacity of the edge and (2) at each vertex except the source and the target, the incoming flow (i.e. the sum of the flow on the incoming edges) is the same as the outgoing flow (i.e. the sum of the flow on the outgoing edges). The value of the flow is the incoming flow at the target vertex. The maximum flow is the flow with the maximum value.

According to a theorem by Ford and Fulkerson (L. R. Ford Jr. and D. R. Fulkerson. Maximal flow through a network. Canadian J. Math., 8:399-404, 1956.) the maximum flow between two vertices is the same as the minimum cut between them (also called the minimum s-t cut). So `igraph_st_mincut_value()` gives the same result in all cases as `igraph_maxflow_value()`.

Note that the value of the maximum flow is the same as the minimum cut in the graph.

Arguments:

graph: The input graph, either directed or undirected.

value: Pointer to a real number, the result will be placed here.

source: The id of the source vertex.

target: The id of the target vertex.

capacity: Vector containing the capacity of the edges. If NULL, then every edge is considered to have capacity 1.0.

stats: Counts of the number of different operations preformed by the algorithm are stored here.

Returns:

Error code.

Time complexity: O(|V|^3).

See also:

`igraph_maxflow()` to calculate the actual flow. `igraph_mincut_value()`, `igraph_edge_connectivity()`, `igraph_vertex_connectivity()` for properties based on the maximum flow.

`igraph_dominator_tree` — Calculates the dominator tree of a flowgraph.

```
igraph_error_t igraph_dominator_tree(const igraph_t *graph,
                                     igraph_int_t root,
                                     igraph_vector_int_t *dom,
                                     igraph_t *domtree,
                                     igraph_vector_int_t *leftout,
                                     igraph_neimode_t mode);
```

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=idom(w), if v dominates w and every other dominator of w dominates v. The edges {(idom(w), w)| w is not 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 please see 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. https://doi.org/10.1145/357062.357071

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 in the *leftout* vector.

root: The ID of the root (or source) vertex, this will be the root of the tree.

dom: Pointer to an initialized vector or a null pointer. If not a null pointer, then the immediate dominator of each vertex will be stored here. For vertices that are not reachable from the root, `-2` is stored here. For the root vertex itself, `-1` is added.

domtree: Pointer to an *uninitialized* igraph_t, or `NULL`. If not a null pointer, then the dominator tree is returned here. The graph contains the vertices that are unreachable from the root (if any), these will be isolates. Graph and vertex attributes are preserved, but edge attributes are discarded.

leftout: Pointer to an initialized vector object, or `NULL`. If not `NULL`, then the IDs of the vertices that are unreachable from the root vertex (and thus not part of the dominator tree) are stored here.

mode: Constant, must be `IGRAPH_IN` or `IGRAPH_OUT`. If it is `IGRAPH_IN`, then all directions are considered as opposite to the original one in the input graph.

Returns:

Error code.

Time complexity: very close to O(|E|+|V|), linear in the number of edges and vertices. More precisely, it is O(|V|+|E|alpha(|E|,|V|)), where alpha(|E|,|V|) is a functional inverse of Ackermann's function.

Example 23.3. File `examples/simple/dominator_tree.c`

`igraph_maxflow_stats_t` — Data structure holding statistics from the push-relabel maximum flow solver.

```
typedef struct {
    igraph_int_t nopush, norelabel, nogap, nogapnodes, nobfs;
```

Arguments:

nopush: The number of push operations performed.

norelabel: The number of relabel operarions performed.

nogap: The number of times the gap heuristics was used.

nogapnodes: The total number of vertices that were omitted form further calculations because of the gap heuristics.

nobfs: The number of times the reverse BFS was run to assign good values to the height function. This includes an initial run before the whole algorithm, so it is always at least one.

Cuts and minimum cuts

`igraph_st_mincut` — Minimum cut between a source and a target vertex.

```
igraph_error_t igraph_st_mincut(const igraph_t *graph, igraph_real_t *value,
                                igraph_vector_int_t *cut, igraph_vector_int_t *
                                igraph_vector_int_t *partition2,
                                igraph_int_t source, igraph_int_t target,
                                const igraph_vector_t *capacity);
```

Finds the edge set that has the smallest total capacity among all edge sets that disconnect the source and target vertices.

The calculation is performed using maximum flow techniques, by calling `igraph_maxflow()`.

Arguments:

graph: The input graph.

value: Pointer to a real variable, the value of the cut is stored here.

cut: Pointer to an initialized vector, the edge IDs that are included in the cut are stored here. This argument is ignored if it is a null pointer.

partition: Pointer to an initialized vector, the vertex IDs of the vertices in the first partition of the cut are stored here. The first partition is always the one that contains the source vertex. This argument is ignored if it is a null pointer.

partition2: Pointer to an initialized vector, the vertex IDs of the vertices in the second partition of the cut are stored here. The second partition is always the one that contains the target vertex. This argument is ignored if it is a null pointer.

source: Integer, the id of the source vertex.

target: Integer, the id of the target vertex.

capacity: Vector containing the capacity of the edges. If a null pointer, then every edge is considered to have capacity 1.0.

Returns:

Error code.

See also:

`igraph_maxflow()`.

Time complexity: see `igraph_maxflow()`.

`igraph_st_mincut_value` — The minimum s-t cut in a graph.

```
igraph_error_t igraph_st_mincut_value(const igraph_t *graph, igraph_real_t *valu
                                      igraph_int_t source, igraph_int_t target,
                                      const igraph_vector_t *capacity);
```

The minimum s-t cut in a weighted (=valued) graph is the total minimum edge weight needed to remove from the graph to eliminate all paths from a given vertex (*source*) to another vertex (*target*). Directed paths are considered in directed graphs, and undirected paths in undirected graphs.

The minimum s-t cut between two vertices is known to be same as the maximum flow between these two vertices. So this function calls `igraph_maxflow_value()` to do the calculation.

Arguments:

graph: The input graph.

value: Pointer to a real variable, the result will be stored here.

source: The id of the source vertex.

target: The id of the target vertex.

capacity: Pointer to the capacity vector, it should contain non-negative numbers and its length should be the same the the number of edges in the graph. It can be a null pointer, then every edge has unit capacity.

Returns:

Error code.

Time complexity: O(|V|^3), see also the discussion for `igraph_maxflow_value()`, |V| is the number of vertices.

`igraph_all_st_cuts` — List all edge-cuts between two vertices in a directed graph

```
igraph_error_t igraph_all_st_cuts(const igraph_t *graph,
                                  igraph_vector_int_list_t *cuts,
                                  igraph_vector_int_list_t *partition1s,
                                  igraph_int_t source,
                                  igraph_int_t target);
```

This function lists all edge-cuts between a source and a target vertex. Every cut is listed exactly once. The implemented algorithm is described in JS Provan and DR Shier: A Paradigm for listing (s,t)-cuts in graphs, Algorithmica 15, 351--372, 1996.

Arguments:

graph: The input graph, is must be directed.

cuts: An initialized list of integer vectors, the cuts are stored here. Each vector will contain the IDs of the edges in the cut. This argument is ignored if it is a null pointer.

partition1s: An initialized list of integer vectors, the list of vertex sets generating the actual edge cuts are stored here. Each vector contains a set of vertex IDs. If X is such a set, then all edges going from X to the complement of X form an (s, t) edge-cut in the graph. This argument is ignored if it is a null pointer.

source: The id of the source vertex.

target: The id of the target vertex.

Returns:

Error code.

Time complexity: O(n(|V|+|E|)), where |V| is the number of vertices, |E| is the number of edges, and n is the number of cuts.

`igraph_all_st_mincuts` — All minimum s-t cuts of a directed graph.

```
igraph_error_t igraph_all_st_mincuts(const igraph_t *graph, igraph_real_t *valu
                                     igraph_vector_int_list_t *cuts,
                                     igraph_vector_int_list_t *partition1s,
                                     igraph_int_t source,
                                     igraph_int_t target,
                                     const igraph_vector_t *capacity);
```

This function lists all edge cuts between two vertices, in a directed graph, with minimum total capacity. Possibly, multiple cuts may have the same total capacity, although there is often only one minimum cut in weighted graphs. It is recommended to supply integer-values capacities. Otherwise, not all minimum cuts may be detected because of numerical roundoff errors. The implemented algorithm is described in JS Provan and DR Shier: A Paradigm for listing (s,t)-cuts in graphs, Algorithmica 15, 351--372, 1996.

Arguments:

`graph`: The input graph, it must be directed.

`value`: Pointer to a real number or `NULL`. The value of the minimum cut is stored here, unless it is a null pointer.

`cuts`: Pointer to initialized list of integer vectors or `NULL`. The cuts are stored here as lists of vertex IDs.

`partition1s`: Pointer to an initialized list of integer vectors or `NULL`. The list of vertex sets, generating the actual edge cuts, are stored here. Each vector contains a set of vertex IDs. If X is such a set, then all edges going from X to the complement of X form an (s,t) edge-cut in the graph.

`source`: The id of the source vertex.

`target`: The id of the target vertex.

`capacity`: Vector of edge capacities. All capacities must be strictly positive. If this is a null pointer, then all edges are assumed to have capacity one.

Returns:

Error code.

Time complexity: O(n(|V|+|E|))+O(F), where |V| is the number of vertices, |E| is the number of edges, and n is the number of cuts; O(F) is the time complexity of the maximum flow algorithm, see `igraph_maxflow()`.

Example 23.4. File `examples/simple/igraph_all_st_mincuts.c`

`igraph_mincut` — Calculates the minimum cut in a graph.

```
igraph_error_t igraph_mincut(const igraph_t *graph,
                             igraph_real_t *value,
                             igraph_vector_int_t *partition,
                             igraph_vector_int_t *partition2,
                             igraph_vector_int_t *cut,
                             const igraph_vector_t *capacity);
```

This function calculates the minimum cut in a graph. The minimum cut is the minimum set of edges which needs to be removed to disconnect the graph. The minimum is calculated using the weights (*capacity*) of the edges, so the cut with the minimum total capacity is calculated.

For directed graphs an implementation based on calculating 2|V|-2 maximum flows is used. For undirected graphs we use the Stoer-Wagner algorithm, as described in M. Stoer and F. Wagner: A simple min-cut algorithm, Journal of the ACM, 44 585-591, 1997.

The first implementation of the actual cut calculation for undirected graphs was made by Gregory Benison, thanks Greg.

Arguments:

graph: The input graph.

value: Pointer to a float, the value of the cut will be stored here.

partition: Pointer to an initialized vector, the ids of the vertices in the first partition after separating the graph will be stored here. The vector will be resized as needed. This argument is ignored if it is a NULL pointer.

partition2: Pointer to an initialized vector the ids of the vertices in the second partition will be stored here. The vector will be resized as needed. This argument is ignored if it is a NULL pointer.

cut: Pointer to an initialized vector, the IDs of the edges in the cut will be stored here. This argument is ignored if it is a NULL pointer.

capacity: A numeric vector giving the capacities of the edges. If a null pointer then all edges have unit capacity.

Returns:

Error code.

See also:

`igraph_mincut_value()`, a simpler interface for calculating the value of the cut only.

Time complexity: for directed graphs it is O(|V|^4), but see the remarks at `igraph_maxflow()`. For undirected graphs it is O(|V||E|+|V|^2 log|V|). |V| and |E| are the number of vertices and edges respectively.

Example 23.5. File `examples/simple/igraph_mincut.c`

`igraph_mincut_value` — The minimum edge cut in a graph.

```
igraph_error_t igraph_mincut_value(const igraph_t *graph, igraph_real_t *res,
                                   const igraph_vector_t *capacity);
```

The minimum edge cut in a graph is the total minimum weight of the edges needed to remove from the graph to make the graph *not* strongly connected. (If the original graph is not strongly connected then this is zero.) Note that in undirected graphs strong connectedness is the same as weak connectedness.

The minimum cut can be calculated with maximum flow techniques, although the current implementation does this only for directed graphs and a separate non-flow based implementation is used for undirected graphs. See Mechthild Stoer and Frank Wagner: A simple min-cut algorithm, Journal of the ACM 44 585--591, 1997. For directed graphs the maximum flow is calculated between a fixed vertex and all the other vertices in the graph and this is done in both directions. Then the minimum is taken to get the minimum cut.

Arguments:

graph: The input graph.

res: Pointer to a real variable, the result will be stored here.

capacity: Pointer to the capacity vector, it should contain the same number of non-negative numbers as the number of edges in the graph. If a null pointer then all edges will have unit capacity.

Returns:

Error code.

See also:

`igraph_mincut()`, `igraph_maxflow_value()`, `igraph_st_mincut_value()`.

Time complexity: O(log(|V|)*|V|^2) for undirected graphs and O(|V|^4) for directed graphs, but see also the discussion at the documentation of `igraph_maxflow_value()`.

`igraph_gomory_hu_tree` — Gomory-Hu tree of a graph.

```
igraph_error_t igraph_gomory_hu_tree(const igraph_t *graph,
                                     igraph_t *tree,
                                     igraph_vector_t *flows,
                                     const igraph_vector_t *capacity);
```

The Gomory-Hu tree is a concise representation of the value of all the maximum flows (or minimum cuts) in a graph. The vertices of the tree correspond exactly to the vertices of the original graph in the same order. Edges of the Gomory-Hu tree are annotated by flow values. The value of the maximum flow (or minimum cut) between an arbitrary (u,v) vertex pair in the original graph is then given by the minimum flow value (i.e. edge annotation) along the shortest path between u and v in the Gomory-Hu tree.

This implementation uses Gusfield's algorithm to construct the Gomory-Hu tree. See the following paper for more details:

Reference:

Gusfield D: Very simple methods for all pairs network flow analysis. SIAM J Comput 19(1):143-155, 1990 https://doi.org/10.1137/0219009.

Arguments:

graph: The input graph.

tree: Pointer to an uninitialized graph; the result will be stored here.

flows: Pointer to an uninitialized vector; the flow values corresponding to each edge in the Gomory-Hu tree will be returned here. You may pass a NULL pointer here if you are not interested in the flow values.

capacity: Vector containing the capacity of the edges. If NULL, then every edge is considered to have capacity 1.0.

Returns:

Error code.

Time complexity: O(|V|^4) since it performs a max-flow calculation between vertex zero and every other vertex and max-flow is O(|V|^3).

See also:

`igraph_maxflow()`

Connectivity

`igraph_st_edge_connectivity` — Edge connectivity of a pair of vertices.

```
igraph_error_t igraph_st_edge_connectivity(const igraph_t *graph,
                                           igraph_int_t *res,
                                           igraph_int_t source,
                                           igraph_int_t target);
```

The edge connectivity of two vertices (*source* and *target*) is the minimum number of edges that have to be deleted from the graph to eliminate all paths from *source* to *target*.

This function uses the maximum flow algorithm to calculate the edge connectivity.

Arguments:

graph: The input graph, it has to be directed.

res: Pointer to an integer, the result will be stored here.

source: The id of the source vertex.

target: The id of the target vertex.

Returns:

Error code.

Time complexity: O(|V|^3).

See also:

`igraph_maxflow_value()`, `igraph_edge_disjoint_paths()`, `igraph_edge_connectivity()`, `igraph_st_vertex_connectivity()`, `igraph_vertex_connectivity()`.

`igraph_edge_connectivity` — The minimum edge connectivity in a graph.

```
igraph_error_t igraph_edge_connectivity(const igraph_t *graph,
                                        igraph_int_t *res,
                                        igraph_bool_t checks);
```

This is the minimum of the edge connectivity over all pairs of vertices in the graph.

The edge connectivity of a graph is the same as group adhesion as defined in Douglas R. White and Frank Harary: The cohesiveness of blocks in social networks: node connectivity and conditional density, Sociological Methodology 31:305--359, 2001 https://doi.org/10.1111/0081-1750.00098.

Arguments:

`graph`: The input graph.

`res`: Pointer to an integer, the result will be stored here.

`checks`: Boolean 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.

Returns:

Error code.

Time complexity: O(log(|V|)*|V|^2) for undirected graphs and O(|V|^4) for directed graphs, but see also the discussion at the documentation of `igraph_maxflow_value()`.

See also:

`igraph_st_edge_connectivity()`, `igraph_maxflow_value()`, `igraph_vertex_connectivity()`.

`igraph_st_vertex_connectivity` — The vertex connectivity of a pair of vertices.

```
igraph_error_t igraph_st_vertex_connectivity(
        const igraph_t *graph,
        igraph_int_t *res,
        igraph_int_t source,
        igraph_int_t target,
        igraph_vconn_nei_t neighbors);
```

The vertex connectivity of two vertices (*source* and *target*) is the minimum number of vertices that must be deleted to eliminate all paths from *source* to *target*. Directed paths are considered in directed graphs.

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 current implementation uses maximum flow calculations to obtain the result.

Arguments:

graph: The input graph.

res: Pointer to an integer, the result will be stored here.

source: The id of the source vertex.

target: The id of the target vertex.

neighbors: A constant giving what to do if the two vertices are connected. Possible values: IGRAPH_VCONN_NEI_ERROR, stop with an error message, IGRAPH_VCONN_NEI_NEGATIVE, return -1. IGRAPH_VCONN_NEI_NUMBER_OF_NODES, return the number of nodes. IGRAPH_VCONN_NEI_IGNORE, ignore the fact that the two vertices are connected and calculate the number of vertices needed to eliminate all paths except for the trivial (direct) paths between *source* and *vertex*.

Returns:

Error code.

Time complexity: O(|V|^3), but see the discussion at igraph_maxflow_value().

See also:

igraph_vertex_connectivity(), igraph_edge_connectivity(), igraph_maxflow_value().

igraph_vertex_connectivity — The vertex connectivity of a graph.

```
igraph_error_t igraph_vertex_connectivity(
        const igraph_t *graph, igraph_int_t *res,
        igraph_bool_t checks);
```

The vertex connectivity of a graph is the minimum vertex connectivity along each pairs of vertices in the graph.

The vertex connectivity of a graph is the same as group cohesion as defined in Douglas R. White and Frank Harary: The cohesiveness of blocks in social networks: node connectivity and conditional density, Sociological Methodology 31:305--359, 2001 https://doi.org/10.1111/0081-1750.00098.

Arguments:

graph: The input graph.

res: Pointer to an integer, the result will be stored here.

checks: Boolean constant. Whether to check if the graph is connected or complete 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 1 then the vertex connectivity is also 1. If the graph is complete, the connectivity is the vertex count minus 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.

Returns:

Error code.

Time complexity: O(|V|^5).

See also:

`igraph_st_vertex_connectivity()`, `igraph_maxflow_value()`, and `igraph_edge_connectivity()`.

Edge- and vertex-disjoint paths

`igraph_edge_disjoint_paths` — The maximum number of edge-disjoint paths between two vertices.

```
igraph_error_t igraph_edge_disjoint_paths(const igraph_t *graph,
                                          igraph_int_t *res,
                                          igraph_int_t source,
                                          igraph_int_t target);
```

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.

Note that the number of disjoint paths is the same as the edge connectivity of the two vertices using uniform edge weights.

Arguments:

graph: The input graph, can be directed or undirected.

res: Pointer to an integer variable, the result will be stored here.

source: The id of the source vertex.

target: The id of the target vertex.

Returns:

Error code.

Time complexity: O(|V|^3), but see the discussion at `igraph_maxflow_value()`.

See also:

`igraph_vertex_disjoint_paths()`, `igraph_st_edge_connectivity()`, `igraph_maxflow_value()`.

`igraph_vertex_disjoint_paths` — Maximum number of vertex-disjoint paths between two vertices.

```
igraph_error_t igraph_vertex_disjoint_paths(const igraph_t *graph,
                                            igraph_int_t *res,
                                            igraph_int_t source,
                                            igraph_int_t target);
```

A set of paths between two vertices is called vertex-disjoint if they share no vertices, other than the endpoints. This function computes the largest number of such paths that can be constructed between a source and a target vertex. The calculation is performed by using maximum flow techniques.

When there are no edges from the source to 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.

Arguments:

`graph`: The input graph.

`res`: Pointer to an integer variable, the result will be stored here.

`source`: The id of the source vertex.

`target`: The id of the target vertex.

Returns:

Error code.

Time complexity: O(|V|^3).

See also:

`igraph_edge_disjoint_paths()`, `igraph_st_vertex_connectivity()`, `igraph_maxflow_value()`.

Graph adhesion and cohesion

`igraph_adhesion` — Graph adhesion, this is (almost) the same as edge connectivity.

```
igraph_error_t igraph_adhesion(const igraph_t *graph,
                               igraph_int_t *res,
                               igraph_bool_t checks);
```

This quantity is defined by White and Harary in The cohesiveness of blocks in social networks: node connectivity and conditional density, (Sociological Methodology 31:305--359, 2001) and basically it is the edge connectivity of the graph with uniform edge weights.

Arguments:

`graph`: The input graph, either directed or undirected.

res: Pointer to an integer, the result will be stored here.

checks: Boolean 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 adhesion is obviously zero. Otherwise if the minimum degree is one then the adhesion is also one. It is a good idea to perform these checks, as they can be done quickly compared to the edge connectivity calculation itself. They were suggested by Peter McMahan, thanks Peter. *

Returns:

Error code.

Time complexity: O(log(|V|)*|V|^2) for undirected graphs and O(|V|^4) for directed graphs, but see also the discussion at the documentation of `igraph_maxflow_value()`.

See also:

`igraph_cohesion()`, `igraph_maxflow_value()`, `igraph_edge_connectivity()`, `igraph_mincut_value()`.

`igraph_cohesion` — Graph cohesion, this is the same as vertex connectivity.

```
igraph_error_t igraph_cohesion(const igraph_t *graph,
                               igraph_int_t *res,
                               igraph_bool_t checks);
```

This quantity was defined by White and Harary in “The cohesiveness of blocks in social networks: node connectivity and conditional density”, (Sociological Methodology 31:305--359, 2001) and it is the same as the vertex connectivity of a graph.

Arguments:

graph: The input graph.

res: Pointer to an integer variable, the result will be stored here.

checks: Boolean 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 cohesion is obviously zero. Otherwise if the minimum degree is one then the cohesion is also one. It is a good idea to perform these checks, as they can be done quickly compared to the vertex connectivity calculation itself. They were suggested by Peter McMahan, thanks Peter.

Returns:

Error code.

Time complexity: O(|V|^4), |V| is the number of vertices. In practice it is more like O(|V|^2), see `igraph_maxflow_value()`.

See also:

`igraph_vertex_connectivity()`, `igraph_adhesion()`, `igraph_maxflow_value()`.

Cohesive blocks

`igraph_cohesive_blocks` — Identifies the hierarchical cohesive block structure of a graph.

```
igraph_error_t igraph_cohesive_blocks(const igraph_t *graph,
                           igraph_vector_int_list_t *blocks,
                           igraph_vector_int_t *cohesion,
                           igraph_vector_int_t *parent,
                           igraph_t *block_tree);
```

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 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 hiearchy of vertex subsets is found, with the entire graph G at its root.

This function implements cohesive blocking and calculates the complete cohesive block hierarchy of a graph.

See the following reference for details:

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. https://doi.org/10.2307/3088904

Arguments:

`graph`: The input graph. It must be undirected and simple. See `igraph_is_simple()`.

`blocks`: If not a null pointer, then it must be an initialized list of integers vectors; the cohesive blocks will be stored here. Each block is encoded with a vector of type `igraph_vector_int_t` that contains the vertex IDs of the block.

`cohesion`: If not a null pointer, then it must be an initialized vector and the cohesion of the blocks is stored here, in the same order as the blocks in the *`blocks`* vector list.

`parent`: If not a null pointer, then it must be an initialized vector and the block hierarchy is stored here. For each block, the ID (i.e. the position in the *`blocks`* vector list) of its parent block is stored. For the top block in the hierarchy, `-1` is stored.

`block_tree`: If not a null pointer, then it must be a pointer to an uninitialized graph, and the block hierarchy is stored here as an igraph graph. The vertex IDs correspond to the order of the blocks in the *`blocks`* vector.

Returns:

Error code.

Time complexity: TODO.

Example 23.6. File `examples/simple/cohesive_blocks.c`

Chapter 24. Vertex separators

`igraph_is_separator` — Would removing this set of vertices disconnect the graph?

```
igraph_error_t igraph_is_separator(const igraph_t *graph,
                                   const igraph_vs_t candidate,
                                   igraph_bool_t *res);
```

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.

Arguments:

graph: The input graph. It may be directed, but edge directions are ignored.

candidate: The candidate separator.

res: Pointer to a boolean variable, the result is stored here.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number vertices and edges.

Example 24.1. File `examples/simple/igraph_is_separator.c`

`igraph_is_minimal_separator` — Decides whether a set of vertices is a minimal separator.

```
igraph_error_t igraph_is_minimal_separator(const igraph_t *graph,
                                           const igraph_vs_t candidate,
                                           igraph_bool_t *res);
```

A vertex separator S is minimal is no proper subset of S is also a separator.

Arguments:

graph: The input graph. It may be directed, but edge directions are ignored.

candidate: The candidate minimal separators.

res: Pointer to a boolean variable, the result is stored here.

Returns:

Error code.

Time complexity: O(|V|+|E|), linear in the number vertices and edges.

Example 24.2. File `examples/simple/igraph_is_minimal_separator.c`

`igraph_all_minimal_st_separators` — List all vertex sets that are minimal (s,t) separators for some s and t.

```
igraph_error_t igraph_all_minimal_st_separators(
    const igraph_t *graph, igraph_vector_int_list_t *separators
);
```

This function lists all vertex sets that are minimal (s,t) separators for some (s,t) vertex pair.

Note that some vertex sets returned by this function may not be minimal with respect to disconnecting the graph (or increasing the number of connected components). Take for example the 5-vertex graph with edges `0-1-2-3-4-1`. This function returns the vertex sets `{1}`, `{2,4}` and `{1,3}`. Notice that `{1,3}` is not minimal with respect to disconnecting the graph, as `{1}` would be sufficient for that. However, it is minimal with respect to separating vertices `2` and `4`.

See more about the implemented algorithm in 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. https://doi.org/10.1007/3-540-46784-X_17

Arguments:

`graph`: The input graph. It may be directed, but edge directions are ignored.

`separators`: Pointer to a list of integer vectors, the separators will be stored here.

Returns:

Error code.

See also:

`igraph_minimum_size_separators()`

Time complexity: O(n|V|^3), |V| is the number of vertices, n is the number of separators.

Example 24.3. File `examples/simple/igraph_minimal_separators.c`

`igraph_minimum_size_separators` — Find all minimum size separating vertex sets.

```
igraph_error_t igraph_minimum_size_separators(
```

```
    const igraph_t *graph, igraph_vector_int_list_t *separators
);
```

This function lists all separator vertex sets of minimum size. A vertex set is a separator if its removal disconnects the graph.

If the graph is already disconnected, no separators are returned. Note that this convention differs from that used by some other funtions such as `igraph_all_minimal_st_separators()`.

Complete graphs have no vertex separators.

The implementation is based on the following paper: Arkady Kanevsky: Finding all minimum-size separating vertex sets in a graph, Networks 23, 533--541, 1993. https://doi.org/10.1002/net.3230230604

Arguments:

graph: The input graph, which must be undirected.

separators: An initialized list of integer vectors, the separators are stored here. It is a list of pointers to igraph_vector_int_t objects. Each vector will contain the IDs of the vertices in the separator. The separators are returned in an arbitrary order.

Returns:

Error code.

Time complexity: TODO.

Example 24.4. File `examples/simple/igraph_minimum_size_separators.c`

`igraph_even_tarjan_reduction` — Even-Tarjan reduction of a graph.

```
igraph_error_t igraph_even_tarjan_reduction(const igraph_t *graph, igraph_t *gra
        igraph_vector_t *capacity);
```

A digraph is created with twice as many vertices and edges. For each original vertex `i`, two vertices `i' = i` and `i'' = i' + n` are created, with a directed edge from `i'` to `i''`. For each original directed edge from `i` to `j`, two new edges are created, from `i'` to `j''` and from `i''` to `j'`.

This reduction is used in the paper (observation 2): Arkady Kanevsky: Finding all minimum-size separating vertex sets in a graph, Networks 23, 533--541, 1993.

The original paper where this reduction was conceived is Shimon Even and R. Endre Tarjan: Network Flow and Testing Graph Connectivity, SIAM J. Comput., 4(4), 507–518. https://doi.org/10.1137/0204043

Arguments:

graph: A graph. Although directness is not checked, this function is commonly used only on directed graphs.

graphbar: Pointer to a new directed graph that will contain the reduction, with twice as many vertices and edges.

capacity: Pointer to an initialized vector or a null pointer. If not a null pointer, then it will be filled the capacity from the reduction: the first |E| elements are 1, the remaining |E| are equal to |V| (which is used to indicate infinity).

Returns:

Error code.

Time complexity: O(|E|+|V|).

Example 24.5. File `examples/simple/even_tarjan.c`

Chapter 25. Detecting community structure

Community detection is concerned with clustering the vertices of networks into tightly connected subgraphs called "communities". The following references provide a good introduction to the topic of community detection:

S. Fortunato: "Community Detection in Graphs". Physics Reports 486, no. 3–5 (2010): 75–174. https://doi.org/10.1016/j.physrep.2009.11.002.

S. Fortunato and D. Hric: "Community Detection in Networks: A User Guide". Physics Reports 659 (2016): 1–44. https://doi.org/10.1016/j.physrep.2016.09.002.

Common functions related to community structure

`igraph_modularity` — Calculates the modularity of a graph with respect to some clusters or vertex types.

```
igraph_error_t igraph_modularity(const igraph_t *graph,
                      const igraph_vector_int_t *membership,
                      const igraph_vector_t *weights,
                      const igraph_real_t resolution,
                      const igraph_bool_t directed,
                      igraph_real_t *modularity);
```

The modularity of a graph with respect to some clustering of the vertices (or assignment of vertex types) measures how strongly separated the different clusters are from each other compared to a random null model. It is defined as

```
Q = 1/(2m) sum_ij (A_ij - # k_i k_j / (2m)) #(c_i,c_j),
```

where `m` is the number of edges, `A_ij` is the adjacency matrix, `k_i` is the degree of vertex `i`, `c_i` is the cluster that vertex `i` belongs to (or its vertex type), `#(i,j)=1` if `i=j` and 0 otherwise, and the sum goes over all `i`, `j` pairs of vertices. Note that in this formula, the diagonal of the adjacency matrix contains twice the number of self-loops.

The resolution parameter # 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 `# = 1`.

Modularity can also be calculated on directed graphs. This only requires a relatively modest change,

```
Q = 1/m sum_ij (A_ij - # k^out_i k^in_j / m) #(c_i,c_j),
```

where `k^out_i` is the out-degree of node `i` and `k^in_j` is the in-degree of node `j`.

Modularity on weighted graphs is also meaningful. When taking edge weights into account, `A_ij` equals the weight of the corresponding edge (or 0 if there is no edge), `k_i` is the strength (i.e. the weighted degree) of vertex `i`, with similar counterparts for a directed graph, and `m` is the total weight of all edges.

Note that the modularity is not well-defined for graphs with no edges. igraph returns `NaN` for graphs with no edges; see https://github.com/igraph/igraph/issues/1539 for a detailed discussion.

For the original definition of modularity, see Newman, M. E. J., and Girvan, M. (2004). Finding and evaluating community structure in networks. Physical Review E 69, 026113. https://doi.org/10.1103/PhysRevE.69.026113

For the directed definition of modularity, see Leicht, E. A., and Newman, M. E. J. (2008). Community Structure in Directed Networks. Physical Review Letters 100, 118703. https://doi.org/10.1103/PhysRevLett.100.118703

For the introduction of the resolution parameter #, see Reichardt, J., and Bornholdt, S. (2006). Statistical mechanics of community detection. Physical Review E 74, 016110. https://doi.org/10.1103/PhysRevE.74.016110

Arguments:

graph: The input graph.

membership: Numeric vector of integer values which gives the type of each vertex, i.e. the cluster to which it belongs. It does not have to be consecutive, i.e. empty communities are allowed. For better performance, ensure that community indices are nonnegative and smaller than the vertex count. This can be ensured using `igraph_reindex_membership()`.

weights: Weight vector or `NULL` if no weights are specified.

resolution: The resolution parameter #. Must not be negative. 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.

modularity: Pointer to a real number, the result will be stored here.

Returns:

Error code.

See also:

`igraph_modularity_matrix()`

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges, assuming that community indices are nonnegative and smaller than the vertex count. Otherwise, O(|V| log |V| + |E|).

`igraph_modularity_matrix` — Calculates the modularity matrix.

```
igraph_error_t igraph_modularity_matrix(const igraph_t *graph,
                            const igraph_vector_t *weights,
                            const igraph_real_t resolution,
                            igraph_matrix_t *modmat,
                            igraph_bool_t directed);
```

This function returns the modularity matrix, which is defined as

```
B_ij = A_ij - # k_i k_j / (2m)
```

for undirected graphs, where `A_ij` is the adjacency matrix, # is the resolution parameter, `k_i` is the degree of vertex `i`, and `m` is the number of edges in the graph. When there are no edges, or the weights add up to zero, the result is undefined.

For directed graphs the modularity matrix is changed to

```
B_ij = A_ij - # k^out_i k^in_j / m
```

where `k^out_i` is the out-degree of node `i` and `k^in_j` is the in-degree of node `j`.

Note that self-loops in undirected graphs are multiplied by 2 in this implementation. If weights are specified, the weighted counterparts of the adjacency matrix and degrees are used.

Arguments:

`graph`: The input graph.

`weights`: Edge weights, pointer to a vector. If this is a null pointer then every edge is assumed to have a weight of 1.

`resolution`: The resolution parameter #. Must not be negative. Default is 1. Lower values favor fewer, larger communities; higher values favor more, smaller communities.

`modmat`: Pointer to an initialized matrix in which the modularity matrix is stored.

`directed`: For directed graphs: if the edges should be treated as undirected. For undirected graphs this is ignored.

Returns:

Error code.

See also:

`igraph_modularity()`

`igraph_community_optimal_modularity` — Calculate the community structure with the highest modularity value.

```
igraph_error_t igraph_community_optimal_modularity(const igraph_t *graph,
                                                   const igraph_vector_t *weigh
                                                   const igraph_real_t resoluti
                                                   igraph_real_t *modularity,
                                                   igraph_vector_int_t *members
```

This function calculates the optimal community structure for a graph, in terms of maximal modularity score. Both undirected and directed graphs are supported.

The calculation is done by transforming the modularity maximization into an integer programming problem, and then calling the GLPK library to solve that. Please see Ulrik Brandes et al.: On Modularity Clustering, IEEE Transactions on Knowledge and Data Engineering 20(2):172-188, 2008 https://doi.org/10.1109/TKDE.2007.190689.

Note that exact 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.

Arguments:

graph: The input graph. It may be undirected or directed.

weights: Vector giving the weights of the edges. If it is NULL then each edge is supposed to have the same weight.

resolution: Resolution parameter. Must be greater than or equal to 0. Lower values favor fewer, larger communities; higher values favor more, smaller communities. Set it to 1 to use the classical definition of modularity.

modularity: Pointer to a real number, or a null pointer. If it is not a null pointer, then a optimal modularity value is returned here.

membership: Pointer to a vector, or a null pointer. If not a null pointer, then the membership vector of the optimal community structure is stored here.

Returns:

Error code. When GLPK is not available, IGRAPH_UNIMPLEMENTED is returned.

See also:

igraph_modularity(), igraph_community_fastgreedy() for an algorithm that finds a local optimum in a greedy way.

Time complexity: exponential in the number of vertices.

Example 25.1. File examples/simple/igraph_community_optimal_modularity.c

igraph_community_to_membership — Cut a dendrogram after a given number of merges.

```
igraph_error_t igraph_community_to_membership(const igraph_matrix_int_t *merges,
                                    igraph_int_t nodes,
                                    igraph_int_t steps,
                                    igraph_vector_int_t *membership,
                                    igraph_vector_int_t *csize);
```

This function creates a membership vector from a dendrogram whose leaves are individual vertices by cutting it at the specified level. It produces a membership vector that contains for each vertex its cluster ID, numbered from zero. This is the same membership vector format that is produced by igraph_connected_components(), as well as all community detection functions in igraph.

It takes as input the number of vertices *n*, and a *merges* matrix encoding the dendrogram, in the format produced by hierarchical clustering functions such as igraph_community_edge_betweenness(), igraph_community_walktrap() or igraph_community_fastgreedy(). The matrix must have two columns and up to n - 1 rows. Each row represents merging two dendrogram nodes into their parent node. The leaf nodes of the dendrogram are indexed from 0 to n - 1 and are identical to the vertices of the graph that is being partitioned into communities. Row i contains the children of dendrogram node with index n + i.

This function performs *steps* merge operations as prescribed by the *merges* matrix and returns the resulting partitioning into `n - steps` communities.

If *merges* is not a complete dendrogram, it is possible to take *steps* steps if *steps* is not bigger than the number lines in *merges*.

Arguments:

merges: The two-column matrix containing the merge operations.

nodes: The number of leaf nodes in the dendrogram.

steps: Integer constant, the number of steps to take.

membership: Pointer to an initialized vector, the membership results will be stored here, if not `NULL`. The vector will be resized as needed.

csize: Pointer to an initialized vector, or `NULL`. If not `NULL` then the sizes of the components will be stored here, the vector will be resized as needed.

Returns:

Error code.

See also:

`igraph_community_walktrap()`, `igraph_community_edge_betweenness()`, `igraph_community_fastgreedy()` for community structure detection algorithms producing merge matrices in this format; `igraph_le_community_to_membership()` to perform merges starting from a given cluster assignment.

Time complexity: O(|V|), the number of vertices in the graph.

`igraph_reindex_membership` — Makes the IDs in a membership vector contiguous.

```
igraph_error_t igraph_reindex_membership(
        igraph_vector_int_t *membership,
        igraph_vector_int_t *new_to_old,
        igraph_int_t *nb_clusters);
```

This function reindexes component IDs in a membership vector in a way that the new IDs start from zero and go up to `C-1`, where `C` is the number of unique component IDs in the original vector.

Arguments:

membership: Numeric vector which gives the type of each vertex, i.e. the component to which it belongs. The vector will be altered in-place.

new_to_old: Pointer to a vector which will contain the old component ID for each new one, or `NULL`, in which case it is not returned. The vector will be resized as needed.

nb_clusters: Pointer to an integer for the number of distinct clusters. If not `NULL`, this will be updated to reflect the number of distinct clusters found in membership.

Returns:

Error code.

Time complexity: Let n be the length of the membership vector. O(n) if cluster indices are within 0..n-1, and O(n log(n)) otherwise.

`igraph_compare_communities` — Compares community structures using various metrics.

```
igraph_error_t igraph_compare_communities(const igraph_vector_int_t *comm1,
                                  const igraph_vector_int_t *comm2, igraph_real_t*
                                  igraph_community_comparison_t method);
```

This function assesses the distance between two community structures using the variation of information (VI) metric of Meila (2003), the normalized mutual information (NMI) of Danon et al (2005), the split-join distance of van Dongen (2000), the Rand index of Rand (1971) or the adjusted Rand index of Hubert and Arabie (1985).

Some of these measures are defined based on the entropy of a discrete random variable associated with a given clustering `C` of vertices. Let `p_i` be the probability that a randomly picked vertex would be part of cluster `i`. Then the entropy of the clustering is

```
H(C) = - sum_i p_i log p_i
```

Similarly, we can define the joint entropy of two clusterings `C_1` and `C_2` based on the probability `p_ij` that a random vertex is part of cluster `i` in the first clustering and cluster `j` in the second one:

```
H(C_1, C_2) = - sum_ii p_ij log p_ij
```

The mutual information of `C_1` and `C_2` is then `MI(C_1, C_2) = H(C_1) + H(C_2) - H(C_1, C_2) >= 0` . A large mutual information indicates a high overlap between the two clusterings. The normalized mutual information, as computed by igraph, is

`NMI(C_1, C_2) = 2 MI(C_1, C_2) / (H(C_1) + H(C_2))`.

It takes its value from the interval (0, 1], with 1 achieved when the two clusterings coincide.

The variation of information is defined as `VI(C_1, C_2) = [H(C_1) - MI(C_1, C_2)] + [H(C_2) - MI(C_1, C_2)]`. Lower values of the variation of information indicate a smaller difference between the two clusterings, with `VI = 0` achieved precisely when they coincide. igraph uses natural units for the variation of information, i.e. it uses the natural logarithm when computing entropies.

The Rand index is defined as the probability that the two clusterings agree about the cluster memberships of a randomly chosen vertex *pair*. All vertex pairs are considered, and the two clusterings are considered to be in agreement about the memberships of a vertex pair if either the two vertices are in the same cluster in both clusterings, or they are in different clusters in both clusterings. The Rand index is then the number of vertex pairs in agreement, divided by the total number of vertex pairs. A Rand index of zero means that the two clusterings disagree about the membership of all vertex pairs, while 1 means that the two clusterings are identical.

The adjusted Rand index is similar to the Rand index, but it takes into account that agreement between the two clusterings may also occur by chance even if the two clusterings are chosen completely randomly. The adjusted Rand index therefore subtracts the expected fraction of agreements from the value of the Rand index, and divides the result by one minus the expected fraction of agreements. The maximum value of the adjusted Rand index is still 1 (similarly to the Rand index), indicating maximum agreement, but the value may be less than zero if there is *less* agreement between the two clusterings than what would be expected by chance.

For an explanation of the split-join distance, see `igraph_split_join_distance()`.

References:

Meil# M: Comparing clusterings by the variation of information. In: Schölkopf 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. https://doi.org/10.1007/978-3-540-45167-9_14

Danon L, Diaz-Guilera A, Duch J, Arenas A: Comparing community structure identification. J Stat Mech P09008, 2005. https://doi.org/10.1088/1742-5468/2005/09/P09008

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. https://ir.cwi.nl/pub/4461

Rand WM: Objective criteria for the evaluation of clustering methods. J Am Stat Assoc 66(336):846-850, 1971. https://doi.org/10.2307/2284239

Hubert L and Arabie P: Comparing partitions. Journal of Classification 2:193-218, 1985. https://doi.org/10.1007/BF01908075

Arguments:

comm1: the membership vector of the first community structure

comm2: the membership vector of the second community structure

result: the result is stored here.

method: the comparison method to use. IGRAPH_COMMCMP_VI selects the variation of information (VI) metric of Meila (2003), IGRAPH_COMMCMP_NMI selects the normalized mutual information measure proposed by Danon et al (2005), IGRAPH_COMMCMP_SPLIT_JOIN selects the split-join distance of van Dongen (2000), IGRAPH_COMMCMP_RAND selects the unadjusted Rand index (1971) and IGRAPH_COMMCMP_ADJUSTED_RAND selects the adjusted Rand index.

Returns:

Error code.

See also:

igraph_split_join_distance().

Time complexity: O(n log(n)).

igraph_split_join_distance — Calculates the split-join distance of two community structures.

```
igraph_error_t igraph_split_join_distance(const igraph_vector_int_t *comm1,
                                          const igraph_vector_int_t *comm2, igraph_int_t *
                                          igraph_int_t *distance21);
```

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:

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 in two arguments, `distance12` will contain the projection distance of the first partition from the second, while `distance21` will be 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.

Reference:

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.

Arguments:

comm1: the membership vector of the first community structure

comm2: the membership vector of the second community structure

distance12: pointer to an `igraph_int_t`, the projection distance of the first community structure from the second one will be returned here.

distance21: pointer to an `igraph_int_t`, the projection distance of the second community structure from the first one will be returned here.

Returns:

Error code.

See also:

`igraph_compare_communities()` with the `IGRAPH_COMMCMP_SPLIT_JOIN` method if you are not interested in the individual distances but only the sum of them.

Time complexity: O(n log(n)).

Community structure based on statistical mechanics

`igraph_community_spinglass` — Community detection based on statistical mechanics.

```
igraph_error_t igraph_community_spinglass(const igraph_t *graph,
                                const igraph_vector_t *weights,
                                igraph_real_t *modularity,
                                igraph_real_t *temperature,
                                igraph_vector_int_t *membership,
                                igraph_vector_int_t *csize,
                                igraph_int_t spins,
                                igraph_bool_t parupdate,
                                igraph_real_t starttemp,
                                igraph_real_t stoptemp,
                                igraph_real_t coolfact,
                                igraph_spincomm_update_t update_rule,
                                igraph_real_t gamma,
```

```
                                    igraph_spinglass_implementation_t implementation,
                                    igraph_real_t gamma_minus);
```

This function implements the community structure detection algorithm proposed by Joerg Reichardt and Stefan Bornholdt. The algorithm is described in their paper: Statistical Mechanics of Community Detection, http://arxiv.org/abs/cond-mat/0603718 .

From version 0.6, igraph also supports an extension to the algorithm that allows negative edge weights. This is described in V. A. Traag and Jeroen Bruggeman: Community detection in networks with positive and negative links, http://arxiv.org/abs/0811.2329 .

Arguments:

graph: The input graph, it may be directed but the direction of the edges is ignored by the algorithm.

weights: The vector giving the edge weights, it may be `NULL`, in which case all edges are weighted equally. The edge weights must be positive unless using the `IGRAPH_SPINCOMM_IMP_NEG` implementation.

modularity: Pointer to a real number, if not `NULL` then the modularity score of the solution will be stored here. This is the gereralized modularity, taking into account the resolution parameter *gamma*. See `igraph_modularity()` for details.

temperature: Pointer to a real number, if not `NULL` then the temperature at the end of the algorithm will be stored here.

membership: Pointer to an initialized vector or `NULL`. If not `NULL` then the result of the clustering will be stored here. For each vertex, the number of its cluster is given, with the first cluster numbered zero. The vector will be resized as needed.

csize: Pointer to an initialized vector or `NULL`. If not `NULL` then the sizes of the clusters will stored here in cluster number order. The vector will be resized as needed.

spins: Integer giving the number of spins, i.e. the maximum number of clusters. Even if the number of spins is high the number of clusters in the result might be small.

parupdate: A Boolean constant, whether to update all spins in parallel. It is not implemented in the `IGRAPH_SPINCOMM_INP_NEG` implementation.

starttemp: Real number, the temperature at the start. A reasonable default is 1.0.

stoptemp: Real number, the algorithm stops at this temperature. A reasonable default is 0.01.

coolfact: Real number, the cooling factor for the simulated annealing. A reasonable default is 0.99.

update_rule: The type of the update rule. Possible values: `IGRAPH_SPINCOMM_UPDATE_SIMPLE` and `IGRAPH_SPINCOMM_UPDATE_CONFIG`. Basically this parameter defines the null model based on which the actual clustering is done. If this is `IGRAPH_SPINCOMM_UPDATE_SIMPLE` then the random graph (i.e. G(n,p)), if it is `IGRAPH_SPINCOMM_UPDATE` then the configuration model is used. The configuration means that the baseline for the clustering is a random graph with the same degree distribution as the input graph.

gamma: Real number. The gamma parameter of the algorithm, acting as a resolution parameter. Smaller values typically lead to larger clusters, larger values typically lead to smaller clusters.

implementation: Constant, chooses between the two implementations of the spin-glass algorithm that are included in igraph. `IGRAPH_SPINCOMM_IMP_ORIG` selects the original implementation, this is faster, `IGRAPH_SPINCOMM_INP_NEG` selects an implementation that allows negative edge weights.

gamma_minus: Real number. Parameter for the `IGRAPH_SPINCOMM_IMP_NEG` implementation. This acts as a resolution parameter for the negative part of the network. Smaller values of *gamma_minus* leads to fewer negative edges within clusters. If this argument is set to zero, the algorithm reduces to a graph coloring algorithm when all edges have negative weights, using the number of spins as the number of colors.

Returns:

Error code.

See also:

`igraph_community_spinglass_single()` for calculating the community of a single vertex.

Time complexity: TODO.

`igraph_community_spinglass_single` — Community of a single node based on statistical mechanics.

```
igraph_error_t igraph_community_spinglass_single(const igraph_t *graph,
                                        const igraph_vector_t *weights,
                                        igraph_int_t vertex,
                                        igraph_vector_int_t *community,
                                        igraph_real_t *cohesion,
                                        igraph_real_t *adhesion,
                                        igraph_real_t *inner_links,
                                        igraph_real_t *outer_links,
                                        igraph_int_t spins,
                                        igraph_spincomm_update_t update_rule,
                                        igraph_real_t gamma);
```

This function implements the community structure detection algorithm proposed by Joerg Reichardt and Stefan Bornholdt. It is described in their paper: Statistical Mechanics of Community Detection, http://arxiv.org/abs/cond-mat/0603718 .

This function calculates the community of a single vertex without calculating all the communities in the graph.

Arguments:

graph: The input graph, it may be directed but the direction of the edges is not used in the algorithm.

weights: Pointer to a vector with the weights of the edges. Alternatively `NULL` can be supplied to have the same weight for every edge.

vertex: The vertex ID of the vertex of which ths community is calculated.

community: Pointer to an initialized vector, the result, the IDs of the vertices in the community of the input vertex will be stored here. The vector will be resized as needed.

cohesion: Pointer to a real variable, if not NULL the cohesion index of the community will be stored here.

adhesion: Pointer to a real variable, if not NULL the adhesion index of the community will be stored here.

inner_links: Pointer to a real, if not NULL the number of edges within the community (or the sum of their weights) is stored here.

outer_links: Pointer to a real, if not NULL the number of edges between the community and the rest of the graph (or the sum of their weights) will be stored here.

spins: The number of spins to use, this can be higher than the actual number of clusters in the network, in which case some clusters will contain zero vertices.

update_rule: The type of the update rule. Possible values: IGRAPH_SPINCOMM_UPDATE_SIMPLE and IGRAPH_SPINCOMM_UPDATE_CONFIG. Basically this parameter defined the null model based on which the actual clustering is done. If this is IGRAPH_SPINCOMM_UPDATE_SIMPLE then the random graph (ie. G(n,p)), if it is IGRAPH_SPINCOMM_UPDATE then the configuration model is used. The configuration means that the baseline for the clustering is a random graph with the same degree distribution as the input graph.

gamma: Real number. The gamma parameter of the algorithm. This defined the weight of the missing and existing links in the quality function for the clustering. The default value in the original code was 1.0, which is equal weight to missing and existing edges. Smaller values make the existing links contibute more to the energy function which is minimized in the algorithm. Bigger values make the missing links more important. (If my understanding is correct.)

Returns:

Error code.

See also:

igraph_community_spinglass() for the traditional version of the algorithm.

Time complexity: TODO.

Community structure based on eigenvectors of matrices

The function documented in these section implements the “leading eigenvector” method developed by Mark Newman and published in MEJ Newman: Finding community structure using the eigenvectors of matrices, Phys Rev E 74:036104 (2006).

The heart of the method is the definition of the modularity matrix `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_ij` 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. See `igraph_modularity_matrix()` for more details.

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 community structure. Check Newman's paper to understand why this is a good method for detecting community structure.

The leading eigenvector community structure detection method is implemented in `igraph_community_leading_eigenvector()`. After the initial split, the following splits are done in a way to optimize modularity regarding to the original network. Note that any further refinement, for example using Kernighan-Lin, as proposed in Section V.A of Newman (2006), is not implemented here.

Example 25.2. File `examples/simple/igraph_community_leading_eigenvector.c`

`igraph_community_leading_eigenvector` — Leading eigenvector community finding (proper version).

```
igraph_error_t igraph_community_leading_eigenvector(
        const igraph_t *graph,
        const igraph_vector_t *weights,
        igraph_matrix_int_t *merges,
        igraph_vector_int_t *membership,
        igraph_int_t steps,
        igraph_arpack_options_t *options,
        igraph_real_t *modularity,
        igraph_bool_t start,
        igraph_vector_t *eigenvalues,
        igraph_vector_list_t *eigenvectors,
        igraph_vector_int_t *history,
        igraph_community_leading_eigenvector_callback_t *callback,
        void *callback_extra);
```

Newman's leading eigenvector method for detecting community structure. This is the proper implementation of the recursive, divisive algorithm: each split is done by maximizing the modularity regarding the original network, see MEJ Newman: Finding community structure in networks using the eigenvectors of matrices, Phys Rev E 74:036104 (2006). https://doi.org/10.1103/PhysRevE.74.036104

Arguments:

`graph`: The input graph. Edge directions will be ignored.

`weights`: The weights of the edges, or `NULL` for unweighted graphs.

`merges`: The result of the algorithm, a matrix containing the information about the splits performed. The matrix is built in the opposite way however, it is like the result of an agglomerative algorithm. Unlike with most other hierarchical community detection functions in igraph, the integers in this matrix represent community indices, not vertex indices. If at the end of the algorithm (after *`steps`* steps was done) there are “p” communities, then these are numbered from zero to `p-1`. The first line of the matrix contains the first “merge” (which is in reality the last split) of two communities into community `p`, the merge in the second line forms community `p+1`, etc. The matrix should be initialized before calling and will be resized as needed. This argument is ignored if it is `NULL`.

`membership`: The membership of the vertices after all the splits were performed will be stored here. The vector must be initialized before calling and will be resized as needed. This argument is ignored if it is `NULL`. This argument can also be used to supply a starting configuration for the community finding, in the

format of a membership vector. In this case the *start* argument must be set to true.

steps: The maximum number of steps to perform. It might happen that some component (or the whole network) has no underlying community structure and no further steps can be done. If you want as many steps as possible then supply the number of vertices in the network here.

options: The options for ARPACK. Supply NULL here to use the defaults. n is always overwritten. ncv is set to at least 4.

modularity: If not a null pointer, then it must be a pointer to a real number and the modularity score of the final division is stored here.

start: Boolean, whether to use the community structure given in the *membership* argument as a starting point.

eigenvalues: Pointer to an initialized vector or a null pointer. If not a null pointer, then the eigenvalues calculated along the community structure detection are stored here. The non-positive eigenvalues, that do not result a split, are stored as well.

eigenvectors: If not a null pointer, then the eigenvectors that are calculated in each step of the algorithm are stored here, in a list of vectors. Each eigenvector is stored in an igraph_vector_t object.

history: Pointer to an initialized vector or a null pointer. If not a null pointer, then a trace of the algorithm is stored here, encoded numerically. The various operations:

IGRAPH_LEVC_HIST_START_FULL	Start the algorithm from an initial state where each connected component is a separate community.
IGRAPH_LEVC_HIST_START_GIVEN	Start the algorithm from a given community structure. The next value in the vector contains the initial number of communities.
IGRAPH_LEVC_HIST_SPLIT	Split a community into two communities. The id of the splitted community is given in the next element of the history vector. The id of the first new community is the same as the id of the splitted community. The id of the second community equals to the number of communities before the split.
IGRAPH_LEVC_HIST_FAILED	Tried to split a community, but it was not worth it, as it does not result in a bigger modularity value. The id of the community is given in the next element of the vector.

callback: A null pointer or a function of type igraph_community_leading_eigenvector_callback_t. If given, this callback function is called after each eigenvector/eigenvalue calculation. If the callback returns IGRAPH_STOP, then the community finding algorithm stops. If it returns IGRAPH_SUCCESS, the algorithm continues normally. Any other return value is considered an igraph error code and will terminete the algo-

rithm with the same error code. See the arguments passed to the callback at the documentation of `igraph_community_leading_eigenvector_callback_t`.

`callback_extra`: Extra argument to pass to the callback function.

Returns:

Error code.

See also:

`igraph_community_walktrap()` and `igraph_community_spinglass()` for other community structure detection methods.

Time complexity: O(|E|+|V|^2*steps), |V| is the number of vertices, |E| the number of edges, “steps” the number of splits performed.

`igraph_community_leading_eigenvector_callback_t` — Callback for the leading eigenvector community finding method.

```
typedef igraph_error_t igraph_community_leading_eigenvector_callback_t(
    const igraph_vector_int_t *membership,
    igraph_int_t comm,
    igraph_real_t eigenvalue,
    const igraph_vector_t *eigenvector,
    igraph_arpack_function_t *arpack_multiplier,
    void *arpack_extra,
    void *extra);
```

The leading eigenvector community finding implementation in igraph is able to call a callback function, after each eigenvalue calculation. This callback function must be of `igraph_community_leading_eigenvector_callback_t` type. The following arguments are passed to the callback:

Arguments:

`membership`: The actual membership vector, before recording the potential change implied by the newly found eigenvalue.

`comm`: The id of the community that the algorithm tried to split in the last iteration. The community IDs are indexed from zero here!

`eigenvalue`: The eigenvalue the algorithm has just found.

`eigenvector`: The eigenvector corresponding to the eigenvalue the algorithm just found.

`arpack_multiplier`: A function that was passed to `igraph_arpack_rssolve()` to solve the last eigenproblem.

`arpack_extra`: The extra argument that was passed to the ARPACK solver.

`extra`: Extra argument that as passed to `igraph_community_leading_eigenvector()`.

See also:

`igraph_community_leading_eigenvector()`, `igraph_arpack_function_t`, `igraph_arpack_rssolve()`.

`igraph_le_community_to_membership` — Cut an incomplete dendrogram after a given number of merges, starting with an initial cluster assignment.

```
igraph_error_t igraph_le_community_to_membership(const igraph_matrix_int_t *merg
                                                 igraph_int_t steps,
                                                 igraph_vector_int_t *membershi
                                                 igraph_vector_int_t *csize);
```

This function takes a dendrogram whose leaves are cluster IDs given in an initial cluster assignment provided in *`membership`*. Then it updates the cluster assignment by performing the specified number of mergers, as given by the dendrogram encoded in *`merges`*. It is a more general version of `igraph_community_to_membership()`, which assumes that the dendrogram leaves are singleton clusters corresponding to individual vertices.

This dendrogram format is suitable for divise hierarchical community detection algorithms that stop before dividing the graph into individual vertices, such as `igraph_community_leading_eigenvector()`.

Initially, *`membership`* is expected to contain `m` contiguous cluster indices, numbered from zero. These correspond to the leaf nodes of the dendrogram. Row `i` of the two-column *`merges`* matrix contains the IDs of clusters that are merged together into dendrogram node `m + i`. It may have up to `m - 1` rows.

This function performs *`steps`* merge operations as prescribed by the *`merges`* matrix and updates *`membership`* to the resulting partitioning into `m - steps` communities.

Arguments:

`merges`: The two-column matrix containing the merge operations. See `igraph_community_leading_eigenvector()` for the detailed syntax. This is usually from the output of the leading eigenvector community structure detection routines.

`steps`: The number of steps to make according to `merges`.

`membership`: Initially the starting membership vector, on output the resulting membership vector, after performing `steps` merges.

`csize`: Optionally the sizes of the communities are stored here, if this is not a null pointer, but an initialized vector.

Returns:

Error code.

See also:

`igraph_community_to_membership()` for a simpler interface that starts by merging individual vertices.

Time complexity: O(|V|), the number of vertices.

Walktrap: Community structure based on random walks

`igraph_community_walktrap` — Community finding using a random walk based similarity measure.

```
igraph_error_t igraph_community_walktrap(const igraph_t *graph,
                                         const igraph_vector_t *weights,
                                         igraph_int_t steps,
                                         igraph_matrix_int_t *merges,
                                         igraph_vector_t *modularity,
                                         igraph_vector_int_t *membership);
```

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

Currently the original C++ implementation is used in igraph, see https://www-complexnetworks.lip6.fr/~latapy/PP/walktrap.html We are grateful to Matthieu Latapy and Pascal Pons for providing this source code.

In contrast to the original implementation, isolated vertices are allowed in the graph and they are assumed to have a single incident loop edge with weight 1.

Arguments:

`graph`: The input graph, edge directions are ignored.

`weights`: Numeric vector giving the weights of the edges. If it is a NULL pointer then all edges will have equal weights. The weights are expected to be positive.

`steps`: Integer constant, the length of the random walks. Typically, good results are obtained with values between 3-8 with 4-5 being a reasonable default.

`merges`: Pointer to a matrix, the merges performed by the algorithm will be stored here (if not `NULL`). Each merge is a row in a two-column matrix and contains the IDs of the merged clusters. Clusters are numbered from zero and cluster numbers smaller than the number of nodes in the network belong to the individual vertices as singleton clusters. In each step a new cluster is created from two other clusters and its id will be one larger than the largest cluster id so far. This means that before the first merge we have `n` clusters (the number of vertices in the graph) numbered from zero to `n - 1`. The first merge creates cluster `n`, the second cluster `n + 1`, etc.

`modularity`: Pointer to a vector. If not `NULL` then the modularity score of the current clustering is stored here after each merge operation.

`membership`: Pointer to a vector. If not a `NULL` pointer, then the membership vector corresponding to the maximal modularity score is stored here.

Returns:

Error code.

See also:

`igraph_community_spinglass()`, `igraph_community_edge_betweenness()`.

Time complexity: O(|E||V|^2) in the worst case, O(|V|^2 log|V|) typically, |V| is the number of vertices, |E| is the number of edges.

Example 25.3. File `examples/simple/walktrap.c`

Edge betweenness based community detection

`igraph_community_edge_betweenness` — Community finding based on edge betweenness.

```
igraph_error_t igraph_community_edge_betweenness(const igraph_t *graph,
                                        igraph_vector_int_t *removed_edges,
                                        igraph_vector_t *edge_betweenness,
                                        igraph_matrix_int_t *merges,
                                        igraph_vector_int_t *bridges,
                                        igraph_vector_t *modularity,
                                        igraph_vector_int_t *membership,
                                        igraph_bool_t directed,
                                        const igraph_vector_t *weights,
                                        const igraph_vector_t *lengths);
```

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

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.

In directed graphs, when *`directed`* is set to true, the directed version of betweenness and modularity are used, however, only splits into *weakly* connected components are detected.

When edge weights are given, the ratio of betweenness and weight values is used to choose which edges to remove first, as described in M. E. J. Newman: Analysis of Weighted Networks (2004), Section C. Thus, edges with large weights are treated as strong connections, and will be removed later than weak connections having similar betweenness. Weights are also used for calculating modularity.

If lengths are given, they will be considered for shortest path length calculations while computing betweenness values.

Note: In igraph 0.10, this function interpreted weights in a different, erroneous way, and issued a warning when weights were used. Please see https://github.com/igraph/igraph/issues/2229 for additional details.

References:

M. Girvan and M. E. J. Newman, Community Structure in Social and Biological Networks, PNAS 99, 7821 (2002). https://doi.org/10.1073/pnas.122653799

M. E. J. Newman, Analysis of Weighted Networks, Phys. Rev. E 70, 9 (2004). https://doi.org/10.1103/PhysRevE.70.056131

Arguments:

graph: The input graph.

removed_edges: Pointer to an initialized integer vector, which will be resized as needed. The IDs of the removed edges in the order of their removal will be stored here. This vector is suitable as input to `igraph_community_eb_get_merges()`. This parameter may be `NULL` if the edge IDs are not needed by the caller.

edge_betweenness: Pointer to an initialized vector or `NULL`. In the former case the edge betweenness of the removed edges is stored here. The vector will be resized as needed. Note that the betweenness values stored here are *not* divided by weights.

merges: Pointer to an initialized matrix or `NULL`. If not `NULL` then merges performed by the algorithm are stored here. Even if this is a divisive algorithm, we can replay it backwards and note which two clusters were merged. Clusters are numbered from zero. See `igraph_community_to_membership()` for details. The matrix will be resized as needed.

bridges: Pointer to an initialized vector of `NULL`. If not `NULL` then the indices into *`result`* of all edges which caused one of the *`merges`* will be put here. This is equivalent to all edge removals which separated the network into more components, in reverse order.

modularity: If not a null pointer, then the modularity values of the different divisions are stored here, in the order corresponding to the merge matrix. The modularity values will take weights into account if *`weights`* is not null.

membership: If not a null pointer, then the membership vector, corresponding to the highest modularity value, is stored here.

directed: Boolean constant. Controls whether to calculate directed betweenness (i.e. directed paths) for directed graphs, and whether to use the directed version of modularity. It is ignored for undirected graphs.

weights: An optional vector containing edge weights. If not `NULL`, the weights will be used to divide the edge betweenness scores, as well as for the calculation of modularity.

lengths: An optional vector containing edge lengths. If not `NULL`, path lengths used in the betweenness calculation will take these lengths into account.

Returns:

Error code.

See also:

`igraph_community_eb_get_merges()`, `igraph_community_spinglass()`, `igraph_community_walktrap()`.

Time complexity: O(|V||E|^2), as the betweenness calculation requires O(|V||E|) and we do it |E|-1 times.

Example 25.4. File `examples/simple/igraph_community_edge_betweenness.c`

`igraph_community_eb_get_merges` — Calculating the merges, i.e. the dendrogram for an edge betweenness community structure.

```
igraph_error_t igraph_community_eb_get_merges(const igraph_t *graph,
                                      const igraph_bool_t directed,
                                      const igraph_vector_int_t *edges,
                                      const igraph_vector_t *weights,
                                      igraph_matrix_int_t *res,
                                      igraph_vector_int_t *bridges,
                                      igraph_vector_t *modularity,
                                      igraph_vector_int_t *membership);
```

This function is handy if you have a sequence of edges which are gradually removed from the network and you would like to know how the network falls apart into separate components. The edge sequence may come from the `igraph_community_edge_betweenness()` function, but this is not necessary. Note that `igraph_community_edge_betweenness()` can also calculate the dendrogram, via its *`merges`* argument. Merges happen when the edge removal process is run backwards and two components become connected.

Arguments:

`graph`: The input graph.

`directed`: Whether to use the directed or undirected version of modularity. Will be ignored for undirected graphs.

`edges`: Vector containing the edges to be removed from the network, all edges are expected to appear exactly once in the vector.

`weights`: An optional vector containing edge weights. If null, the unweighted modularity scores will be calculated. If not null, the weighted modularity scores will be calculated. Ignored if both *`modularity`* and *`membership`* are `NULL` pointers.

`res`: Pointer to an initialized matrix, if not `NULL` then the dendrogram will be stored here, in the same form as for the `igraph_community_walktrap()` function: the matrix has two columns and each line is a merge given by the IDs of the merged components. The component IDs are numbered from zero and component IDs smaller than the number of vertices in the graph belong to individual vertices. The non-trivial components containing at least two vertices are numbered from `n`, where `n` is the number of vertices in the graph. So if the first line contains `a` and `b` that means that components `a` and `b` are merged into component `n`, the second line creates component `n + 1`, etc. The matrix will be resized as needed.

`bridges`: Pointer to an initialized vector of `NULL`. If not `NULL` then the indices into *`edges`* of all edges which caused one of the merges will be put here. This is equal to all edge removals which separated the network into more components, in reverse order.

`modularity`: If not a null pointer, then the modularity values for the different divisions, corresponding to the merges matrix, will be stored here.

`membership`: If not a null pointer, then the membership vector for the best division (in terms of modularity) will be stored here.

Returns:

Error code.

See also:

`igraph_community_edge_betweenness()`.

Time complexity: O(|E|+|V|log|V|), |V| is the number of vertices, |E| is the number of edges.

Community structure based on the optimization of modularity

`igraph_community_fastgreedy` — Finding community structure by greedy optimization of modularity.

```
igraph_error_t igraph_community_fastgreedy(const igraph_t *graph,
                                 const igraph_vector_t *weights,
                                 igraph_matrix_int_t *merges,
                                 igraph_vector_t *modularity,
                                 igraph_vector_int_t *membership);
```

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.

Some improvements proposed in K Wakita, T Tsurumi: Finding community structure in mega-scale social networks, http://www.arxiv.org/abs/cs.CY/0702048v1 have also been implemented.

Arguments:

`graph`: The input graph. It must be a graph without multiple edges. This is checked and an error message is given for graphs with multiple edges.

`weights`: Potentially a numeric vector containing edge weights. Supply a null pointer here for unweighted graphs. The weights are expected to be non-negative.

`merges`: Pointer to an initialized matrix or `NULL`, the result of the computation is stored here as a merges matrix representing a dendrogram. The matrix has two columns and each merge corresponds to one merge, the IDs of the two merged components are stored. The component IDs are numbered from zero and the first `n` components are the individual vertices, `n` is the number of vertices in the graph. Component `n` is created in the first merge, component `n+1` in the second merge, etc. The matrix will be resized as needed. If this argument is `NULL` then it is ignored completely.

`modularity`: Pointer to an initialized vector or `NULL` pointer, in the former case the modularity scores along the stages of the computation are recorded here. The vector will be resized as needed.

`membership`: Pointer to a vector. If not a null pointer, then the membership vector corresponding to the best split (in terms of modularity) is stored here.

Returns:

Error code.

See also:

`igraph_community_to_membership()` to cut the dendrogram at an arbitrary number of steps.

Time complexity: O(|E||V|log|V|) in the worst case, O(|E|+|V|log^2|V|) typically, |V| is the number of vertices, |E| is the number of edges.

Example 25.5. File `examples/simple/igraph_community_fastgreedy.c`

`igraph_community_multilevel` — Finding community structure by multi-level optimization of modularity (Louvain).

```
igraph_error_t igraph_community_multilevel(const igraph_t *graph,
                                           const igraph_vector_t *weights,
                                           const igraph_real_t resolution,
                                           igraph_vector_int_t *membership,
                                           igraph_matrix_int_t *memberships,
                                           igraph_vector_t *modularity);
```

This function implements a multi-level modularity optimization algorithm for finding community structure, sometimes known as the Louvain algorithm.

The algorithm 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: in a random order, 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.

The resolution parameter # allows finding communities at different resolutions. Higher values of the resolution parameter typically result in more, smaller communities. Lower values typically result in fewer, larger communities. The original definition of modularity is retrieved when setting `#=1`. Note that the returned modularity value is calculated using the indicated resolution parameter. See `igraph_modularity()` for more details.

The original version of this function was contributed by Tom Gregorovic.

Reference:

Blondel, V. D., Guillaume, J.-L., Lambiotte, R., & Lefebvre, E.: Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment, 10008(10), 6 (2008). https://doi.org/10.1088/1742-5468/2008/10/P10008

Arguments:

`graph`: The input graph. It must be an undirected graph.

`weights`: Numeric vector containing edge weights. If `NULL`, every edge has equal weight. The weights are expected to be non-negative.

`resolution`: Resolution parameter. Must be greater than or equal to 0. Lower values favor fewer, larger communities; higher values favor more, smaller communities. Set it to 1 to use the classical definition of modularity.

`membership`: The membership vector, the result is returned here. For each vertex it gives the ID of its community. The vector must be initialized and it will be resized accordingly.

memberships: Numeric matrix that will contain the membership vector after each level, if not `NULL`. It must be initialized and it will be resized accordingly.

modularity: Numeric vector that will contain the modularity score after each level, if not `NULL`. It must be initialized and it will be resized accordingly.

Returns:

Error code.

Time complexity: in average near linear on sparse graphs.

Example 25.6. File `examples/simple/igraph_community_multilevel.c`

`igraph_community_leiden` — Finding community structure using the Leiden algorithm.

```
igraph_error_t igraph_community_leiden(
        const igraph_t *graph,
        const igraph_vector_t *edge_weights,
        const igraph_vector_t *vertex_out_weights,
        const igraph_vector_t *vertex_in_weights,
        igraph_real_t resolution,
        igraph_real_t beta,
        igraph_bool_t start,
        igraph_int_t n_iterations,
        igraph_vector_int_t *membership,
        igraph_int_t *nb_clusters,
        igraph_real_t *quality);
```

This function implements the Leiden algorithm for finding community structure.

It is similar to the multilevel algorithm, often called the Louvain algorithm, 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 Traag, Van Dooren & Nesterov).

The Leiden algorithm consists of three phases: (1) local moving of vertices, (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 vertices whose neighborhood has changed are visited. Only moves that strictly improve the quality function are made. 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 vertices (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 (weakly) connected and well-separated. After an iteration in which nothing has changed, all vertices and some parts are guaranteed to be locally optimally assigned. Note that even if a single iteration did not result in any change, it is still possible that a subsequent iteration might find some improvement. Each iteration explores different subsets of vertices to consider for moving from one cluster to another. 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

```
1 / 2m sum_ij (A_ij - # n_i n_j) #(s_i, s_j)
```

in the undirected case and

```
1 / m sum_ij (A_ij - # n^out_i n^in_j) #(s_i, s_j)
```

in the directed case. Here `m` is the total edge weight, `A_ij` is the weight of edge (i, j), # is the so-called resolution parameter, `n_i` is the vertex weight of vertex `i` (separate out- and in-weights are used with directed graphs), `s_i` is the cluster of vertex `i` and `#(x, y) = 1` if and only if `x = y` and 0 otherwise.

By setting `n_i = k_i`, the degree of vertex `i`, and dividing # by `2m` (by `m` in the directed case), we effectively obtain an expression for modularity. Hence, the standard modularity will be optimized when you supply the degrees (out- and in-degrees with directed graphs) as the vertex weights and by supplying as a resolution parameter `1/(2m)` (`1/m` with directed graphs). Use the `igraph_community_leiden_simple()` convenience function to compute vertex weights automatically for modularity maximization.

References:

V. A. Traag, L. Waltman, N. J. van Eck: From Louvain to Leiden: guaranteeing well-connected communities. Scientific Reports, 9(1), 5233 (2019). http://dx.doi.org/10.1038/s41598-019-41695-z

V. A. Traag, P. Van Dooren, and Y. Nesterov: Narrow scope for resolution-limit-free community detection. Phys. Rev. E 84, 016114 (2011). https://doi.org/10.1103/PhysRevE.84.016114

Arguments:

`graph`:	The input graph.
`edge_weights`:	Numeric vector containing edge weights. If `NULL`, every edge has equal weight of 1. The weights need not be non-negative.
`vertex_out_weights`:	Numeric vector containing vertex weights, or vertex out-weights for directed graphs. If `NULL`, every vertex has equal weight of 1.
`vertex_in_weights`:	Numeric vector containing vertex in-weights for directed graphs. If set to `NULL`, in-weights are assumed to be the same as out-weights, which effectively ignores edge directions. Must be `NULL` for undirected graphs.
`n_iterations`:	Iterate the core Leiden algorithm the indicated number of times. If this is a negative number, it will continue iterating until an iteration did not change the clustering. Two iterations are often sufficient, thus 2 is a reasonable default.
`beta`:	The randomness used in the refinement step when merging. A small amount of randomness (`beta` = 0.01) typically works well.
`start`:	Start from membership vector. If this is true, the optimization will start from the provided membership vector. If this is false, the optimization will start from a singleton partition.
`n_iterations`:	Iterate the core Leiden algorithm for the indicated number of times. If this is a negative number, it will continue iterating until an iteration did not change the clustering.
`membership`:	The membership vector. This is both used as the initial membership from which optimisation starts and is updated in place. It must hence be properly initialized. When finding clusters from scratch it is typically started using a singleton clustering. This can be achieved using `igraph_vector_int_init_range()`.

nb_clusters: The number of clusters contained in the final *membership*. If NULL, the number of clusters will not be returned.

quality: The quality of the partition, in terms of the objective function as included in the documentation. If NULL the quality will not be calculated.

Returns:

Error code.

Time complexity: near linear on sparse graphs.

See also:

igraph_community_leiden_simple() for a simplified interface that allows specifying an objective function directly and does not require vertex weights.

Example 25.7. File examples/simple/igraph_community_leiden.c

igraph_community_leiden_simple — Finding community structure using the Leiden algorithm, simple interface.

```
igraph_error_t igraph_community_leiden_simple(
        const igraph_t *graph,
        const igraph_vector_t *weights,
        igraph_leiden_objective_t objective,
        igraph_real_t resolution,
        igraph_real_t beta,
        igraph_bool_t start,
        igraph_int_t n_iterations,
        igraph_vector_int_t *membership,
        igraph_int_t *nb_clusters,
        igraph_real_t *quality);
```

This is a simplified interface to igraph_community_leiden() for convenience purposes. Instead of requiring vertex weights, it allows choosing from a set of objective functions to maximize. It implements these objective functions by passing suitable vertex weights to igraph_community_leiden(), as explained in the documentation of that function.

Arguments:

graph: The input graph. May be directed or undirected.

weights: The edge weights. If NULL, all weights are assumed to be 1.

objective: The objective function to maximize.

IGRAPH_LEIDEN_OBJECTIVE_MODULARITY	Use the generalized modularity, defined as Q = 1/(2m) sum_ij (A_ij - # k_i k_j / (2m)) #(c_i, c_j) for undirected graphs and as Q = 1/m sum_ij (A_ij - # k^out_i k^in_j / m) #(c_i, c_j) for directed graphs. This effectively uses a

multigraph configuration model as the null model. Edge weights must not be negative.

`IGRAPH_LEIDEN_OBJECTIVE_CPM`: Use the constant Potts model, whose objective function is defined as `Q = 1/(2m) sum_ij (A_ij - #) #(c_i, c_j)` for undirected graphs and as `Q = 1/m sum_ij (A_ij - #) #(c_i, c_j)` for directed graphs. Edge weights are allowed to be negative. Edge directions have no impact on the result.

`IGRAPH_LEIDEN_OBJECTIVE_ER`: Use an objective function based on the multigraph Erd#s-Rényi G(n,p) null model, defined as `Q = 1/(2m) sum_ij (A_ij - # p) #(c_i, c_j)` for undirected graphs and as `Q = 1/m sum_ij (A_ij - # p) #(c_i, c_j)` for directed graphs. `p` is the weighted density, i.e. the average link strength between all vertex pairs (whether adjacent or not). Edge weights must not be negative. Edge directions have no impact on the result.

In the above formulas, `A` is the adjacency matrix, `m` is the total edge weight, `k` are the (out- and in-) degrees, `#` is the resolution parameter, and `#(c_i, c_j)` is 1 if vertices `i` and `j` are in the same community and 0 otherwise. Edge directions are only relevant with `IGRAPH_LEIDEN_OBJECTIVE_MODULARITY`. The other two objective functions are equivalent between directed and undirected graphs: the formal difference is due to each edge being included twice in undirected (symmetric) adjacency matrices.

`resolution`: The resolution parameter, which is represented by γ in the objective functions detailed above.

`beta`: The randomness used in the refinement step when merging. A small amount of randomness (`beta` = 0.01) typically works well.

`start`: Start from membership vector. If this is true, the optimization will start from the provided membership vector. If this is false, the optimization will start from a singleton partition.

`n_iterations`: Iterate the core Leiden algorithm the indicated number of times. If this is a negative number, it will continue iterating until an iteration did not change the clustering. Two iterations are often sufficient, thus 2 is a reasonable default.

`membership`: The membership vector. If *`start`* is set to `false`, it will be resized appropriately. If *`start`* is `true`, it must be a valid membership vector for the given *`graph`*.

`nb_clusters`: The number of clusters contained in the final *`membership`*. If `NULL`, the number of clusters will not be returned.

`quality`: The quality of the partition, in terms of the objective function selected by *`objective`*. If `NULL` the quality will not be calculated.

Returns:

Error code.

Time complexity: near linear on sparse graphs.

See also:

`igraph_community_leiden()` for a more flexible interface that allows specifying raw vertex weights.

Fluid communities

`igraph_community_fluid_communities` — Community detection based on fluids interacting on the graph.

```
igraph_error_t igraph_community_fluid_communities(
        const igraph_t *graph,
        igraph_int_t no_of_communities,
        igraph_vector_int_t *membership);
```

The algorithm is 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. Weighted graphs are not supported.

This function implements the community detection method described in: 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. https://doi.org/10.1007/978-3-319-72150-7_19

Arguments:

`graph`: The input graph. The graph must be simple and connected. Edge directions will be ignored.

`no_of_communities`: The number of communities to be found. Must be greater than 0 and fewer than number of vertices in the graph.

`membership`: The result vector mapping vertices to the communities they are assigned to.

Returns:

Error code.

Time complexity: O(|E|)

Label propagation

`igraph_community_label_propagation` — Community detection based on label propagation.

```
igraph_error_t igraph_community_label_propagation(const igraph_t *graph,
        igraph_vector_int_t *membership,
```

```
igraph_neimode_t mode,
const igraph_vector_t *weights,
const igraph_vector_int_t *initial,
const igraph_vector_bool_t *fixed,
igraph_lpa_variant_t lpa_variant);
```

This function implements the label propagation-based community detection algorithm described by Raghavan, Albert and Kumara (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. In addition, it implements the fast label propagation alternative introduced by Traag and Šubelj (2023).

The algorithm works by iterating over nodes and updating the label of a node based on the labels of its neighbors. The labels that are most frequent among the neighbors are said to be dominant labels. The label of a node is always updated to a dominant label. The algorithm guarantees that the label for each is dominant when it terminates.

There are several variants implemented, which work slightly differently with the dominance of labels. Nodes with a dominant label might no longer have a dominant label if one of their neighbors change label. In `IGRAPH_LPA_DOMINANCE` an additional iteration over all nodes is made after updating all labels to double check whether all nodes indeed have a dominant label. When updating the label of a node, labels are always sampled from among all dominant labels. The algorithm stops when all nodes have dominant labels. In `IGRAPH_LPA_RETENTION` instead labels are only updated when they are not dominant. That is, they retain their current label whenever the current label is already dominant. The algorithm does not make an additional iteration to check for dominance. Instead, it simply keeps track whether a label has been updated, and terminates if no updates have been made. In `IGRAPH_LPA_FAST` labels are sampled from among all dominant labels, similar to `IGRAPH_LPA_DOMINANCE`. Instead of iterating over all nodes, it keeps track of a queue of nodes that should be considered. Nodes are popped from the queue when they are considered for update. When the label of a node is updated, the node's neighbors are added to the queue again (if they weren't already in the queue). The algorithm terminates when the queue is empty. All variants guarantee that the labels for all nodes are dominant.

Weights are taken into account as follows: when the new label of node `i` is determined, the algorithm iterates over all edges incident on node `i` and calculate the total weight of edges leading to other nodes with label 0, 1, 2, ..., `k` - 1 (where `k` is the number of possible labels). The new label of node `i` will then be the label whose edges (among the ones incident on node `i`) have the highest total weight.

For directed graphs, it is important to know that labels can circulate freely only within the strongly connected components of the graph and may propagate in only one direction (or not at all) *between* strongly connected components. You should treat directed edges as directed only if you are aware of the consequences.

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). https://doi.org/10.1103/PhysRevE.76.036106

Šubelj, L.: Label propagation for clustering. Chapter in "Advances in Network Clustering and Blockmodeling" edited by P. Doreian, V. Batagelj & A. Ferligoj (Wiley, New York, 2018). https://doi.org/10.1002/9781119483298.ch5 https://arxiv.org/abs/1709.05634

Traag, V. A., and Šubelj, L.: Large network community detection by fast label propagation. Scientific Reports, 13:1, (2023). https://doi.org/10.1038/s41598-023-29610-z https://arxiv.org/abs/2209.13338

Arguments:

`graph`: The input graph. Note that the algorithm was originally defined for undirected graphs. You are advised to set *`mode`* to `IGRAPH_ALL` if you pass a directed graph here to treat it as undirected.

membership: The membership vector, the result is returned here. For each vertex it gives the ID of its community (label).

mode: Whether to consider edge directions for the label propagation, and if so, which direction the labels should propagate. Ignored for undirected graphs. `IGRAPH_ALL` means to ignore edge directions (even in directed graphs). `IGRAPH_OUT` means to propagate labels along the natural direction of the edges. `IGRAPH_IN` means to propagate labels *backwards* (i.e. from head to tail). It is advised to set this to `IGRAPH_ALL` unless you are specifically interested in the effect of edge directions.

weights: The weight vector, it should contain a positive weight for all the edges.

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. Unlabeled vertices which are not reachable from any labeled ones will remain unlabeled at the end of the label propagation process, and will be labeled in an additional step to avoid returning negative values in *membership*. In undirected graphs, this happens when entire connected components are unlabeled. Then, each unlabeled component will receive its own separate label. In directed graphs, the outcome of the additional labeling should be considered undefined and may change in the future; please do not rely on it.

fixed: Boolean vector denoting which labels are fixed. Of course this makes sense only if you provided an initial state, otherwise this element will be ignored. Note that vertices without labels cannot be fixed. The fixed status will be ignored for these with a warning. Also note that label numbers by themselves have no meaning, and igraph may renumber labels. However, co-membership constraints will be respected: two vertices can be fixed to be in the same or in different communities.

lpa_variant: Which variant of the label propagation algorithm to run.

`IGRAPH_LPA_DOMINANCE`	check for dominance of all nodes after each iteration.
`IGRAPH_LPA_RETENTION`	keep current label if among dominant labels, only check if labels changed.
`IGRAPH_LPA_FAST`	sample from dominant labels, only check neighbors.

Returns:

Error code.

Time complexity: O(m+n)

Example 25.8. File `examples/simple/igraph_community_label_propagation.c`

The InfoMAP algorithm

`igraph_community_infomap` — Community structure that minimizes the expected description length of a random walker trajectory.

```
igraph_error_t igraph_community_infomap(
        const igraph_t *graph,
        const igraph_vector_t *edge_weights,
        const igraph_vector_t *vertex_weights,
        igraph_int_t nb_trials,
        igraph_bool_t is_regularized,
        igraph_real_t regularization_strength,
        igraph_vector_int_t *membership,
        igraph_real_t *codelength);
```

Implementation of the Infomap community detection algorithm of Martin Rosvall and Carl T. Bergstrom. This algorithm takes edge directions into account. For more details, see the visualization of the math and the map generator at https://www.mapequation.org.

Infomap is based on a random walker model similar to PageRank: the walker either chooses out-edges to follow with probabilities proportional to edge weights, or teleports to a random vertex with probability 0.15. Vertex weights can be given to control the probability of choosing different vertices as the target of the teleportation. In addition, Infomap can be regularized to account for potential missing links.

As of igraph 1.0, the Infomap library written by Daniel Edler, Anton Holmgren and Martin Rosvall is used. See https://github.com/mapequation/infomap/.

If you want to specify a random seed (as in the original implementation) you can use `igraph_rng_seed()`.

References:

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

M. Rosvall, D. Axelsson, and C. T. Bergstrom: The map equation, Eur. Phys. J. Special Topics 178, 13 (2009). https://dx.doi.org/10.1140/epjst/e2010-01179-1, https://arxiv.org/abs/0906.1405

Smiljani#, Jelena, Daniel Edler, and Martin Rosvall: Mapping Flows on Sparse Networks with Missing Links. Phys Rev E 102 (1–1): 012302 (2020). https://doi.org/10.1103/PhysRevE.102.012302, https://arxiv.org/abs/2106.14798

Arguments:

graph: The input graph. Edge directions are taken into account.

edge_weights: Numeric vector giving the weights of the edges. The random walker will favour edges with high weights over edges with low weights; the probability of picking a particular outbound edge from a node is directly proportional to its weight. If it is `NULL` then all edges will have equal weights. The weights are expected to be non-negative.

vertex_weights: Numeric vector giving the weights of the vertices. Vertices with higher weights are favoured by the random walker when it teleports to a new vertex. The probability of picking a vertex when the random walker teleports is directly proportional to the weight of the vertex. If this argument is `NULL` then all vertices will have equal weights. Weights are expected to be positive.

nb_trials: The number of attempts to partition the network (can be any integer value equal to or larger than 1).

is_regularized: If true, adds a fully connected Bayesian prior network to avoid overfitting due to missing links.

regularization_strength: Adjust relative strength of the Bayesian prior network used for regularization. This multiplies the default strength, a parameter of 1 hence uses the default regularization strength. Ignored when *is_regularized* is set to `false`.

membership: Pointer to a vector. The membership vector is stored here. `NULL` means that the caller is not interested in the membership vector.

codelength: Pointer to a real. If not `NULL` the code length of the partition is stored here.

Returns:

Error code.

See also:

`igraph_community_spinglass()`, `igraph_community_edge_betweenness()`, `igraph_community_walktrap()`.

Time complexity: TODO.

Voronoi communities

`igraph_community_voronoi` — Finds communities using Voronoi partitioning.

```
igraph_error_t igraph_community_voronoi(
        const igraph_t *graph,
        igraph_vector_int_t *membership, igraph_vector_int_t *generators,
        igraph_real_t *modularity,
        const igraph_vector_t *lengths, const igraph_vector_t *weights,
        igraph_neimode_t mode, igraph_real_t r);
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

This function finds communities using a Voronoi partitioning of vertices based on the given edge lengths divided by the edge clustering coefficient (`igraph_ecc()`). The generator vertices are chosen to be those with the largest local relative density within a radius *r*, with the local relative density of a vertex defined as `s m / (m + k)`, where `s` is the strength of the vertex, `m` is the number of edges within the vertex's first order neighborhood, while `k` is the number of edges with only one endpoint within this neighborhood.

References:

Deritei et al., Community detection by graph Voronoi diagrams, New Journal of Physics 16, 063007 (2014) https://doi.org/10.1088/1367-2630/16/6/063007

Molnár et al., Community Detection in Directed Weighted Networks using Voronoi Partitioning, Scientific Reports 14, 8124 (2024) https://doi.org/10.1038/s41598-024-58624-4

Arguments:

graph: The input graph. It must be simple.

membership: If not NULL, the membership of each vertex is returned here.

generators: If not NULL, the generator points used for Voronoi partitioning are returned here.

modularity: If not NULL, the modularity score of the partitioning is returned here.

lengths: Edge lengths, or NULL to consider all edges as having unit length. Voronoi partitioning will use edge lengths equal to lengths / ECC where ECC is the edge clustering coefficient.

weights: Edge weights, or NULL to consider all edges as having unit weight. Weights are used when selecting generator points, as well as for computing modularity.

mode: If IGRAPH_OUT, distances from generator points to all other nodes are considered. If IGRAPH_IN, the reverse distances are used. If IGRAPH_ALL, edge directions are ignored. This parameter is ignored for undirected graphs.

r: The radius/resolution to use when selecting generator points. The larger this value, the fewer partitions there will be. Pass in a negative value to automatically select the radius that maximizes modularity.

Returns:

Error code.

See also:

igraph_voronoi(), igraph_ecc().

Time complexity: TODO.

Chapter 26. Graphlets

Introduction

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 in the thresholded input graph. In the second step, the graph is projected onto the candidate basis, resulting in a weight coefficient for each clique in the candidate basis.

For more information on graphlet decomposition, see Hossein Azari Soufiani and Edoardo M Airoldi: "Graphlet decomposition of a weighted network", https://arxiv.org/abs/1203.2821 and http://proceedings.mlr.press/v22/azari12/azari12.pdf

igraph contains three functions for performing the graphlet decomponsition of a graph. The first is `igraph_graphlets()`, which performs both steps of the method and returns a list of subgraphs with their corresponding weights. The other two functions correspond to the first and second steps of the algorithm, and they are useful if the user wishes to perform them individually: `igraph_graphlets_candidate_basis()` and `igraph_graphlets_project()`.

Note: The term "graphlet" is used for several unrelated concepts in the literature. If you are looking to count induced subgraphs, see `igraph_motifs_randesu()` and `igraph_subisomorphic_lad()`.

Performing graphlet decomposition

`igraph_graphlets` — Calculate graphlets basis and project the graph on it.

```
igraph_error_t igraph_graphlets(const igraph_t *graph,
                    const igraph_vector_t *weights,
                    igraph_vector_int_list_t *cliques,
                    igraph_vector_t *Mu, igraph_int_t niter);
```

This function simply calls `igraph_graphlets_candidate_basis()` and `igraph_graphlets_project()`, and then orders the graphlets according to decreasing weights.

Arguments:

`graph`: The input graph, it must be a simple graph, edge directions are ignored.

`weights`: Weights of the edges, a vector.

`cliques`: An initialized list of integer vectors. The graphlet basis is stored here. Each element of the list is an integer vector of vertex IDs, encoding a single basis subgraph.

`Mu`: An initialized vector, the weights of the graphlets will be stored here.

`niter`: The number of iterations to perform for the projection step.

Returns:

Error code.

See also: `igraph_graphlets_candidate_basis()` and `igraph_graphlets_project()`.

`igraph_graphlets_candidate_basis` — Calculate a candidate graphlets basis

```
igraph_error_t igraph_graphlets_candidate_basis(const igraph_t *graph,
                                        const igraph_vector_t *weights,
                                        igraph_vector_int_list_t *cliques,
                                        igraph_vector_t *thresholds);
```

Arguments:

`graph`: The input graph, it must be a simple graph, edge directions are ignored.

`weights`: Weights of the edges, a vector.

`cliques`: An initialized list of integer vectors. The graphlet basis is stored here. Each element of the list is an integer vector of vertex IDs, encoding a single basis subgraph.

`thresholds`: An initialized vector, the (highest possible) weight thresholds for finding the basis subgraphs are stored here.

Returns:

Error code.

See also: `igraph_graphlets()` and `igraph_graphlets_project()`.

`igraph_graphlets_project` — Project a graph on a graphlets basis.

```
igraph_error_t igraph_graphlets_project(const igraph_t *graph,
                                const igraph_vector_t *weights,
                                const igraph_vector_int_list_t *cliques,
                                igraph_vector_t *Mu, igraph_bool_t startMu,
                                igraph_int_t niter);
```

Note that the graph projected does not have to be the same that was used to calculate the graphlet basis, but it is assumed that it has the same number of vertices, and the vertex IDs of the two graphs match.

Arguments:

`graph`: The input graph, it must be a simple graph, edge directions are ignored.

`weights`: Weights of the edges in the input graph, a vector.

`cliques`: An initialized list of integer vectors. The graphlet basis is stored here. Each element of the list is an integer vector of vertex IDs, encoding a single basis subgraph.

`Mu`: An initialized vector, the weights of the graphlets will be stored here. This vector is also used to initialize the the weight vector for the iterative algorithm, if the `startMu` argument is true.

`startMu`: If true, then the supplied Mu vector is used as the starting point of the iteration. Otherwise a constant 1 vector is used.

niter: The number of iterations to perform.

Returns:

Error code.

See also: `igraph_graphlets()` and `igraph_graphlets_candidate_basis()`.

Chapter 27. Hierarchical random graphs

Introduction

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 read the following two articles for more about hierarchical random graphs: A. Clauset, C. Moore, and M.E.J. Newman. Hierarchical structure and the prediction of missing links in networks. Nature 453, 98 - 101 (2008); and 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).

igraph contains functions for fitting HRG models to a given network (`igraph_hrg_fit`), for generating networks from a given HRG ensemble (`igraph_hrg_game`, `igraph_hrg_sample`), converting an igraph graph to a HRG and back (`igraph_hrg_create`, `igraph_hrg_dendrogram`), for calculating a consensus tree from a set of sampled HRGs (`igraph_hrg_consensus`) and for predicting missing edges in a network based on its HRG models (`igraph_hrg_predict`).

The igraph HRG implementation is heavily based on the code published by Aaron Clauset, at his website, https://aaronclauset.github.io/hierarchy/

Representing HRGs

`igraph_hrg_t` — Data structure to store a hierarchical random graph.

```
typedef struct igraph_hrg_t {
    igraph_vector_int_t left;
    igraph_vector_int_t right;
    igraph_vector_t prob;
    igraph_vector_int_t vertices;
    igraph_vector_int_t edges;
} igraph_hrg_t;
```

A hierarchical random graph (HRG) can be given as a binary tree, where the internal vertices are labeled with real numbers.

Note that you don't necessarily have to know this internal representation for using the HRG functions, just pass the HRG objects created by one igraph function, to another igraph function.

It has the following members:

Values:

`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.

`igraph_hrg_init` — Allocate memory for a HRG.

```
igraph_error_t igraph_hrg_init(igraph_hrg_t *hrg, igraph_int_t n);
```

This function must be called before passing an `igraph_hrg_t` to an igraph function.

Arguments:

`hrg`: Pointer to the HRG data structure to initialize.

`n`: The number of vertices in the graph that is modeled by this HRG. It can be zero, if this is not yet known.

Returns:

Error code.

Time complexity: O(n), the number of vertices in the graph.

`igraph_hrg_destroy` — Deallocate memory for an HRG.

```
void igraph_hrg_destroy(igraph_hrg_t *hrg);
```

The HRG data structure can be reinitialized again with an `igraph_hrg_destroy` call.

Arguments:

`hrg`: Pointer to the HRG data structure to deallocate.

Time complexity: operating system dependent.

`igraph_hrg_size` — Returns the size of the HRG, the number of leaf nodes.

```
igraph_int_t igraph_hrg_size(const igraph_hrg_t *hrg);
```

Arguments:

`hrg`: Pointer to the HRG.

Returns:

The number of leaf nodes in the HRG.

Time complexity: O(1).

`igraph_hrg_resize` — Resize a HRG.

```
igraph_error_t igraph_hrg_resize(igraph_hrg_t *hrg, igraph_int_t newsize);
```

Arguments:

`hrg`: Pointer to an initialized (see `igraph_hrg_init`) HRG.

`newsize`: The new size, i.e. the number of leaf nodes.

Returns:

Error code.

Time complexity: O(n), n is the new size.

Fitting HRGs

`igraph_hrg_fit` — Fit a hierarchical random graph model to a network.

```
igraph_error_t igraph_hrg_fit(const igraph_t *graph,
                  igraph_hrg_t *hrg,
                  igraph_bool_t start,
                  igraph_int_t steps);
```

Arguments:

`graph`: The igraph graph to fit the model to. Edge directions are ignored in directed graphs.

`hrg`: Pointer to an initialized HRG, the result of the fitting is stored here. It can also be used to pass a HRG to the function, that can be used as the starting point of the Markov Chain Monte Carlo fitting, if the *`start`* argument is true.

`start`: Whether to start the fitting from the given HRG model.

`steps`: Integer, the number of MCMC steps to take in the fitting procedure. If this is zero, then the fitting stops if a convergence criteria is fulfilled.

Returns:

Error code.

Time complexity: TODO.

`igraph_hrg_consensus` — Calculate a consensus tree for a HRG.

```
igraph_error_t igraph_hrg_consensus(const igraph_t *graph,
                            igraph_vector_int_t *parents,
                            igraph_vector_t *weights,
                            igraph_hrg_t *hrg,
                            igraph_bool_t start,
                            igraph_int_t num_samples);
```

The calculation can be started from the given HRG (*hrg*), or (if *start* is false), a HRG is first fitted to the given graph.

Arguments:

graph: The input graph.

parents: An initialized vector, the results are stored here. For each vertex, the id of its parent vertex is stored, or -1, 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 occured in the generated network samples, for each internal vertices. The order is the same as in *parents*.

hrg: A hierarchical random graph. It is used as a starting point for the sampling, if the *start* argument is true. It is modified along the MCMC.

start: Whether to use the supplied HRG (in *hrg*) as a starting point for the MCMC.

num_samples: The number of samples to generate for creating the consensus tree.

Returns:

Error code.

Time complexity: TODO.

HRG sampling

`igraph_hrg_sample` — Sample from a hierarchical random graph model.

```
igraph_error_t igraph_hrg_sample(const igraph_hrg_t *hrg, igraph_t *sample);
```

This function draws a single sample from a hierarchical random graph model.

Arguments:

hrg: A HRG model to sample from

sample: Pointer to an uninitialized graph; the sample is stored here.

Returns:

Error code.

Time complexity: TODO.

igraph_hrg_game — Generate a hierarchical random graph.

```
igraph_error_t igraph_hrg_game(igraph_t *graph,
                    const igraph_hrg_t *hrg);
```

This function is a simple shortcut to `igraph_hrg_sample`. It creates a single graph from the given HRG.

Arguments:

graph: Pointer to an uninitialized graph, the new graph is created here.

hrg: The hierarchical random graph model to sample from.

Returns:

Error code.

Time complexity: TODO.

Conversion to and from igraph graphs

igraph_from_hrg_dendrogram — Create a graph representation of the dendrogram of a hierarchical random graph model.

```
igraph_error_t igraph_from_hrg_dendrogram(
    igraph_t *graph, const igraph_hrg_t *hrg, igraph_vector_t *prob
);
```

Creates the igraph graph equivalent of the dendrogram encoded in an `igraph_hrg_t` data structure. The probabilities associated to the nodes are returned in a vector so this function works without an attribute handler.

Arguments:

graph: Pointer to an uninitialized graph, the result is stored here.

hrg: The hierarchical random graph to convert.

prob: Pointer to an *initialized* vector; the probabilities associated to the nodes of the dendrogram will be stored here. Leaf nodes will have an associated probability of `IGRAPH_NAN`. You may set this to `NULL` if you do not need the probabilities.

Returns:

Error code.

Time complexity: O(n), the number of vertices in the graph.

igraph_hrg_create — Create a HRG from an igraph graph.

```
igraph_error_t igraph_hrg_create(igraph_hrg_t *hrg,
                     const igraph_t *graph,
                     const igraph_vector_t *prob);
```

Arguments:

hrg: Pointer to an initialized `igraph_hrg_t`. The result is stored here.

graph: The igraph graph to convert. It must be a directed binary tree, with n-1 internal and n leaf vertices. The root vertex must have in-degree zero.

prob: The vector of probabilities, this is used to label the internal nodes of the hierarchical random graph.

Returns:

Error code.

Time complexity: O(n), the number of vertices in the tree.

Predicting missing edges

`igraph_hrg_predict` — Predict missing edges in a graph, based on HRG models.

```
igraph_error_t igraph_hrg_predict(const igraph_t *graph,
                      igraph_vector_int_t *edges,
                      igraph_vector_t *prob,
                      igraph_hrg_t *hrg,
                      igraph_bool_t start,
                      igraph_int_t num_samples,
                      igraph_int_t num_bins);
```

Samples HRG models for a network, and estimated the probability that an edge was falsely observed as non-existent in the network.

Arguments:

graph: The input graph.

edges: The list of missing edges is stored here, the first two elements are the first edge, the next two the second edge, etc.

prob: Vector of probabilies for the existence of missing edges, in the order corresponding to `edges`.

hrg: A HRG, it is used as a starting point if `start` is true. It is also modified during the MCMC sampling.

start: Whether to start the MCMC from the given HRG.

num_samples: The number of samples to generate.

num_bins: Controls the resolution of the edge probabilities. Higher numbers result higher resolution.

Returns:

Error code.

Time complexity: TODO.

Deprecated functions

`igraph_hrg_dendrogram` — Create a dendrogram from a hierarchical random graph.

```
igraph_error_t igraph_hrg_dendrogram(igraph_t *graph, const igraph_hrg_t *hrg);
```

Creates the igraph graph equivalent of an `igraph_hrg_t` data structure.

Arguments:

`graph`: Pointer to an uninitialized graph, the result is stored here.

`hrg`: The hierarchical random graph to convert.

Returns:

Error code.

Time complexity: O(n), the number of vertices in the graph.

Warning

Deprecated since version 0.10.5. Please do not use this function in new code; use `igraph_from_hrg_dendrogram()` instead.

Chapter 28. Embedding of graphs

Spectral embedding

`igraph_adjacency_spectral_embedding` — Adjacency spectral embedding

```
igraph_error_t igraph_adjacency_spectral_embedding(const igraph_t *graph,
                                                   igraph_int_t n,
                                                   const igraph_vector_t *weights,
                                                   igraph_eigen_which_position_t which,
                                                   igraph_bool_t scaled,
                                                   igraph_matrix_t *X,
                                                   igraph_matrix_t *Y,
                                                   igraph_vector_t *D,
                                                   const igraph_vector_t *cvec,
                                                   igraph_arpack_options_t *options);
```

Spectral decomposition of the adjacency matrices of graphs. This function computes an n-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=U D V^T. In the case, where the graph is a random dot product graph generated using latent position vectors in R^n for each vertex, the embedding will provide an estimate of these latent vectors.

For undirected graphs, the latent positions are calculated as X = U^n D^(1/2) where U^n equals to the first no columns of U, and D^(1/2) is a diagonal matrix containing the square root of the selected singular values on the diagonal.

For directed graphs, the embedding is defined as the pair X = U^n D^(1/2), Y = V^n D^(1/2). (For undirected graphs U=V, so it is sufficient to keep one of them.)

Arguments:

graph: The input graph, can be directed or undirected.

n: An integer scalar. This value is the embedding dimension of the spectral embedding. Should be smaller than the number of vertices. The largest n-dimensional non-zero singular values are used for the spectral embedding.

weights: Optional edge weights. Supply a null pointer for unweighted graphs.

which: Which eigenvalues (or singular values, for directed graphs) to use, possible values:

`IGRAPH_EIGEN_LM` the ones with the largest magnitude

`IGRAPH_EIGEN_LA` the (algebraic) largest ones

`IGRAPH_EIGEN_SA` the (algebraic) smallest ones.
For directed graphs, `IGRAPH_EIGEN_LM` and `IGRAPH_EIGEN_LA` are the same because singular values are used for the ordering instead of eigenvalues.

scaled: Whether to return X and Y (if `scaled` is true), or U and V.

X: Initialized matrix, the estimated latent positions are stored here.

Y: Initialized matrix or a null pointer. If not a null pointer, then the second half of the latent positions are stored here. (For undirected graphs, this always equals X.)

D: Initialized vector or a null pointer. If not a null pointer, then the eigenvalues (for undirected graphs) or the singular values (for directed graphs) are stored here.

cvec: A numeric vector, its length is the number vertices in the graph. This vector is added to the diagonal of the adjacency matrix, before performing the SVD.

options: Options to ARPACK. See `igraph_arpack_options_t` for details. Supply `NULL` to use the defaults. Note that the function overwrites the `n` (number of vertices), `nev` and `which` parameters and it always starts the calculation from a random start vector.

Returns:

Error code.

`igraph_laplacian_spectral_embedding` — Spectral embedding of the Laplacian of a graph

```
igraph_error_t igraph_laplacian_spectral_embedding(const igraph_t *graph,
                                                   igraph_int_t n,
                                                   const igraph_vector_t *weights,
                                                   igraph_eigen_which_position_t which,
                                                   igraph_laplacian_spectral_embedding_typ
                                                   igraph_bool_t scaled,
                                                   igraph_matrix_t *X,
                                                   igraph_matrix_t *Y,
                                                   igraph_vector_t *D,
                                                   igraph_arpack_options_t *options);
```

This function essentially does the same as `igraph_adjacency_spectral_embedding`, but works on the Laplacian of the graph, instead of the adjacency matrix.

Arguments:

graph: The input graph.

n: The number of eigenvectors (or singular vectors if the graph is directed) to use for the embedding.

weights: Optional edge weights. Supply a null pointer for unweighted graphs.

which: Which eigenvalues (or singular values, for directed graphs) to use, possible values:

`IGRAPH_EIGEN_LM` the ones with the largest magnitude

`IGRAPH_EIGEN_LA` the (algebraic) largest ones

`IGRAPH_EIGEN_SA` the (algebraic) smallest ones.
For directed graphs, `IGRAPH_EIGEN_LM` and `IGRAPH_EIGEN_LA` are the same because singular values are used for the ordering instead of eigenvalues.

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:

`IGRAPH_EMBEDDING_D_A` means D - A where D is the degree matrix and A is the adjacency matrix

`IGRAPH_EMBEDDING_DAD` means Di times A times Di, where Di is the inverse of the square root of the degree matrix;

`IGRAPH_EMBEDDING_I_DAD` means I - Di A Di, where I is the identity matrix.

`scaled`: Whether to return X and Y (if `scaled` is true), or U and V.

`X`: Initialized matrix, the estimated latent positions are stored here.

`Y`: Initialized matrix or a null pointer. If not a null pointer, then the second half of the latent positions are stored here. (For undirected graphs, this always equals X.)

`D`: Initialized vector or a null pointer. If not a null pointer, then the eigenvalues (for undirected graphs) or the singular values (for directed graphs) are stored here.

`options`: Options to ARPACK. See `igraph_arpack_options_t` for details. Supply `NULL` to use the defaults. Note that the function overwrites the `n` (number of vertices), `nev` and `which` parameters and it always starts the calculation from a random start vector.

Returns:

Error code.

See also:

`igraph_adjacency_spectral_embedding` to embed the adjacency matrix.

`igraph_dim_select` — Dimensionality selection.

```
igraph_error_t igraph_dim_select(const igraph_vector_t *sv, igraph_int_t *dim);
```

Dimensionality selection for singular values using profile likelihood.

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.

Arguments:

`sv`: A numeric vector, the ordered singular values.

`dim`: The result is stored here.

Returns:

Error code.

Time complexity: O(n), n is the number of values in sv.

See also:

`igraph_adjacency_spectral_embedding()`.

Chapter 29. Generating layouts for graph drawing

2D layout generators

Layout generator functions (or at least most of them) try to place the vertices and edges of a graph on a 2D plane or in 3D space in a way which visually pleases the human eye.

They take a graph object and a number of parameters as arguments and return an igraph_matrix_t, in which each row gives the coordinates of a vertex.

`igraph_layout_random` — Places the vertices uniformly randomly within a square.

```
igraph_error_t igraph_layout_random(const igraph_t *graph, igraph_matrix_t *res
```

Vertex coordinates range from -1 to 1, and are placed in two columns of a matrix, with a row for each vertex.

Arguments:

graph: Pointer to an initialized graph object.

res: Pointer to an initialized matrix object. This will contain the result and will be resized as needed.

Returns:

Error code.

Time complexity: O(|V|), the number of vertices.

`igraph_layout_circle` — Places the vertices uniformly on a circle in arbitrary order.

```
igraph_error_t igraph_layout_circle(const igraph_t *graph, igraph_matrix_t *res
                        igraph_vs_t order);
```

Arguments:

graph: Pointer to an initialized graph object.

res: Pointer to an initialized matrix object. This will contain the result and will be resized as needed.

order: The order of the vertices on the circle. The vertices not included here, will be placed at (0,0). Supply `igraph_vss_all()` here to place vertices in the order of their vertex IDs.

Returns:

Error code.

Time complexity: O(|V|), the number of vertices.

igraph_layout_star — Generates a star-like layout.

```
igraph_error_t igraph_layout_star(const igraph_t *graph, igraph_matrix_t *res,
                      igraph_int_t center, const igraph_vector_int_t *order);
```

Arguments:

graph: The input graph. Its edges are ignored by this function.

res: Pointer to an initialized matrix object. This will contain the result and will be resized as needed.

center: The id of the vertex to put in the center. You can set it to any arbitrary value for the special case when the input graph has no vertices; otherwise it must be between 0 and the number of vertices minus 1.

order: A numeric vector giving the order of the vertices (including the center vertex!). If a null pointer, then the vertices are placed in increasing vertex ID order.

Returns:

Error code.

Time complexity: O(|V|), linear in the number of vertices.

See also:

`igraph_layout_circle()` and other layout generators.

igraph_layout_grid — Places the vertices on a regular grid on the plane.

```
igraph_error_t igraph_layout_grid(const igraph_t *graph, igraph_matrix_t *res,
```

Arguments:

graph: Pointer to an initialized graph object.

res: Pointer to an initialized matrix object. This will contain the result and will be resized as needed.

width: The number of vertices in a single row of the grid. When zero or negative, the width of the grid will be the square root of the number of vertices, rounded up if needed.

Returns:

Error code. The current implementation always returns with success.

Time complexity: O(|V|), the number of vertices.

`igraph_layout_graphopt` — Optimizes vertex layout via the graphopt algorithm.

```
igraph_error_t igraph_layout_graphopt(const igraph_t *graph, igraph_matrix_t *r
                            igraph_int_t niter,
                            igraph_real_t node_charge, igraph_real_t node_mass,
                            igraph_real_t spring_length,
                            igraph_real_t spring_constant,
                            igraph_real_t max_sa_movement,
                            igraph_bool_t use_seed);
```

This is a port of the graphopt layout algorithm by Michael Schmuhl. graphopt version 0.4.1 was rewritten in C, the support for layers was removed and the code was reorganized to avoid some unnecessary steps when 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.)

See also https://web.archive.org/web/20220611030748/http://www.schmuhl.org/graphopt/ and https://sourceforge.net/projects/graphopt/ for the original graphopt.

Arguments:

`graph`: The input graph.

`res`: Pointer to an initialized matrix, the result will be stored here and its initial contents are used as the starting point of the simulation if the *`use_seed`* argument is true. Note that in this case the matrix should have the proper size, otherwise a warning is issued and the supplied values are ignored. If no starting positions are given (or they are invalid) then a random starting position is used. The matrix will be resized if needed.

`niter`: Integer constant, 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 *`res`* argument. The original graphopt default is 500.

`node_charge`: The charge of the vertices, used to calculate electric repulsion. The original graphopt default is 0.001.

`node_mass`: The mass of the vertices, used for the spring forces. The original graphopt defaults to 30.

`spring_length`: The length of the springs. The original graphopt defaults to zero.

`spring_constant`: The spring constant, the original graphopt defaults to one.

`max_sa_movement`: Real constant, it gives the maximum amount of movement allowed in a single step along a single axis. The original graphopt default is 5.

`use_seed`: Boolean, whether to use the positions in *`res`* as a starting configuration. See also *`res`* above.

Returns:

Error code.

Time complexity: O(n (|V|^2+|E|)), n is the number of iterations, |V| is the number of vertices, |E| the number of edges. If *node_charge* is zero then it is only O(n|E|).

igraph_layout_bipartite — Simple layout for bipartite graphs.

```
igraph_error_t igraph_layout_bipartite(const igraph_t *graph,
                            const igraph_vector_bool_t *types,
                            igraph_matrix_t *res, igraph_real_t hgap,
                            igraph_real_t vgap, igraph_int_t maxiter);
```

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, by calling `igraph_layout_sugiyama()`.

Arguments:

graph: The input graph.

types: A boolean vector containing ones and zeros, the vertex types. Its length must match the number of vertices in the graph.

res: Pointer to an initialized matrix, the result, the x and y coordinates are stored here.

hgap: The preferred minimum horizontal gap between vertices in the same layer (i.e. vertices of the same type).

vgap: The distance between layers.

maxiter: 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.

Returns:

Error code.

See also:

`igraph_layout_sugiyama()`.

The DrL layout generator

DrL is a sophisticated layout generator developed and implemented by Shawn Martin et al. As of October 2012 the original DrL homepage is unfortunately not available. You can read more about this algorithm in 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.

Only a subset of the complete DrL functionality is included in igraph, parallel runs and recursive, multi-level layouting is not supported.

The parameters of the layout are stored in an `igraph_layout_drl_options_t` structure, this can be initialized by calling the function `igraph_layout_drl_options_init()`. The fields of this structure can then be adjusted by hand if needed. The layout is calculated by an `igraph_layout_drl()` call.

`igraph_layout_drl_options_t` — Parameters for the DrL layout generator

```
typedef struct igraph_layout_drl_options_t {
    igraph_real_t    edge_cut;
    igraph_int_t init_iterations;
    igraph_real_t    init_temperature;
    igraph_real_t    init_attraction;
    igraph_real_t    init_damping_mult;
    igraph_int_t liquid_iterations;
    igraph_real_t    liquid_temperature;
    igraph_real_t    liquid_attraction;
    igraph_real_t    liquid_damping_mult;
    igraph_int_t expansion_iterations;
    igraph_real_t    expansion_temperature;
    igraph_real_t    expansion_attraction;
    igraph_real_t    expansion_damping_mult;
    igraph_int_t cooldown_iterations;
    igraph_real_t    cooldown_temperature;
    igraph_real_t    cooldown_attraction;
    igraph_real_t    cooldown_damping_mult;
    igraph_int_t crunch_iterations;
    igraph_real_t    crunch_temperature;
    igraph_real_t    crunch_attraction;
    igraph_real_t    crunch_damping_mult;
    igraph_int_t simmer_iterations;
    igraph_real_t    simmer_temperature;
    igraph_real_t    simmer_attraction;
    igraph_real_t    simmer_damping_mult;
} igraph_layout_drl_options_t;
```

Values:

`edge_cut`: The edge cutting parameter. 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. The default value is 32/40.

`init_iterations`: Number of iterations, initial phase.

`init_temperature`: Start temperature, initial phase.

`init_attraction`: Attraction, initial phase.

`init_damping_mult`: Damping factor, initial phase.

`liquid_iterations`: Number of iterations in the liquid phase.

`liquid_temperature`: Start temperature in the liquid phase.

`liquid_attraction`: Attraction in the liquid phase.

`liquid_damping_mult`: Multiplicatie damping factor, liquid phase.

`expansion_iterations`: Number of iterations in the expansion phase.

`expansion_temperature`: Start temperature in the expansion phase.

`expansion_attraction`: Attraction, expansion phase.

`expansion_damping_mult`: Damping factor, expansion phase.

`cooldown_iterations`: Number of iterations in the cooldown phase.

`cooldown_temperature`: Start temperature in the cooldown phase.

`cooldown_attraction`: Attraction in the cooldown phase.

`cooldown_damping_mult`: Damping fact int the cooldown phase.

`crunch_iterations`: Number of iterations in the crunch phase.

`crunch_temperature`: Start temperature in the crunch phase.

`crunch_attraction`: Attraction in the crunch phase.

`crunch_damping_mult`: Damping factor in the crunch phase.

`simmer_iterations`: Number of iterations in the simmer phase.

`simmer_temperature`: Start temperature in te simmer phase.

`simmer_attraction`: Attraction in the simmer phase.

`simmer_damping_mult`: Multiplicative damping factor in the simmer phase.

`igraph_layout_drl_default_t` — Predefined parameter templates for the DrL layout generator

```
typedef enum { IGRAPH_LAYOUT_DRL_DEFAULT = 0,
               IGRAPH_LAYOUT_DRL_COARSEN,
               IGRAPH_LAYOUT_DRL_COARSEST,
               IGRAPH_LAYOUT_DRL_REFINE,
               IGRAPH_LAYOUT_DRL_FINAL
             } igraph_layout_drl_default_t;
```

These constants can be used to initialize a set of DrL parameters. These can then be modified according to the user's needs.

Values:

`IGRAPH_LAYOUT_DRL_DEFAULT`: The deafult parameters.

`IGRAPH_LAYOUT_DRL_COARSEN`: Slightly modified parameters to get a coarser layout.

`IGRAPH_LAYOUT_DRL_COARSEST`: An even coarser layout.

`IGRAPH_LAYOUT_DRL_REFINE`: Refine an already calculated layout.

`IGRAPH_LAYOUT_DRL_FINAL`: Finalize an already refined layout.

`igraph_layout_drl_options_init` — Initialize parameters for the DrL layout generator

```
igraph_error_t igraph_layout_drl_options_init(igraph_layout_drl_options_t *opti
                                    igraph_layout_drl_default_t templ);
```

This function can be used to initialize the struct holding the parameters for the DrL layout generator. There are a number of predefined templates available, it is a good idea to start from one of these by modifying some parameters.

Arguments:

`options`: The struct to initialize.

`templ`: The template to use. Currently the following templates are supplied: `IGRAPH_LAYOUT_DRL_DEFAULT`, `IGRAPH_LAYOUT_DRL_COARSEN`, `IGRAPH_LAYOUT_DRL_COARSEST`, `IGRAPH_LAYOUT_DRL_REFINE` and `IGRAPH_LAYOUT_DRL_FINAL`.

Returns:

Error code.

Time complexity: O(1).

`igraph_layout_drl` — The DrL layout generator

```
igraph_error_t igraph_layout_drl(const igraph_t *graph, igraph_matrix_t *res,
                     igraph_bool_t use_seed,
                     const igraph_layout_drl_options_t *options,
                     const igraph_vector_t *weights);
```

This function implements the force-directed DrL layout generator. Please see more in 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.

Arguments:

`graph`: The input graph.

`use_seed`: Boolean, if true, then the coordinates supplied in the *`res`* argument are used as starting points.

`res`: Pointer to a matrix, the result layout is stored here. It will be resized as needed.

`options`: The parameters to pass to the layout generator.

`weights`: Edge weights, pointer to a vector. If this is a null pointer then every edge will have the same weight.

Returns:

Error code.

Time complexity: ???.

`igraph_layout_drl_3d` — The DrL layout generator, 3d version.

```
igraph_error_t igraph_layout_drl_3d(const igraph_t *graph, igraph_matrix_t *res,
                          igraph_bool_t use_seed,
                          const igraph_layout_drl_options_t *options,
                          const igraph_vector_t *weights);
```

This function implements the force-directed DrL layout generator. Please see more in the technical report: Martin, S., Brown, W.M., Klavans, R., Boyack, K.W., DrL: Distributed Recursive (Graph) Layout. SAND Reports, 2008. 2936: p. 1-10.

This function uses a modified DrL generator that does the layout in three dimensions.

Arguments:

graph: The input graph.

use_seed: Boolean, if true, then the coordinates supplied in the *res* argument are used as starting points.

res: Pointer to a matrix, the result layout is stored here. It will be resized as needed.

options: The parameters to pass to the layout generator.

weights: Edge weights, pointer to a vector. If this is a null pointer then every edge will have the same weight.

Returns:

Error code.

Time complexity: ???.

See also:

`igraph_layout_drl()` for the standard 2d version.

`igraph_layout_fruchterman_reingold` — Places the vertices on a plane according to the Fruchterman-Reingold algorithm.

```
igraph_error_t igraph_layout_fruchterman_reingold(const igraph_t *graph,
                                        igraph_matrix_t *res,
                                        igraph_bool_t use_seed,
                                        igraph_int_t niter,
                                        igraph_real_t start_temp,
                                        igraph_layout_grid_t grid,
                                        const igraph_vector_t *weights,
                                        const igraph_vector_t *minx,
                                        const igraph_vector_t *maxx,
                                        const igraph_vector_t *miny,
                                        const igraph_vector_t *maxy);
```

This is a force-directed layout that simulates an attractive force `f_a` between connected vertex pairs and a repulsive force `f_r` between all vertex pairs. The forces are computed as a function of the distance `d` between the two vertices as

`f_a(d) = -w * d^2` and `f_r(d) = 1/d`,

where `w` represents the edge weight. The equilibrium distance of two connected vertices is thus `1/w^3`, assuming no other forces acting on them.

In disconnected graphs, igraph effectively inserts a weak connection of weight `n^(-3/2)` between all pairs of vertices, where `n` is the vertex count. This ensures that components are kept near each other.

Reference:

Fruchterman, T.M.J. and Reingold, E.M.: Graph Drawing by Force-directed Placement. Software -- Practice and Experience, 21/11, 1129--1164, 1991. https://doi.org/10.1002/spe.4380211102

Arguments:

graph: Pointer to an initialized graph object.

res: Pointer to an initialized matrix object. This will contain the result and will be resized as needed.

use_seed: If true the supplied values in the *res* argument are used as an initial layout, if false a random initial layout is used.

niter: The number of iterations to do. A reasonable default value is 500.

start_temp: Start temperature. This is the maximum amount of movement allowed along one axis, within one step, for a vertex. Currently it is decreased linearly to zero during the iteration.

grid: Whether to use the (fast but less accurate) grid based version of the algorithm. Possible values: `IGRAPH_LAYOUT_GRID`, `IGRAPH_LAYOUT_NOGRID`, `IGRAPH_LAYOUT_AUTOGRID`. The last one uses the grid based version only for large graphs, currently the ones with more than 1000 vertices.

weights: Pointer to a vector containing edge weights. Weights must be positive. If `NULL`, all edges are assumed to have weight 1. The attraction along the edges will be multiplied by the weights, resulting in vertices connected by a high-weight edge being placed closer together.

minx: Pointer to a vector, or a `NULL` pointer. If not a `NULL` pointer then the vector gives the minimum “x” coordinate for every vertex.

maxx: Same as *minx*, but the maximum “x” coordinates.

miny: Pointer to a vector, or a `NULL` pointer. If not a `NULL` pointer then the vector gives the minimum “y” coordinate for every vertex.

maxy: Same as *miny*, but the maximum “y” coordinates.

Returns:

Error code.

Time complexity: O(|V|^2) in each iteration, |V| is the number of vertices in the graph.

`igraph_layout_kamada_kawai` — Places the vertices on a plane according to the Kamada-Kawai algorithm.

```
igraph_error_t igraph_layout_kamada_kawai(const igraph_t *graph, igraph_matrix_
                                  igraph_bool_t use_seed, igraph_int_t maxiter,
                                  igraph_real_t epsilon, igraph_real_t kkconst,
                                  const igraph_vector_t *weights,
                                  const igraph_vector_t *minx, const igraph_vector
                                  const igraph_vector_t *miny, const igraph_vector
```

This is a force-directed layout. A spring is inserted between all pairs of vertices, both those which are directly connected and those that are not. The unstretched length of springs is chosen based on the undirected graph distance between the corresponding pair of vertices. Thus, in a weighted graph, increasing the weight between two vertices pushes them apart. The Young modulus of springs is inversely proportional to the graph distance, ensuring that springs between far-apart veritces will have a smaller effect on the layout.

Disconnected graphs are handled by assuming that the graph distance between vertices in different components is the same as the graph diameter.

This layout works particularly well for locally connected spatial networks such as lattices.

This layout algorithm is not suitable for large graphs. The memory requirements are of the order O(|V|^2).

Reference:

Kamada, T. and Kawai, S.: An Algorithm for Drawing General Undirected Graphs. Information Processing Letters, 31/1, 7--15, 1989. https://doi.org/10.1016/0020-0190(89)90102-6

Arguments:

graph: A graph object.

res: Pointer to an initialized matrix object. This will contain the result (x-positions in column zero and y-positions in column one) and will be resized if needed.

use_seed: Boolean, whether to use the values supplied in the *res* argument as the initial configuration. If zero and there are any limits on the X or Y coordinates, then a random initial configuration is used. Otherwise the vertices are placed on a circle of radius 1 as the initial configuration.

maxiter: The maximum number of iterations to perform. A reasonable default value is at least ten (or more) times the number of vertices.

epsilon: Stop the iteration, if the maximum delta value of the algorithm is smaller than this. It is safe to leave it at zero, and then *maxiter* iterations are performed.

kkconst: The Kamada-Kawai vertex attraction constant. Typical value: number of vertices.

weights: A vector of edge weights. Weights are interpreted as edge *lengths* in the shortest path calculation used by the Kamada-Kawai algorithm. Therefore, vertices connected by high-weight edges will be placed further apart. Pass `NULL` to assume unit weights for all edges.

minx: Pointer to a vector, or a `NULL` pointer. If not a `NULL` pointer then the vector gives the minimum “x” coordinate for every vertex.

maxx: Same as *minx*, but the maximum “x” coordinates.

miny: Pointer to a vector, or a `NULL` pointer. If not a `NULL` pointer then the vector gives the minimum “y” coordinate for every vertex.

maxy: Same as *miny*, but the maximum “y” coordinates.

Returns:

Error code.

Time complexity: O(|V|) for each iteration, after an O(|V|^2 log|V|) initialization step. |V| is the number of vertices in the graph.

`igraph_layout_gem` — Layout graph according to GEM algorithm.

```
igraph_error_t igraph_layout_gem(const igraph_t *graph, igraph_matrix_t *res,
                      igraph_bool_t use_seed, igraph_int_t maxiter,
                      igraph_real_t temp_max, igraph_real_t temp_min,
                      igraph_real_t temp_init);
```

The GEM layout algorithm, as described in Arne Frick, Andreas Ludwig, Heiko Mehldau: A Fast Adaptive Layout Algorithm for Undirected Graphs, Proc. Graph Drawing 1994, LNCS 894, pp. 388-403, 1995.

Arguments:

`graph`: The input graph. Edge directions are ignored in directed graphs.

`res`: The result is stored here. If the *`use_seed`* argument is true, then this matrix is also used as the starting point of the algorithm.

`use_seed`: Boolean, whether to use the supplied coordinates in *`res`* as the starting point. If false (zero), then a uniform random starting point is used.

`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.

Returns:

Error code.

Time complexity: O(t * n * (n+e)), where n is the number of vertices, e is the number of edges and t is the number of time steps performed.

`igraph_layout_davidson_harel` — Davidson-Harel layout algorithm.

```
igraph_error_t igraph_layout_davidson_harel(const igraph_t *graph, igraph_matri
                                  igraph_bool_t use_seed, igraph_int_t maxiter,
```

```
                                    igraph_int_t fineiter, igraph_real_t cool_fact
                                    igraph_real_t weight_node_dist, igraph_real_t w
                                    igraph_real_t weight_edge_lengths,
                                    igraph_real_t weight_edge_crossings,
                                    igraph_real_t weight_node_edge_dist);
```

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. https://doi.org/10.1145/234535.234538

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.

Arguments:

graph:	The input graph, edge directions are ignored.
res:	A matrix, the result is stored here. It can be used to supply start coordinates, see *use_seed*.
use_seed:	Boolean, whether to use the supplied *res* as start coordinates.
maxiter:	The maximum number of annealing iterations. A reasonable value for smaller graphs is 10.
fineiter:	The number of fine tuning iterations. A reasonable value is `max(10, log2(n))` where `n` is the number of vertices.
cool_fact:	Cooling factor. A reasonable value is 0.75.
weight_node_dist:	Weight for the node-node distances component of the energy function. Reasonable value: 1.0.
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, a reasonable value is the density of the graph divided by 10.
weight_edge_crossings:	Weight for the edge crossing component of the energy function, a reasonable default is 1 minus the square root of the density of the graph.
weight_node_edge_dist:	Weight for the node-edge distance component of the energy function. A reasonable value is 1 minus the density, divided by 5.

Returns:

Error code.

Time complexity: one first phase iteration has time complexity O(n^2+m^2), one fine tuning iteration has time complexity O(mn). Time complexity might be smaller if some of the weights of the components of the energy function are set to zero.

`igraph_layout_mds` — Place the vertices on a plane using multidimensional scaling.

```
igraph_error_t igraph_layout_mds(const igraph_t *graph, igraph_matrix_t *res,
                      const igraph_matrix_t *dist, igraph_int_t dim);
```

This layout requires a distance matrix, where the intersection of row i and column j specifies the desired distance between vertex i and vertex j. The algorithm will try to place the vertices in a space having a given number of dimensions in a way that approximates the distance relations prescribed in the distance matrix. igraph uses the classical multidimensional scaling by Torgerson; for more details, see Cox & Cox: Multidimensional Scaling (1994), Chapman and Hall, London.

If the input graph is disconnected, igraph will decompose it first into its subgraphs, lay out the subgraphs one by one using the appropriate submatrices of the distance matrix, and then merge the layouts using `igraph_layout_merge_dla()`. Since `igraph_layout_merge_dla()` works for 2D layouts only, you cannot run the MDS layout on disconnected graphs for more than two dimensions.

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.

Arguments:

`graph`: A graph object.

`res`: Pointer to an initialized matrix object. This will contain the result and will be resized if needed.

`dist`: The distance matrix. It must be symmetric and this function does not check whether the matrix is indeed symmetric. Results are unspecified if you pass a non-symmetric matrix here. You can set this parameter to null; in this case, the undirected shortest path lengths between vertices will be used as distances.

`dim`: The number of dimensions in the embedding space. For 2D layouts, supply 2 here.

Returns:

Error code.

Added in version 0.6.

Time complexity: usually around O(|V|^2 dim).

`igraph_layout_lgl` — Force based layout algorithm for large graphs.

```
igraph_error_t igraph_layout_lgl(const igraph_t *graph, igraph_matrix_t *res,
                     igraph_int_t maxit, igraph_real_t maxdelta,
                     igraph_real_t area, igraph_real_t coolexp,
                     igraph_real_t repulserad, igraph_real_t cellsize,
                     igraph_int_t proot);
```

This is a layout generator similar to the Large Graph Layout algorithm and program (http://lgl.sourceforge.net/). But unlike LGL, this version uses a Fruchterman-Reingold style simulated annealing al-

gorithm for placing the vertices. The speedup is achieved by placing the vertices on a grid and calculating the repulsion only for vertices which are closer to each other than a limit.

Arguments:

graph: The (initialized) graph object to place. It must be connnected; disconnected graphs are not handled by the algorithm.

res: Pointer to an initialized matrix object to hold the result. It will be resized if needed.

maxit: The maximum number of cooling iterations to perform for each layout step. A reasonable default is 150.

maxdelta: The maximum length of the move allowed for a vertex in a single iteration. A reasonable default is the number of vertices.

area: This parameter gives the area of the square on which the vertices will be placed. A reasonable default value is the number of vertices squared.

coolexp: The cooling exponent. A reasonable default value is 1.5.

repulserad: Determines the radius at which vertex-vertex repulsion cancels out attraction of adjacent vertices. A reasonable default value is *area* times the number of vertices.

cellsize: The size of the grid cells, one side of the square. A reasonable default value is the fourth root of *area* (or the square root of the number of vertices if *area* is also left at its default value).

proot: The root vertex, this is placed first, its neighbors in the first iteration, second neighbors in the second, etc. If negative then a random vertex is chosen.

Returns:

Error code.

Added in version 0.2.

Time complexity: ideally O(dia*maxit*(|V|+|E|)), |V| is the number of vertices, dia is the diameter of the graph, worst case complexity is still O(dia*maxit*(|V|^2+|E|)), this is the case when all vertices happen to be in the same grid cell.

Layouts for trees and acyclic graphs

`igraph_layout_reingold_tilford` — Reingold-Tilford layout for tree graphs.

```
igraph_error_t igraph_layout_reingold_tilford(const igraph_t *graph,
                                      igraph_matrix_t *res,
                                      igraph_neimode_t mode,
                                      const igraph_vector_int_t *roots,
                                      const igraph_vector_int_t *rootlevel);
```

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, see:

Reingold, E and Tilford, J: Tidier drawing of trees. IEEE Trans. Softw. Eng., SE-7(2):223--228, 1981. https://doi.org/10.1109/TSE.1981.234519

If the given graph is not a tree, a breadth-first search is executed first to obtain a possible spanning tree.

Arguments:

graph: The graph object.

res: The result, the coordinates in a matrix. The parameter should point to an initialized matrix object and will be resized.

mode: Specifies which edges to consider when building the tree. If it is `IGRAPH_OUT` then only the outgoing, if it is `IGRAPH_IN` then only the incoming edges of a parent are considered. If it is `IGRAPH_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.

roots: The index of the root vertex or root vertices. The set of roots should be specified so that all vertices of the graph are reachable from them. Simply put, in the undirected case, one root should be given from each connected component. If *roots* is `NULL` or a pointer to an empty vector, then the roots will be selected automatically. Currently, automatic root selection prefers low eccentricity vertices in graphs with fewer than 500 vertices, and high degree vertices (according to *mode*) in larger graphs. The root selection heuristic may change without notice. To ensure a consistent output, please specify the roots manually. The `igraph_roots_for_tree_layout()` function gives more control over automatic root selection.

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 not a null pointer and the *roots* argument is also given (and it is not a null pointer of an empty vector).

Returns:

Error code.

Added in version 0.2.

See also:

`igraph_layout_reingold_tilford_circular()`, `igraph_roots_for_tree_layout()`

Example 29.1. File `examples/simple/igraph_layout_reingold_tilford.c`

`igraph_layout_reingold_tilford_circular` — Circular Reingold-Tilford layout for trees.

```
igraph_error_t igraph_layout_reingold_tilford_circular(const igraph_t *graph,
        igraph_matrix_t *res,
        igraph_neimode_t mode,
        const igraph_vector_int_t *roots,
        const igraph_vector_int_t *rootlevel);
```

This layout is almost the same as `igraph_layout_reingold_tilford()`, but the tree is drawn in a circular way, with the root vertex in the center.

Arguments:

graph: The graph object.

res: The result, the coordinates in a matrix. The parameter should point to an initialized matrix object and will be resized.

mode: Specifies which edges to consider when building the tree. If it is `IGRAPH_OUT` then only the outgoing, if it is `IGRAPH_IN` then only the incoming edges of a parent are considered. If it is `IGRAPH_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.

roots: The index of the root vertex or root vertices. The set of roots should be specified so that all vertices of the graph are reachable from them. Simply put, in the undirected case, one root should be given from each connected component. If *roots* is `NULL` or a pointer to an empty vector, then the roots will be selected automatically. Currently, automatic root selection prefers low eccentricity vertices in graphs with fewer than 500 vertices, and high degree vertices (according to *mode*) in larger graphs. The root selection heuristic may change without notice. To ensure a consistent output, please specify the roots manually.

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 not a null pointer and the *roots* argument is also given (and it is not a null pointer or an empty vector).

Returns:

Error code.

See also:

`igraph_layout_reingold_tilford()`.

`igraph_roots_for_tree_layout` — Roots suitable for a nice tree layout.

```
igraph_error_t igraph_roots_for_tree_layout(
        const igraph_t *graph,
        igraph_neimode_t mode,
        igraph_vector_int_t *roots,
        igraph_root_choice_t heuristic);
```

This function chooses a root, or a set of roots suitable for visualizing a tree, or a tree-like graph. It is typically used with `igraph_layout_reingold_tilford()`. The principle is to select a minimal set of roots so that all other vertices will be reachable from them.

In the undirected case, one root is chosen from each connected component. In the directed case, one root is chosen from each strongly connected component that has no incoming (or outgoing) edges (depending on 'mode'). When more than one root choice is possible, vertices are prioritized based on the given *heuristic*.

Arguments:

graph: The graph, typically a tree, but any graph is accepted.

mode: Whether to interpret the input as undirected, a directed out-tree or in-tree.

roots: An initialized integer vector, the roots will be returned here.

heuristic: The heuristic to use for breaking ties when multiple root choices are possible.

IGRAPH_ROOT_CHOICE_DEGREE	Choose the vertices with the highest degree (out- or in-degree in directed mode). This simple heuristic is fast even in large graphs.
IGRAPH_ROOT_CHOICE_ECCENTRICITY	Choose the vertices with the lowest eccentricity. This usually results in a "wide and shallow" tree layout. While this heuristic produces high-quality results, it is slow for large graphs: computing the eccentricities has quadratic complexity in the number of vertices.

Returns:

Error code.

Time complexity: depends on the heuristic.

igraph_layout_sugiyama — Sugiyama layout algorithm for layered directed acyclic graphs.

```
igraph_error_t igraph_layout_sugiyama(
    const igraph_t *graph, igraph_matrix_t *res, igraph_matrix_list_t *routing,
    const igraph_vector_int_t* layers, igraph_real_t hgap, igraph_real_t vgap,
    igraph_int_t maxiter, const igraph_vector_t *weights
);
```

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. The additional control points of the edges are returned in a separate list of matrices, one matrix per edge in the original graph. If an edge requires no additional control points, the corresponding matrix will be empty, otherwise the matrix will contain the coordinates of the control points, one point per row. When drawing the graph, edges should be drawn in a way that the curve representing the edge passes through the control points.

For more details, see 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.

Arguments:

graph: Pointer to an initialized graph object.

res: Pointer to an initialized matrix object. This will contain the result and will be resized as needed. The coordinates of the vertices in the layout will be stored in the rows of the matrix, one row per vertex.

routing: Pointer to an uninitialized list of matrices or NULL. When not NULL, the list will be resized as needed such that there will be one matrix for each edge of the graph, and the matrix will hold the additional control points that the edge must pass through, starting from the source vertex of the edge and ending at the target vertex. The matrix will have zero rows if an edge does not require control points.

layers: The layer index for each vertex or NULL if the layers should be determined automatically by igraph.

hgap: The preferred minimum horizontal gap between vertices in the same layer.

vgap: The distance between layers.

maxiter: 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: Weights of the edges. These are used only if the graph contains cycles; igraph will tend to reverse edges with smaller weights when breaking the cycles.

Returns:

Error code.

igraph_layout_umap — Layout using Uniform Manifold Approximation and Projection (UMAP).

```
igraph_error_t igraph_layout_umap(const igraph_t *graph,
                                  igraph_matrix_t *res,
                                  igraph_bool_t use_seed,
                                  const igraph_vector_t *distances,
                                  igraph_real_t min_dist,
                                  igraph_int_t epochs,
                                  igraph_bool_t distances_are_weights);
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

UMAP is mostly used to embed high-dimensional vectors in a low-dimensional space (most commonly 2D). The algorithm is probabilistic and introduces nonlinearities, unlike e.g. PCA and similar to T-distributed Stochastic Neighbor Embedding (t-SNE). Nonlinearity helps "cluster" very similar vectors together without imposing a global geometry on the embedded space (e.g. a rigid rotation + compression in PCA). UMAP uses graphs as intermediate data structures, hence it can be used as a graph layout algorithm as well.

The general UMAP workflow is to start from vectors, compute a sparse distance graph that only contains edges between simiar points (e.g. a k-nearest neighbors graph), and then convert these distances into exponentially decaying weights between 0 and 1 that are larger for points that are closest neighbors in the distance graph. If a graph without any distances associated to the edges is used, all weights will be set to 1.

If you are trying to use this function to embed high-dimensional vectors, you should first compute a k-nearest neighbors graph between your vectors and compute the associated distances, and then call this

function on that graph. If you already have a distance graph, or you have a graph with no distances, you can call this function directly. If you already have a graph with meaningful weights associated to each edge, you can also call this function, but set the argument *distances_are_weights* to true. To compute weights from distances without computing the layout, see igraph_layout_umap_compute_weights().

References:

Leland McInnes, John Healy, and James Melville: UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction (2020) https://arxiv.org/abs/1802.03426

Arguments:

graph:	Pointer to the graph to find a layout for (i.e. to embed). This is typically a sparse graph with only edges for the shortest distances stored, e.g. a k-nearest neighbors graph.
res:	Pointer to the n by 2 matrix where the layout coordinates will be stored.
use_seed:	If true the supplied values in the *res* argument are used as an initial layout, if false a random initial layout is used.
distances:	Pointer to a vector of distances associated with the graph edges. If this argument is NULL, all weights will be set to 1.
min_dist:	A fudge parameter that decides how close two unconnected vertices can be in the embedding before feeling a repulsive force. It must not be negative. Typical values are between 0 and 1.
epochs:	Number of iterations of the main stochastic gradient descent loop on the cross-entropy. Typical values are between 30 and 500.
distances_are_weights:	Whether to use precomputed weights. If true, the *distances* vector contains precomputed weights. If false (the typical use case), this function will compute weights from distances and then use them to compute the layout.

Returns:

Error code.

See also:

igraph_layout_umap_3d()

igraph_layout_umap_compute_weights — Compute weights for a UMAP layout starting from distances.

```
igraph_error_t igraph_layout_umap_compute_weights(
        const igraph_t *graph,
        const igraph_vector_t *distances,
        igraph_vector_t *weights);
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

UMAP is used to embed high-dimensional vectors in a low-dimensional space (most commonly 2D). It uses a distance graph as an intermediate data structure, making it also a useful graph layout algorithm. See `igraph_layout_umap()` for more information.

An early step in UMAP is to compute exponentially decaying "weights" from the distance graph. Connectivities can also be viewed as edge weights that quantify similarity between two vertices. This function computes weights from the distance graph. To compute the layout from precomputed weights, call `igraph_layout_umap()` with the *`distances_are_weights`* argument set to `true`.

While the distance graph can be directed (e.g. in a k-nearest neighbors, it is clear *whom* you are a neighbor of), the weights are usually undirected. Whenever two vertices are doubly connected in the distance graph, the resulting weight `W` is set as:

`W = W1 + W2 - W1 * W2` Because UMAP weights are interpreted as probabilities, this is just the probability that either edge is present, without double counting. It is called "fuzzy union" in the original UMAP implementation and is the default. One could also require that both edges are there, i.e. W = W1 * W2: this would represent the fuzzy intersection and is not implemented in igraph. As a consequence of this symmetrization, information is lost, i.e. one needs fewer weights than one had distances. To keep things efficient, here we set the weight for one of the two edges as above and the weight for its opposite edge as 0, so that it will be skipped in the UMAP gradient descent later on.

Technical note: For each vertex, this function computes its scale factor (sigma), its connectivity correction (rho), and finally the weights themselves.

References:

Leland McInnes, John Healy, and James Melville: UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction (2020) https://arxiv.org/abs/1802.03426

Arguments:

`graph`: Pointer to the distance graph. This can be directed (e.g. connecting each vertex to its neighbors in a k-nearest neighbor) or undirected, but must have no loops nor parallel edges. The only exception is: if the graph is directed, having pairs of edges with opposite direction is accepted.

`distances`: Pointer to the vector with the vertex-to-vertex distance associated with each edge. This argument can be NULL, in which case all edges are assumed to have the same distance.

`weights`: Pointer to an initialized vector where the result will be stored. If the input graph is directed, the weights represent a symmetrized version which contains less information. Therefore, whenever two edges between the same vertices and opposite direction are present in the input graph, only one of the weights is set and the other is fixed to zero. That format is accepted by `igraph_layout_umap()`, which skips all zero-weight edges from the layout optimization.

Returns:

Error code.

See also:

`igraph_layout_umap()`, `igraph_layout_umap_3d()`

3D layout generators

`igraph_layout_random_3d` — Places the vertices uniformly randomly in a cube.

```
igraph_error_t igraph_layout_random_3d(const igraph_t *graph, igraph_matrix_t *
```

Vertex coordinates range from -1 to 1, and are placed in three columns of a matrix, with a row for each vertex.

Arguments:

graph: The graph to place.

res: Pointer to an initialized matrix object. It will be resized to hold the result.

Returns:

Error code.

Added in version 0.2.

Time complexity: O(|V|), the number of vertices.

`igraph_layout_sphere` — Places vertices (more or less) uniformly on a sphere.

```
igraph_error_t igraph_layout_sphere(const igraph_t *graph, igraph_matrix_t *res
```

The vertices are placed with approximately equal spacing on a spiral wrapped around a sphere, in the order of their vertex IDs. Vertices with consecutive vertex IDs are placed near each other.

The algorithm was described in the following paper:

Distributing many points on a sphere by E.B. Saff and A.B.J. Kuijlaars, *Mathematical Intelligencer* 19.1 (1997) 5--11. https://doi.org/10.1007/BF03024331

Arguments:

graph: Pointer to an initialized graph object.

res: Pointer to an initialized matrix object. This will contain the result and will be resized as needed.

Returns:

Error code. The current implementation always returns with success.

Added in version 0.2.

Time complexity: O(|V|), the number of vertices in the graph.

igraph_layout_grid_3d — Places the vertices on a regular grid in the 3D space.

```
igraph_error_t igraph_layout_grid_3d(const igraph_t *graph, igraph_matrix_t *res
                          igraph_int_t width, igraph_int_t height);
```

Arguments:

graph: Pointer to an initialized graph object.

res: Pointer to an initialized matrix object. This will contain the result and will be resized as needed.

width: The number of vertices in a single row of the grid. When zero or negative, the width is determined automatically.

height: The number of vertices in a single column of the grid. When zero or negative, the height is determined automatically.

Returns:

Error code. The current implementation always returns with success.

Time complexity: O(|V|), the number of vertices.

igraph_layout_fruchterman_reingold_3d — 3D Fruchterman-Reingold algorithm.

```
igraph_error_t igraph_layout_fruchterman_reingold_3d(const igraph_t *graph,
        igraph_matrix_t *res,
        igraph_bool_t use_seed,
        igraph_int_t niter,
        igraph_real_t start_temp,
        const igraph_vector_t *weights,
        const igraph_vector_t *minx,
        const igraph_vector_t *maxx,
        const igraph_vector_t *miny,
        const igraph_vector_t *maxy,
        const igraph_vector_t *minz,
        const igraph_vector_t *maxz);
```

This is the 3D version of the force based Fruchterman-Reingold layout. See `igraph_layout_fruchterman_reingold()` for the 2D version.

Arguments:

graph: Pointer to an initialized graph object.

res: Pointer to an initialized matrix object. This will contain the result and will be resized as needed.

use_seed: If true the supplied values in the *res* argument are used as an initial layout, if false a random initial layout is used.

niter: The number of iterations to do. A reasonable default value is 500.

start_temp: 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.

weights: Pointer to a vector containing edge weights. Weights must be positive. If `NULL`, all edges are assumed to have weight 1. The attraction along the edges will be multiplied by the weights, resulting in vertices connected by a high-weight edge being placed closer together.

minx: Pointer to a vector, or a `NULL` pointer. If not a `NULL` pointer then the vector gives the minimum "x" coordinate for every vertex.

maxx: Same as *minx*, but the maximum "x" coordinates.

miny: Pointer to a vector, or a `NULL` pointer. If not a `NULL` pointer then the vector gives the minimum "y" coordinate for every vertex.

maxy: Same as *miny*, but the maximum "y" coordinates.

minz: Pointer to a vector, or a `NULL` pointer. If not a `NULL` pointer then the vector gives the minimum "z" coordinate for every vertex.

maxz: Same as *minz*, but the maximum "z" coordinates.

Returns:

Error code.

Added in version 0.2.

Time complexity: O(|V|^2) in each iteration, |V| is the number of vertices in the graph.

`igraph_layout_kamada_kawai_3d` — 3D version of the Kamada-Kawai layout generator.

```
igraph_error_t igraph_layout_kamada_kawai_3d(const igraph_t *graph, igraph_matr
                                     igraph_bool_t use_seed, igraph_int_t maxiter,
                                     igraph_real_t epsilon, igraph_real_t kkconst,
                                     const igraph_vector_t *weights,
                                     const igraph_vector_t *minx, const igraph_vec
                                     const igraph_vector_t *miny, const igraph_vec
                                     const igraph_vector_t *minz, const igraph_vec
```

This is the 3D version of `igraph_layout_kamada_kawai()`. See the documentation of that function for more information.

This layout algorithm is not suitable for large graphs. The memory requirements are of the order O(|V|^2).

Arguments:

graph: A graph object.

res: Pointer to an initialized matrix object. This will contain the result (x-, y- and z-positions in columns one through three) and will be resized if needed.

use_seed: Boolean, whether to use the values supplied in the *res* argument as the initial configuration. If zero and there are any limits on the z, y or z coordinates, then a random

initial configuration is used. Otherwise the vertices are placed uniformly on a sphere of radius 1 as the initial configuration.

maxiter: The maximum number of iterations to perform. A reasonable default value is at least ten (or more) times the number of vertices.

epsilon: Stop the iteration, if the maximum delta value of the algorithm is smaller than this. It is safe to leave it at zero, and then *maxiter* iterations are performed.

kkconst: The Kamada-Kawai vertex attraction constant. Typical value: number of vertices.

weights: A vector of edge weights. Weights are interpreted as edge *lengths* in the shortest path calculation used by the Kamada-Kawai algorithm. Therefore, vertices connected by high-weight edges will be placed further apart. Pass `NULL` to assume unit weights for all edges.

minx: Pointer to a vector, or a `NULL` pointer. If not a `NULL` pointer then the vector gives the minimum “x” coordinate for every vertex.

maxx: Same as *minx*, but the maximum “x” coordinates.

miny: Pointer to a vector, or a `NULL` pointer. If not a `NULL` pointer then the vector gives the minimum “y” coordinate for every vertex.

maxy: Same as *miny*, but the maximum “y” coordinates.

minz: Pointer to a vector, or a `NULL` pointer. If not a `NULL` pointer then the vector gives the minimum “z” coordinate for every vertex.

maxz: Same as *minz*, but the maximum “z” coordinates.

Returns:

Error code.

Time complexity: O(|V|) for each iteration, after an O(|V|^2 log|V|) initialization step. |V| is the number of vertices in the graph.

`igraph_layout_umap_3d` — 3D layout using UMAP.

```
igraph_error_t igraph_layout_umap_3d(const igraph_t *graph,
                                     igraph_matrix_t *res,
                                     igraph_bool_t use_seed,
                                     const igraph_vector_t *distances,
                                     igraph_real_t min_dist,
                                     igraph_int_t epochs,
                                     igraph_bool_t distances_are_weights);
```

Warning

This function is experimental and its signature is not considered final yet. We reserve the right to change the function signature without changing the major version of igraph. Use it at your own risk.

This is the 3D version of the UMAP algorithm (see `igraph_layout_umap()` for the 2D version).

Arguments:

graph: Pointer to the graph to find a layout for (i.e. to embed). This is typically a directed, sparse graph with only edges for the shortest distances stored, e.g. a k-nearest neighbors graph with the edges going from each focal vertex to its neighbors. However, it can also be an undirected graph. If the *distances_are_weights* is `true`, this is treated as an undirected graph.

res: Pointer to the n by 3 matrix where the layout coordinates will be stored.

use_seed: If true the supplied values in the *res* argument are used as an initial layout, if false a random initial layout is used.

distances: Pointer to a vector of distances associated with the graph edges. If this argument is `NULL`, all edges are assumed to have the same distance.

min_dist: A fudge parameter that decides how close two unconnected vertices can be in the embedding before feeling a repulsive force. It must not be negative. Typical values are between 0 and 1.

epochs: Number of iterations of the main stochastic gradient descent loop on the cross-entropy. Typical values are between 30 and 500.

distances_are_weights: Whether to use precomputed weights. If `false` (the typical use case), this function will compute weights from distances and then use them to compute the layout. If `true`, the *distances* vector contains precomputed weights, including possibly some weights equal to zero that are inconsequential for the layout optimization.

Returns:

Error code.

See also:

`igraph_layout_umap()`

Post-processing layouts

`igraph_layout_merge_dla` — Merges multiple layouts by using a DLA algorithm.

```
igraph_error_t igraph_layout_merge_dla(
    const igraph_vector_ptr_t *thegraphs, const igraph_matrix_list_t *coords,
    igraph_matrix_t *res
);
```

First each layout is covered by a circle. Then the layout of the largest graph is placed at the origin. Then the other layouts are placed by the DLA algorithm, larger ones first and smaller ones last.

Arguments:

thegraphs: Pointer vector containing the graph objects of which the layouts will be merged.

coords: List of matrices with the 2D layouts of the graphs in *thegraphs*.

res: Pointer to an initialized matrix object, the result will be stored here. It will be resized if needed.

Returns:

Error code.

Added in version 0.2.

Time complexity: TODO.

igraph_layout_align — Aligns a graph layout with the coordinate axes.

```
igraph_error_t igraph_layout_align(const igraph_t *graph, igraph_matrix_t *layou
```

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 `igraph_layout_fruchterman_reingold()`. Layouts in arbitrary dimensional spaces are supported.

Arguments:

graph: The graph whose layout is to be aligned.

layout: A matrix whose rows are the coordinates of vertices. It will be modified in-place.

Returns:

Error code.

Time complexity: O(|E| + |V|), linear in the number of edges and vertices.

Chapter 30. Processes on graphs

Epidemic models

`igraph_sir` — Performs a number of SIR epidemics model runs on a graph.

```
igraph_error_t igraph_sir(const igraph_t *graph, igraph_real_t beta,
              igraph_real_t gamma, igraph_int_t no_sim,
              igraph_vector_ptr_t *result);
```

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. See these parameters below.

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.

Arguments:

`graph`: The graph to perform the model on. For directed graphs edge directions are ignored and a warning is given.

`beta`: 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`: The rate of recovery of an infected individual. Formally, this is the rate parameter of an exponential distribution.

`no_sim`: The number of simulation runs to perform.

`result`: The result of the simulation is stored here, in a list of `igraph_sir_t` objects. To deallocate memory, the user needs to call `igraph_sir_destroy` on each element, before destroying the pointer vector itself using `igraph_vector_ptr_destroy_all()`.

Returns:

Error code.

Time complexity: O(no_sim * (|V| + |E| log(|V|))).

`igraph_sir_t` — The result of one SIR model simulation.

```
typedef struct igraph_sir_t {
    igraph_vector_t times;
    igraph_vector_int_t no_s, no_i, no_r;
} igraph_sir_t;
```

Data structure to store the results of one simulation of the SIR (susceptible-infected-recovered) model on a graph. It has the following members. They are all (real or integer) vectors, and they are of the same length.

Values:

`times`: A vector, the times of the events are stored here.

`no_s`: An integer vector, the number of susceptibles in each time step is stored here.

`no_i`: An integer vector, the number of infected individuals at each time step, is stored here.

`no_r`: An integer vector, the number of recovered individuals is stored here at each time step.

`igraph_sir_destroy` — Deallocates memory associated with a SIR simulation run.

```
void igraph_sir_destroy(igraph_sir_t *sir);
```

Arguments:

`sir`: The `igraph_sir_t` object storing the simulation.

Chapter 31. Reading and writing graphs from and to files

These functions can write a graph to a file, or read a graph from a file.

They assume that the current locale uses a decimal point and not a decimal comma. See `igraph_enter_safelocale()` and `igraph_exit_safelocale()` for more information.

Note that as **igraph** uses the traditional C streams, it is possible to read/write files from/to memory, at least on GNU operating systems supporting “non-standard” streams.

Simple edge list and similar formats

`igraph_read_graph_edgelist` — Reads an edge list from a file and creates a graph.

```
igraph_error_t igraph_read_graph_edgelist(igraph_t *graph, FILE *instream,
                                          igraph_int_t n, igraph_bool_t directed);
```

This format is simply a series of an even number of non-negative integers separated by whitespace. The integers represent vertex IDs. Placing each edge (i.e. pair of integers) on a separate line is not required, but it is recommended for readability. Edges of directed graphs are assumed to be in "from, to" order.

The largest vertex ID plus one, or the parameter *n* determines the vertex count, whichever is larger. See `igraph_read_graph_ncol()` for reading files where vertices are specified by name instead of by a numerical vertex ID.

Arguments:

graph: Pointer to an uninitialized graph object.

instream: Pointer to a stream, it should be readable.

n: The number of vertices in the graph. If smaller than the largest integer in the file it will be ignored. It is thus safe to supply zero here.

directed: If true the graph is directed, if false it will be undirected.

Returns:

Error code: `IGRAPH_PARSEERROR`: if there is a problem reading the file, or the file is syntactically incorrect.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges. It is assumed that reading an integer requires O(1) time.

`igraph_write_graph_edgelist` — Writes the edge list of a graph to a file.

```
igraph_error_t igraph_write_graph_edgelist(const igraph_t *graph, FILE *outstrea
```

Edges are represented as pairs of 0-based vertex indices. One edge is written per line, separated by a single space. For directed graphs edges are written in from, to order.

Arguments:

graph: The graph object to write.

outstream: Pointer to a stream, it should be writable.

Returns:

Error code: IGRAPH_EFILE if there is an error writing the file.

Time complexity: O(|E|), the number of edges in the graph. It is assumed that writing an integer to the file requires O(1) time.

igraph_read_graph_ncol — Reads an .ncol file used by LGL.

```
igraph_error_t igraph_read_graph_ncol(igraph_t *graph, FILE *instream,
                           const igraph_strvector_t *predefnames,
                           igraph_bool_t names,
                           igraph_add_weights_t weights,
                           igraph_bool_t directed);
```

Also useful for creating graphs from “named” (and optionally weighted) edge lists.

This format is used by the Large Graph Layout program (https://lgl.sourceforge.net), and it is simply a symbolic weighted edge list. It is a simple text file with one edge per line. An edge is defined by two symbolic vertex names separated by whitespace. The vertex names themselves cannot contain whitespace. They may be followed by an optional number, the weight of the edge; the number can be negative and can be in scientific notation. If there is no weight specified to an edge it is assumed to be zero.

The resulting graph is always undirected. LGL cannot deal with files which contain multiple or loop edges, this is however not checked here, as **igraph** is happy with these.

Arguments:

graph: Pointer to an uninitialized graph object.

instream: Pointer to a stream, it should be readable.

predefnames: Pointer to the symbolic names of the vertices in the file. If NULL 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 NULL and some unknown vertex names are found in the .ncol file then new vertex ids will be assigned to them.

names: Boolean value. If true, 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”. IGRAPH_ADD_WEIGHTS_YES adds the weights (even if they are not present in the file, in this case they are assumed to be 1). IGRAPH_ADD_WEIGHTS_NO does not add any edge attribute. IGRAPH_ADD_WEIGHTS_IF_PRESENT adds the attribute if and only if there is at least one

explicit edge weight in the input file, and edges without an explicit weight are assumed to have a weight of 1.

directed: Whether to create a directed graph. As this format was originally used only for undirected graphs there is no information in the file about the directedness of the graph. Set this parameter to `IGRAPH_DIRECTED` or `IGRAPH_UNDIRECTED` to create a directed or undirected graph.

Returns:

Error code: `IGRAPH_PARSEERROR`: if there is a problem reading the file, or the file is syntactically incorrect.

Time complexity: O(|V|+|E|log(|V|)) if we neglect the time required by the parsing. As usual |V| is the number of vertices, while |E| is the number of edges.

See also:

`igraph_read_graph_lgl()`, `igraph_write_graph_ncol()`

`igraph_write_graph_ncol` — Writes the graph to a file in `.ncol` format.

```
igraph_error_t igraph_write_graph_ncol(const igraph_t *graph, FILE *outstream,
                            const char *names, const char *weights);
```

`.ncol` is a format used by LGL, see `igraph_read_graph_ncol()` for details.

Note that having multiple or loop edges in an `.ncol` file breaks the LGL software but **igraph** does not check for this condition.

This format cannot represent zero-degree vertices.

Arguments:

graph: The graph to write.

outstream: The stream object to write to, it should be writable.

names: The name of a string vertex attribute, if symbolic names are to be written to the file. Supply `NULL` to write vertex ids instead.

weights: The name of a numerical edge attribute, which will be written as weights to the file. Supply `NULL` to skip writing edge weights.

Returns:

Error code: `IGRAPH_EFILE` if there is an error writing the file.

Time complexity: O(|E|), the number of edges. All file operations are expected to have time complexity O(1).

See also:

`igraph_read_graph_ncol()`, `igraph_write_graph_lgl()`

`igraph_read_graph_lgl` — Reads a graph from an `.lgl` file.

```
igraph_error_t igraph_read_graph_lgl(igraph_t *graph, FILE *instream,
                          igraph_bool_t names,
                          igraph_add_weights_t weights,
                          igraph_bool_t directed);
```

The `.lgl` format is used by the Large Graph Layout visualization software (https://lgl.sourceforge.net), it can describe undirected optionally weighted graphs. From the LGL manual:

> The second format is the LGL file format (`.lgl` file suffix). This is yet another graph file format that tries to be as stingy as possible with space, yet keeping the edge file in a human readable (not binary) format. The format itself is like the following:
>
> ```
>  # vertex1name
> vertex2name [optionalWeight]
> vertex3name [optionalWeight]
> ```
>
> Here, the first vertex of an edge is preceded with a pound sign '#'. Then each vertex that shares an edge with that vertex is listed one per line on subsequent lines.

LGL cannot handle loop and multiple edges or directed graphs, but in **igraph** it is not an error to have multiple and loop edges.

Arguments:

`graph`: Pointer to an uninitialized graph object.

`instream`: A stream, it should be readable.

`names`: Boolean value, if `true` 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”. `IGRAPH_ADD_WEIGHTS_YES` adds the weights (even if they are not present in the file, in this case they are assumed to be 1). `IGRAPH_ADD_WEIGHTS_NO` does not add any edge attribute. `IGRAPH_ADD_WEIGHTS_IF_PRESENT` adds the attribute if and only if there is at least one explicit edge weight in the input file, and edges without an explicit weight are assumed to have a weight of 1.

`directed`: Whether to create a directed graph. As this format was originally used only for undirected graphs there is no information in the file about the directedness of the graph. Set this parameter to `IGRAPH_DIRECTED` or `IGRAPH_UNDIRECTED` to create a directed or undirected graph.

Returns:

Error code: `IGRAPH_PARSEERROR`: if there is a problem reading the file, or the file is syntactically incorrect.

Time complexity: O(|V|+|E|log(|V|)) if we neglect the time required by the parsing. As usual |V| is the number of vertices, while |E| is the number of edges.

See also:

`igraph_read_graph_ncol()`, `igraph_write_graph_lgl()`

Example 31.1. File `examples/simple/igraph_read_graph_lgl.c`

`igraph_write_graph_lgl` — Writes the graph to a file in `.lgl` format.

```
igraph_error_t igraph_write_graph_lgl(const igraph_t *graph, FILE *outstream,
                            const char *names, const char *weights,
                            igraph_bool_t isolates);
```

`.lgl` is a format used by LGL, see `igraph_read_graph_lgl()` for details.

Note that having multiple or loop edges in an `.lgl` file breaks the LGL software but **igraph** does not check for this condition.

Arguments:

`graph`: The graph to write.

`outstream`: The stream object to write to, it should be writable.

`names`: The name of a string vertex attribute, if symbolic names are to be written to the file. Supply `NULL` to write vertex ids instead.

`weights`: The name of a numerical edge attribute, which will be written as weights to the file. Supply `NULL` to skip writing edge weights.

`isolates`: If `true`, isolated vertices are also written to the file. If `false`, they will be omitted.

Returns:

Error code: `IGRAPH_EFILE` if there is an error writing the file.

Time complexity: O(|E|), the number of edges if *`isolates`* is `false`, O(|V|+|E|) otherwise. All file operations are expected to have time complexity O(1).

See also:

`igraph_read_graph_lgl()`, `igraph_write_graph_ncol()`

Example 31.2. File `examples/simple/igraph_write_graph_lgl.c`

`igraph_read_graph_dimacs_flow` — Read a graph in DIMACS format.

```
igraph_error_t igraph_read_graph_dimacs_flow(
        igraph_t *graph, FILE *instream,
        igraph_strvector_t *problem,
        igraph_vector_int_t *label,
        igraph_int_t *source,
        igraph_int_t *target,
```

```
        igraph_vector_t *capacity,
        igraph_bool_t directed);
```

This function reads the DIMACS file format, more specifically the version for network flow problems, see the files at http://archive.dimacs.rutgers.edu/pub/netflow/general-info/

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`).

Within DIMACS files, vertex IDs are numbered from 1.

Arguments:

graph: Pointer to an uninitialized graph object.

instream: The file to read from.

problem: If not `NULL`, it will contain the problem type.

label: If not `NULL`, node labels will be stored here for `edge` problems. Ignored for `max` problems.

source: Pointer to an integer, the ID of the source node will be stored here. (The igraph vertex ID, which is one less than the actual number in the file.) It is ignored if `NULL`.

target: Pointer to an integer, the (igraph) ID of the target node will be stored here. It is ignored if `NULL`.

capacity: Pointer to an initialized vector, the capacity of the edges will be stored here if not \ NULL.

directed: Boolean, whether to create a directed graph.

Returns:

Error code.

Time complexity: O(|V|+|E|+c), the number of vertices plus the number of edges, plus the size of the file in characters.

See also:

`igraph_write_graph_dimacs_flow()`

`igraph_write_graph_dimacs_flow` — Write a graph in DIMACS format.

```
igraph_error_t igraph_write_graph_dimacs_flow(const igraph_t *graph, FILE *outs
```

```
                                    igraph_int_t source, igraph_int_t target,
                                    const igraph_vector_t *capacity);
```

This function writes a graph to an output stream in DIMACS format, describing a maximum flow problem. See ftp://dimacs.rutgers.edu/pub/netflow/general-info/

This file format is discussed in the documentation of `igraph_read_graph_dimacs_flow()`, see that for more information.

Arguments:

graph: The graph to write to the stream.

outstream: The stream.

source: Integer, the id of the source vertex for the maximum flow.

target: Integer, the id of the target vertex.

capacity: Pointer to an initialized vector containing the edge capacity values.

Returns:

Error code.

Time complexity: O(|E|), the number of edges in the graph.

See also:

`igraph_read_graph_dimacs_flow()`

Binary formats

`igraph_read_graph_graphdb` — Read a graph in the binary graph database format.

```
igraph_error_t igraph_read_graph_graphdb(igraph_t *graph, FILE *instream,
                                    igraph_bool_t directed);
```

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/

From the graph database homepage:

> The graphs are stored in a compact binary format, one graph per file. The file is composed of 16 bit words, which are represented using the so-called little-endian convention, i.e. the least significant byte of the word is stored first.
>
> Then, for each node, the file contains the list of edges coming out of the node itself. The list is represented by a word encoding its length, followed by a word for each edge, representing the destination node of the edge. Node numeration is 0-based, so the first node of the graph has index 0.

As of igraph 0.10, only unlabelled graphs are implemented.

References:

M. De Santo, P. Foggia, C. Sansone, and M. Vento: A large database of graphs and its use for benchmarking graph isomorphism algorithms. Pattern Recognition Letters, 24(8), 1067-1079 (2003). https://doi.org/10.1016/S0167-8655(02)00253-2

MIVIA ARG Dataset, https://zenodo.org/records/11204020, https://mivia.unisa.it/datasets/graph-database/arg-database/

Arguments:

graph: Pointer to an uninitialized graph object.

instream: The stream to read from. It should be opened in binary mode.

directed: Whether to create a directed graph.

Returns:

Error code.

Time complexity: O(|V|+|E|), the number of vertices plus the number of edges.

Example 31.3. File examples/simple/igraph_read_graph_graphdb.c

GraphML format

igraph_read_graph_graphml — Reads a graph from a GraphML file.

```
igraph_error_t igraph_read_graph_graphml(igraph_t *graph, FILE *instream, igraph
```

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. Attributes of the graph are loaded only if an attribute interface is attached, see `igraph_set_attribute_table()`. String attrribute values are returned in UTF-8 encoding.

Graph attribute names are taken from the `attr.name` attributes of the `key` tags in the GraphML file. Since `attr.name` is not mandatory, igraph will fall back to the `id` attribute of the `key` tag if `attr.name` is missing.

Arguments:

graph: Pointer to an uninitialized graph object.

instream: A stream, it should be readable.

index: If the GraphML file contains more than one graph, the one specified by this index will be loaded. Indices start from zero, so supply zero here if your GraphML file contains only a single graph.

Returns:

Error code: `IGRAPH_PARSEERROR`: if there is a problem reading the file, or the file is syntactically incorrect. `IGRAPH_UNIMPLEMENTED`: the GraphML functionality was disabled at compile-time

Example 31.4. File `examples/simple/graphml.c`

`igraph_write_graph_graphml` — Writes the graph to a file in GraphML format.

```
igraph_error_t igraph_write_graph_graphml(const igraph_t *graph, FILE *outstrea
                                  igraph_bool_t prefixattr);
```

GraphML is an XML-based file format for representing various types of graphs. See the GraphML Primer (http://graphml.graphdrawing.org/primer/graphml-primer.html) for the detailed format description.

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.

Arguments:

`graph`: The graph to write.

`outstream`: The stream object to write to, it should be writable.

`prefixattr`: Boolean value. Whether to put a prefix in front of the attribute names to ensure uniqueness if the graph has vertex and edge (or graph) attributes with the same name.

Returns:

Error code: `IGRAPH_EFILE` if there is an error writing the file.

Time complexity: O(|V|+|E|) otherwise. All file operations are expected to have time complexity O(1).

Example 31.5. File `examples/simple/graphml.c`

GML format

`igraph_read_graph_gml` — Read a graph in GML format.

```
igraph_error_t igraph_read_graph_gml(igraph_t *graph, FILE *instream);
```

GML is a simple textual format, see https://web.archive.org/web/20190207140002/http://www.fim.uni-passau.de/index.php?id=17297%26L=1 for details.

Although all syntactically correct GML can be parsed, we implement only a subset of this format. Some attributes might be ignored. Here is a list of all the differences:

1. Only attributes with a simple type are used: integer, real or string. If an attribute is composite, i.e. an array or a record, then it is ignored. When some values of the attribute are simple and some compound, the composite ones are replaced with a default value (NaN for numeric, `""` for string).

2. `comment` fields are not ignored. They are treated as any other field and converted to attributes.

3. Top level attributes except for `Version` and the first `graph` attribute are completely ignored.

4. There is no maximum line length or maximum keyword length.

5. Only the `quot`, `amp`, `apos`, `lt` and `gt` character entities are supported. Any other entity is passed through unchanged by the reader after issuing a warning, and is expected to be decoded by the user.

6. We allow `inf`, `-inf` and `nan` (not a number) as a real number. This is case insensitive, so `nan`, `NaN` and `NAN` are equivalent.

Please contact us if you cannot live with these limitations of the GML parser.

Arguments:

`graph`: Pointer to an uninitialized graph object.

`instream`: The stream to read the GML file from.

Returns:

Error code.

Time complexity: should be proportional to the length of the file.

See also:

`igraph_read_graph_graphml()` for a more modern format, `igraph_write_graph_gml()` for writing GML files.

Example 31.6. File `examples/simple/gml.c`

`igraph_write_graph_gml` — Write the graph to a stream in GML format.

```
igraph_error_t igraph_write_graph_gml(const igraph_t *graph, FILE *outstream,
                                      igraph_write_gml_sw_t options,
                                      const igraph_vector_t *id, const char *cr
```

GML is a quite general textual format, see https://web.archive.org/web/20190207140002/http://www.fim.uni-passau.de/index.php?id=17297%26L=1 for details.

The graph, vertex and edges attributes are written to the file as well, if they are numeric or string. Boolean attributes are converted to numeric, with 0 and 1 used for false and true, respectively. NaN values of numeric attributes are skipped, as NaN is not part of the GML specification and other software may not be able to read files containing them. This is consistent with `igraph_read_graph_gml()`, which produces NaN when an attribute value is missing. In contrast with NaN, infinite values are retained. Ensure that none of the numeric attributes values are infinite to produce a conformant GML file that can be read by other software.

As igraph is more forgiving about attribute names, it might be necessary to simplify the them before writing to the GML file. This way we'll have a syntactically correct GML file. The following simple procedure is performed on each attribute name: first the alphanumeric characters are extracted, the others are ignored. Then if the first character is not a letter then the attribute name is prefixed with “igraph”. Note that this might result identical names for two attributes, igraph does not check this.

The “id” vertex attribute is treated specially. If the *id* argument is not NULL then it should be a numeric vector with the vertex IDs and the “id” vertex attribute is ignored (if there is one). If *id* is NULL and there is a numeric “id” vertex attribute, it will be used instead. If ids are not specified in either way then the regular igraph vertex IDs are used. If some of the supplied id values are invalid (non-integer or NaN), all supplied id are ignored and igraph vertex IDs are used instead.

Note that whichever way vertex IDs are specified, their uniqueness is not checked.

If the graph has edge attributes that become “source” or “target” after encoding, or the graph has an attribute that becomes “directed”, they will be ignored with a warning. GML uses these attributes to specify the edge endpoints, and the graph directedness, so we cannot write them to the file. Rename them before calling this function if you want to preserve them.

Arguments:

graph: The graph to write to the stream.

outstream: The stream to write the file to.

options: Set of |-combinable boolean flags for writing the GML file.

0	All options turned off.
IGRAPH_WRITE_GML_DEFAULT_SW	Default options, currently equivalent to 0. May change in future versions.
IGRAPH_WRITE_GML_ENCODE_ONLY_QUOT_SW	Do not encode any other characters than " as entities. Specifically, this option prevents the encoding of &. Useful when re-exporting a graph that was read from a GML file in which igraph could not interpret all entities, and thus passed them through without decoding.

id: Either NULL or a numeric vector with the vertex IDs. See details above.

creator: An optional string to write to the stream in the creator line. If NULL, the igraph version with the current date and time is added. If "", the creator line is omitted. Otherwise, the supplied string is used verbatim.

Returns:

Error code.

Time complexity: should be proportional to the number of characters written to the file.

See also:

igraph_read_graph_gml() for reading GML files, igraph_read_graph_graphml() for a more modern format.

Example 31.7. File examples/simple/gml.c

Pajek format

igraph_read_graph_pajek — Reads a file in Pajek format.

```
igraph_error_t igraph_read_graph_pajek(igraph_t *graph, FILE *instream);
```

Only a subset of the Pajek format is implemented. This is partially because there is no formal specification for this format, but also because **igraph** does not support some Pajek features, like mixed graphs.

Starting from version 0.6.1 igraph reads bipartite (two-mode) graphs from Pajek files and adds the `type` Boolean vertex attribute for them. Warnings are given for invalid edges, i.e. edges connecting vertices of the same type.

The list of the current limitations:

1. Only `.net` files are supported, Pajek project files (`.paj`) are not.
2. Temporal networks (i.e. with time events) are not supported.
3. Graphs with both directed and non-directed edges are not supported, as they cannot be represented in **igraph**.
4. Only Pajek networks are supported; permutations, hierarchies, clusters and vectors are not.
5. Multi-relational networks (i.e. networks with multiple edge types) are not supported.
6. Unicode characters encoded as `&#dddd;`, or newlines encoded as `\n` will not be decoded.

If an attribute handler is installed, **igraph** also reads the vertex and edge attributes from the file. Most attributes are renamed to be more informative: `color` instead of `c`, `xfact` instead of `x_fact`, `yfact` instead of y_fact, `labeldist` instead of `lr`, `labeldegree2` instead of `lphi`, `framewidth` instead of `bw`, `fontsize` instead of `fos`, `rotation` instead of `phi`, `radius` instead of `r`, `diamondratio` instead of `q`, `labeldegree` instead of `la`, `color` instead of `ic`, `framecolor` instead of `bc`, `labelcolor` instead of `lc`; these belong to vertices.

Edge attributes are also renamed, `s` to `arrowsize`, `w` to `edgewidth`, `h1` to `hook1`, `h2` to `hook2`, `a1` to `angle1`, `a2` to `angle2`, `k1` to `velocity1`, `k2` to `velocity2`, `ap` to `arrowpos`, `lp` to `labelpos`, `lr` to `labelangle`, `lphi` to `labelangle2`, `la` to `labeldegree`, `fos` to `fontsize`, `a` to `arrowtype`, `p` to `linepattern`, `l` to `label`, `lc` to `labelcolor`, `c` to `color`.

Unknown vertex or edge parameters are read as string vertex or edge attributes. If the parameter name conflicts with one the standard attribute names mentioned above, a _ character is appended to it to avoid conflict.

In addition the following vertex attributes might be added: `name` is added (with the same value) if there are vertex IDs in the file. `x` and `y`, and potentially `z` are also added if there are vertex coordinates in the file.

The `weight` edge attribute will be added if there are edge weights present.

See the Pajek homepage: http://vlado.fmf.uni-lj.si/pub/networks/pajek/ for more info on Pajek. The Pajek manual, http://vlado.fmf.uni-lj.si/pub/networks/pajek/doc/pajekman.pdf, and http://mrvar.fdv.uni-lj.si/pajek/DrawEPS.htm have information on the Pajek file format. There is additional useful information and sample files at http://mrvar.fdv.uni-lj.si/pajek/history.htm

Arguments:

graph: Pointer to an uninitialized graph object.

instream: An already opened file handler.

Returns:

Error code.

Time complexity: O(|V|+|E|+|A|), |V| is the number of vertices, |E| the number of edges, |A| the number of attributes (vertex + edge) in the graph if there are attribute handlers installed.

See also:

`igraph_write_graph_pajek()` for writing Pajek files, `igraph_read_graph_graphml()` for reading GraphML files.

Example 31.8. File `examples/simple/foreign.c`

`igraph_write_graph_pajek` — Writes a graph to a file in Pajek format.

```
igraph_error_t igraph_write_graph_pajek(const igraph_t *graph, FILE *outstream)
```

Writes files in the native format of the Pajek software. This format is not recommended for data exchange or archival. It is meant solely for interoperability with Pajek.

The Pajek vertex and edge parameters (like color) are determined by the attributes of the vertices and edges. Of course this requires an attribute handler to be installed. The names of the corresponding vertex and edge attributes are listed at `igraph_read_graph_pajek()`, e.g. the `color` vertex attributes determines the color (`c` in Pajek) parameter.

Vertex and edge attributes that do not correspond to any documented Pajek parameter are discarded.

As of version 0.6.1 igraph writes bipartite graphs into Pajek files correctly, i.e. they will be also bipartite when read into Pajek. As Pajek is less flexible for bipartite graphs (the numeric IDs of the vertices must be sorted according to vertex type), igraph might need to reorder the vertices when writing a bipartite Pajek file. This effectively means that numeric vertex IDs usually change when a bipartite graph is written to a Pajek file, and then read back into igraph.

Early versions of Pajek supported only Windows-style line endings in Pajek files, but recent versions support both Windows and Unix line endings. igraph therefore uses the platform-native line endings when the input file is opened in text mode, and uses Unix-style line endings when the input file is opened in binary mode. If you are using an old version of Pajek, you are on Unix and you are having problems reading files written by igraph on a Windows machine, convert the line endings manually with a text editor or with `unix2dos` or `iconv` from the command line).

Pajek will only interpret UTF-8 encoded files if they contain a byte-order mark (BOM) at the beginning. igraph is agnostic of string attribute encodings and therefore it will never write a BOM. You need to add this manually if/when necessary.

Arguments:

`graph`: The graph object to write.

`outstream`: The file to write to. It should be opened and writable.

Returns:

Error code.

Time complexity: O(|V|+|E|+|A|), |V| is the number of vertices, |E| is the number of edges, |A| the number of attributes (vertex + edge) in the graph if there are attribute handlers installed.

See also:

`igraph_read_graph_pajek()` for reading Pajek graphs, `igraph_write_graph_graphml()` for writing a graph in GraphML format, this suites **igraph** graphs better.

Example 31.9. File `examples/simple/igraph_write_graph_pajek.c`

UCINET's DL file format

`igraph_read_graph_dl` — Reads a file in the DL format of UCINET.

```
igraph_error_t igraph_read_graph_dl(igraph_t *graph, FILE *instream,
                        igraph_bool_t directed);
```

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.

Arguments:

`graph`: Pointer to an uninitialized graph object.

`instream`: The stream to read the DL file from.

`directed`: Boolean, whether to create a directed file.

Returns:

Error code.

Time complexity: linear in terms of the number of edges and vertices, except for the matrix format, which is quadratic in the number of vertices.

Example 31.10. File `examples/simple/igraph_read_graph_dl.c`

Graphviz format

`igraph_write_graph_dot` — Write the graph to a stream in DOT format.

```
igraph_error_t igraph_write_graph_dot(const igraph_t *graph, FILE* outstream);
```

DOT is the format used by the widely known GraphViz software, see http://www.graphviz.org for details. The grammar of the DOT format can be found here: http://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.

Arguments:

graph: The graph to write to the stream.

outstream: The stream to write the file to.

Returns:

Error code.

Time complexity: should be proportional to the number of characters written to the file.

See also:

`igraph_write_graph_graphml()` for a more modern format.

Example 31.11. File `examples/simple/dot.c`

LEDA format

`igraph_write_graph_leda` — Write a graph in LEDA native graph format.

```
igraph_error_t igraph_write_graph_leda(const igraph_t *graph, FILE *outstream,
                                       const char *vertex_attr_name,
                                       const char *edge_attr_name);
```

This function writes a graph to an output stream in LEDA format. See http://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.

Arguments:

graph: The graph to write to the stream.

outstream: The stream.

vertex_attr_name: The name of the vertex attribute whose values are to be stored in the output, or `NULL` if no vertex attribute should be stored.

edge_attr_name: The name of the edge attribute whose values are to be stored in the output, or `NULL` if no edge attribute should be stored.

Returns:

Error code.

Time complexity: O(|V|+|E|), the number of vertices and edges in the graph.

Convenience functions for locale change

`igraph_enter_safelocale` — Temporarily set the C locale.

```
igraph_error_t igraph_enter_safelocale(igraph_safelocale_t *loc);
```

igraph's foreign format readers and writers require a locale that uses a decimal point instead of a decimal comma. This is a convenience function that temporarily sets the C locale so that readers and writers would work correctly. It *must* be paired with a call to `igraph_exit_safelocale()`, otherwise a memory leak will occur.

This function tries to set the locale for the current thread only on a best-effort basis. Restricting the locale change to a single thread is not supported on all platforms. In these cases, this function falls back to using the standard `setlocale()` function, which affects the entire process and is not safe to use from concurrent threads.

It is generally recommended to run igraph within a thread that has been permanently set to the C locale using system-specific means. This is a convenience function for situations when this is not easily possible because the programmer is not in control of the process, such as when developing plugins/extensions. Note that processes start up in the C locale by default, thus nothing needs to be done unless the locale has been changed away from the default.

Arguments:

`loc`: Pointer to a variable of type `igraph_safelocale_t`. The current locale will be stored here, so that it can be restored using `igraph_exit_safelocale()`.

Returns:

Error code.

Example 31.12. File `examples/simple/safelocale.c`

`igraph_exit_safelocale` — Temporarily set the C locale.

```
void igraph_exit_safelocale(igraph_safelocale_t *loc);
```

Restores a locale saved by `igraph_enter_safelocale()` and deallocates all associated data. This function *must* be paired with a call to `igraph_enter_safelocale()`.

Arguments:

`loc`: A variable of type `igraph_safelocale_t`, originally set by `igraph_enter_safelocale()`.

Chapter 32. Using BLAS, LAPACK and ARPACK for igraph matrices and graphs

BLAS interface in igraph

BLAS is a highly optimized library for basic linear algebra operations such as vector-vector, matrix-vector and matrix-matrix product. Please see http://www.netlib.org/blas/ for details and a reference implementation in Fortran. igraph contains some wrapper functions that can be used to call BLAS routines in a somewhat more user-friendly way. Not all BLAS routines are included in igraph, and even those which are included might not have wrappers; the extension of the set of wrapped functions will probably be driven by igraph's internal requirements. The wrapper functions usually substitute double-precision floating point arrays used by BLAS with igraph_vector_t and igraph_matrix_t instances and also remove those parameters (such as the number of rows/columns) that can be inferred from the passed arguments directly.

`igraph_blas_ddot` — Dot product of two vectors.

```
igraph_error_t igraph_blas_ddot(const igraph_vector_t *v1, const igraph_vector_
                        igraph_real_t *res);
```

Arguments:

v1: The first vector.

v2: The second vector.

res: Pointer to a real, the result will be stored here.

Returns:

Error code.

Time complexity: O(n) where n is the length of the vectors.

Example 32.1. File `examples/simple/blas.c`

`igraph_blas_dnrm2` — Euclidean norm of a vector.

```
igraph_real_t igraph_blas_dnrm2(const igraph_vector_t *v);
```

Arguments:

v: The vector.

Returns:

Real value, the norm of *v*.

Time complexity: O(n) where n is the length of the vector.

igraph_blas_dgemv — Matrix-vector multiplication using BLAS, vector version.

```
igraph_error_t igraph_blas_dgemv(igraph_bool_t transpose, igraph_real_t alpha,
                        const igraph_matrix_t *a, const igraph_vector_t *x,
                        igraph_real_t beta, igraph_vector_t *y);
```

This function is a somewhat more user-friendly interface to the `dgemv` function in BLAS. `dgemv` performs the operation `y = alpha*A*x + beta*y`, where *x* and *y* are vectors and *A* is an appropriately sized matrix (symmetric or non-symmetric).

Arguments:

transpose: Whether to transpose the matrix *A*.

alpha: The constant *alpha*.

a: The matrix *A*.

x: The vector *x*.

beta: The constant *beta*.

y: The vector *y* (which will be modified in-place). It must always have the correct length, but its elements need not be set when `beta=0`.

Time complexity: O(nk) if the matrix is of size n x k

Returns:

`IGRAPH_EOVERFLOW` if the matrix is too large for BLAS, `IGRAPH_SUCCESS` otherwise.

See also:

`igraph_blas_dgemv_array` if you have arrays instead of vectors.

Example 32.2. File `examples/simple/blas.c`

igraph_blas_dgemm — Matrix-matrix multiplication using BLAS.

```
igraph_error_t igraph_blas_dgemm(igraph_bool_t transpose_a, igraph_bool_t transp
        igraph_real_t alpha, const igraph_matrix_t *a, const igraph_matrix_t *b
        igraph_real_t beta, igraph_matrix_t *c);
```

This function is a somewhat more user-friendly interface to the `dgemm` function in BLAS. `dgemm` calculates alpha*a*b + beta*c, where a, b and c are matrices, of which a and b can be transposed.

Arguments:

transpose_a: whether to transpose the matrix *a*

transpose_b: whether to transpose the matrix *b*

alpha: the constant `alpha`

a: the matrix `a`

b: the matrix `b`

beta: the constant `beta`

c: the matrix `c`. The result will also be stored here. If beta is zero, c will be resized to fit the result.

Time complexity: O(n m k) where matrix a is of size n × k, and matrix b is of size k × m.

Returns:

`IGRAPH_EOVERFLOW` if the matrix is too large for BLAS, `IGRAPH_EINVAL` if the matrices have incompatible sizes, `IGRAPH_SUCCESS` otherwise.

Example 32.3. File `examples/simple/blas_dgemm.c`

`igraph_blas_dgemv_array` — Matrix-vector multiplication using BLAS, array version.

```
igraph_error_t igraph_blas_dgemv_array(igraph_bool_t transpose, igraph_real_t a
                              const igraph_matrix_t* a, const igraph_real_t* x,
                              igraph_real_t beta, igraph_real_t* y);
```

This function is a somewhat more user-friendly interface to the `dgemv` function in BLAS. `dgemv` performs the operation y = alpha*A*x + beta*y, where x and y are vectors and A is an appropriately sized matrix (symmetric or non-symmetric).

Arguments:

transpose: whether to transpose the matrix *A*

alpha: the constant *alpha*

a: the matrix *A*

x: the vector *x* as a regular C array

beta: the constant *beta*

y: the vector *y* as a regular C array (which will be modified in-place)

Time complexity: O(nk) if the matrix is of size n x k

Returns:

`IGRAPH_EOVERFLOW` if the matrix is too large for BLAS, `IGRAPH_SUCCESS` otherwise.

See also:

`igraph_blas_dgemv` if you have vectors instead of arrays.

LAPACK interface in igraph

LAPACK is written in Fortran90 and provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. The associated matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized

Schur) are also provided, as are related computations such as reordering of the Schur factorizations and estimating condition numbers. Dense and banded matrices are handled, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices, in both single and double precision.

igraph provides an interface to a very limited set of LAPACK functions, using the regular igraph data structures.

See more about LAPACK at http://www.netlib.org/lapack/

Matrix factorization, solving linear systems

`igraph_lapack_dgetrf` — LU factorization of a general M-by-N matrix.

```
igraph_error_t igraph_lapack_dgetrf(igraph_matrix_t *a, igraph_vector_int_t *ip
                          int *info);
```

The factorization has the form A = P * L * U where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n).

Arguments:

a: The input/output matrix. On entry, the M-by-N matrix to be factored. On exit, the factors L and U from the factorization A = P * L * U; the unit diagonal elements of L are not stored.

ipiv: An integer vector, the pivot indices are stored here, unless it is a null pointer. Row `i` of the matrix was interchanged with row `ipiv[i]`.

info: LAPACK error code. Zero on successful exit. If its value is a positive number i, it indicates that U(i,i) is exactly zero. The factorization has been completed, but the factor U is exactly singular, and division by zero will occur if it is used to solve a system of equations. If LAPACK returns an error, i.e. a negative info value, then an igraph error is generated as well.

Returns:

Error code.

Time complexity: TODO.

`igraph_lapack_dgetrs` — Solve general system of linear equations using LU factorization.

```
igraph_error_t igraph_lapack_dgetrs(igraph_bool_t transpose, const igraph_matri
                          const igraph_vector_int_t *ipiv, igraph_matrix_t *b);
```

This function calls LAPACK to solve a system of linear equations A * X = B or A' * X = B with a general N-by-N matrix A using the LU factorization computed by `igraph_lapack_dgetrf`.

Arguments:

transpose: Boolean, whether to transpose the input matrix.

a: A matrix containing the L and U factors from the factorization A = P*L*U. L is expected to be unitriangular, diagonal entries are those of U. If A is singular, no warning or error wil be given and random output will be returned.

ipiv: An integer vector, the pivot indices from `igraph_lapack_dgetrf()` must be given here. Row `i` of A was interchanged with row `ipiv[i]`.

b: The right hand side matrix must be given here. The solution will also be placed here.

Returns:

Error code.

Time complexity: TODO.

`igraph_lapack_dgesv` — Solve system of linear equations with LU factorization.

```
igraph_error_t igraph_lapack_dgesv(igraph_matrix_t *a, igraph_vector_int_t *ipi
                                   igraph_matrix_t *b, int *info);
```

This function computes the solution to a real system of linear equations A * X = B, where A is an N-by-N matrix and X and B are N-by-NRHS matrices.

The LU decomposition with partial pivoting and row interchanges is used to factor A as A = P * L * U, where P is a permutation matrix, L is unit lower triangular, and U is upper triangular. The factored form of A is then used to solve the system of equations A * X = B.

Arguments:

a: Matrix. On entry the N-by-N coefficient matrix, on exit, the factors L and U from the factorization A=P*L*U; the unit diagonal elements of L are not stored.

ipiv: An integer vector or a null pointer. If not a null pointer, then the pivot indices that define the permutation matrix P, are stored here. Row i of the matrix was interchanged with row IPIV(i).

b: Matrix, on entry the right hand side matrix should be stored here. On exit, if there was no error, and the info argument is zero, then it contains the solution matrix X.

info: The LAPACK info code. If it is positive, then U(info,info) is exactly zero. In this case the factorization has been completed, but the factor U is exactly singular, so the solution could not be computed.

Returns:

Error code.

Time complexity: TODO.

Example 32.4. File `examples/simple/igraph_lapack_dgesv.c`

Eigenvalues and eigenvectors of matrices

`igraph_lapack_dsyevr` — Selected eigenvalues and optionally eigenvectors of a symmetric matrix.

```
igraph_error_t igraph_lapack_dsyevr(const igraph_matrix_t *A,
                            igraph_lapack_dsyev_which_t which,
                            igraph_real_t vl, igraph_real_t vu, int vestimate,
                            int il, int iu, igraph_real_t abstol,
                            igraph_vector_t *values, igraph_matrix_t *vectors,
                            igraph_vector_int_t *support);
```

Calls the DSYEVR LAPACK function to compute selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

See more in the LAPACK documentation.

Arguments:

A: Matrix, on entry it contains the symmetric input matrix. Only the leading N-by-N upper triangular part is used for the computation.

which: Constant that gives which eigenvalues (and possibly the corresponding eigenvectors) to calculate. Possible values are `IGRAPH_LAPACK_DSYEV_ALL`, all eigenvalues; `IGRAPH_LAPACK_DSYEV_INTERVAL`, all eigenvalues in the half-open interval `(vl, vu]`; `IGRAPH_LAPACK_DSYEV_SELECT`, the il-th through iu-th eigenvalues.

vl: If *which* is `IGRAPH_LAPACK_DSYEV_INTERVAL`, then this is the lower bound of the interval to be searched for eigenvalues. See also the *vestimate* argument.

vu: If *which* is `IGRAPH_LAPACK_DSYEV_INTERVAL`, then this is the upper bound of the interval to be searched for eigenvalues. See also the *vestimate* argument.

vestimate: An upper bound for the number of eigenvalues in the `(vl, vu]` interval, if *which* is `IGRAPH_LAPACK_DSYEV_INTERVAL`. Memory is allocated only for the given number of eigenvalues (and eigenvectors), so this upper bound must be correct.

il: The index of the smallest eigenvalue to return, if *which* is `IGRAPH_LAPACK_DSYEV_SELECT`.

iu: The index of the largets eigenvalue to return, if *which* is `IGRAPH_LAPACK_DSYEV_SELECT`.

abstol: The absolute error tolerance for the eigevalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval `[a,b]` of width less than or equal to `abstol + EPS * max(|a|,|b|)`, where `EPS` is the machine precision.

values: An initialized vector, the eigenvalues are stored here, unless it is a null pointer. It will be resized as needed.

vectors: An initialized matrix. A set of orthonormal eigenvectors are stored in its columns, unless it is a null pointer. It will be resized as needed.

support: An integer vector. If not a null pointer, then it will be resized to (2*max(1,M)) (M is a the total number of eigenvalues found). Then the support of the eigenvectors in *vectors* is stored here, i.e., the indices indicating the nonzero elements in *vectors*. The i-th eigenvector is nonzero only in elements support(2*i-1) through support(2*i).

Returns:

Error code.

Time complexity: TODO.

Example 32.5. File `examples/simple/igraph_lapack_dsyevr.c`

`igraph_lapack_dgeev` — Eigenvalues and optionally eigenvectors of a non-symmetric matrix.

```
igraph_error_t igraph_lapack_dgeev(const igraph_matrix_t *A,
                        igraph_vector_t *valuesreal,
                        igraph_vector_t *valuesimag,
                        igraph_matrix_t *vectorsleft,
                        igraph_matrix_t *vectorsright,
                        int *info);
```

This function calls LAPACK to compute, for an N-by-N real nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors.

The right eigenvector v(j) of A satisfies A * v(j) = lambda(j) * v(j) where lambda(j) is its eigenvalue. The left eigenvector u(j) of A satisfies u(j)^H * A = lambda(j) * u(j)^H where u(j)^H denotes the conjugate transpose of u(j).

The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Arguments:

A: matrix. On entry it contains the N-by-N input matrix.

valuesreal: Pointer to an initialized vector, or a null pointer. If not a null pointer, then the real parts of the eigenvalues are stored here. The vector will be resized as needed.

valuesimag: Pointer to an initialized vector, or a null pointer. If not a null pointer, then the imaginary parts of the eigenvalues are stored here. The vector will be resized as needed.

vectorsleft: Pointer to an initialized matrix, or a null pointer. If not a null pointer, then the left eigenvectors are stored in the columns of the matrix. The matrix will be resized as needed.

vectorsright: Pointer to an initialized matrix, or a null pointer. If not a null pointer, then the right eigenvectors are stored in the columns of the matrix. The matrix will be resized as needed.

info: This argument is used for two purposes. As an input argument it gives whether an igraph error should be generated if the QR algorithm fails to compute all eigenvalues. If *info* is non-zero, then an error is generated, otherwise only a warning is given. On exit it contains the LAPACK error code. Zero means successful exit. A negative values means that some of the arguments had an illegal value, this always triggers an igraph error. An i positive value means that the QR algorithm failed to compute all the eigenvalues, and no eigenvectors have been computed; element i+1:N of *valuesreal* and *valuesimag* contain eigenvalues which have converged. This case only generates an igraph error, if *info* was non-zero on entry.

Returns:

Error code.

Time complexity: TODO.

Example 32.6. File `examples/simple/igraph_lapack_dgeev.c`

`igraph_lapack_dgeevx` — Eigenvalues/vectors of nonsymmetric matrices, expert mode.

```
igraph_error_t igraph_lapack_dgeevx(igraph_lapack_dgeevx_balance_t balance,
                          const igraph_matrix_t *A,
                          igraph_vector_t *valuesreal,
                          igraph_vector_t *valuesimag,
                          igraph_matrix_t *vectorsleft,
                          igraph_matrix_t *vectorsright,
                          int *ilo, int *ihi, igraph_vector_t *scale,
                          igraph_real_t *abnrm,
                          igraph_vector_t *rconde,
                          igraph_vector_t *rcondv,
                          int *info);
```

This function calculates the eigenvalues and optionally the left and/or right eigenvectors of a nonsymmetric N-by-N real matrix.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (*`ilo`*, *`ihi`*, *`scale`*, and *`abnrm`*), reciprocal condition numbers for the eigenvalues (*`rconde`*), and reciprocal condition numbers for the right eigenvectors (*`rcondv`*).

The right eigenvector v(j) of A satisfies A * v(j) = lambda(j) * v(j) where lambda(j) is its eigenvalue. The left eigenvector u(j) of A satisfies u(j)^H * A = lambda(j) * u(j)^H where u(j)^H denotes the conjugate transpose of u(j).

The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Balancing a matrix means permuting the rows and columns to make it more nearly upper triangular, and applying a diagonal similarity transformation D * A * D^(-1), where D is a diagonal matrix, to make its rows and columns closer in norm and the condition numbers of its eigenvalues and eigenvectors smaller. The computed reciprocal condition numbers correspond to the balanced matrix. Permuting rows and columns will not change the condition numbers (in exact arithmetic) but diagonal scaling will. For further explanation of balancing, see section 4.10.2 of the LAPACK Users' Guide. Note that the eigenvectors obtained for the balanced matrix are backtransformed to those of *`A`*.

Arguments:

`balance`: Indicates whether the input matrix should be balanced. Possible values:

`IGRAPH_LAPACK_DGEEVX_BALANCE_NONE`	no not diagonally scale or permute.
`IGRAPH_LAPACK_DGEEVX_BALANCE_PERM`	perform permutations to make the matrix more nearly upper triangular. Do not diagonally scale.
`IGRAPH_LAPACK_DGEEVX_BALANCE_SCALE`	diagonally scale the matrix, i.e. replace A by D*A*D^(-1), where D is a diagonal matrix, chosen to make the rows and columns of A more equal in norm. Do not permute.

`IGRAPH_LAPACK_DGEEVX_BALANCE_BOTH`	both diagonally scale and permute A.

A: The input matrix, must be square.

valuesreal: An initialized vector, or a `NULL` pointer. If not a `NULL` pointer, then the real parts of the eigenvalues are stored here. The vector will be resized, as needed.

valuesimag: An initialized vector, or a `NULL` pointer. If not a `NULL` pointer, then the imaginary parts of the eigenvalues are stored here. The vector will be resized, as needed.

vectorsleft: An initialized matrix or a `NULL` pointer. If not a null pointer, then the left eigenvectors are stored here. The order corresponds to the eigenvalues and the eigenvectors are stored in a compressed form. If the j-th eigenvalue is real then column j contains the corresponding eigenvector. If the j-th and (j+1)-th eigenvalues form a complex conjugate pair, then the j-th and (j+1)-th columns contain the real and imaginary parts of the corresponding eigenvectors.

vectorsright: An initialized matrix or a `NULL` pointer. If not a null pointer, then the right eigenvectors are stored here. The format is the same, as for the *vectorsleft* argument.

ilo:

ihi: if not NULL, *ilo* and *ihi* point to integer values determined when A was balanced. The balanced A(i,j) = 0 if I>J and J=1,...,ilo-1 or I=ihi+1,...,N.

scale: Pointer to an initialized vector or a NULL pointer. If not a NULL pointer, then details of the permutations and scaling factors applied when balancing *A*, are stored here. If P(j) is the index of the row and column interchanged with row and column j, and D(j) is the scaling factor applied to row and column j, then

```
scale(J) = P(J), for J =
1,...,ilo-1
```

```
scale(J) = D(J), for J =
ilo,...,ihi
```

```
scale(J) = P(J) for J =
ihi+1,...,N.
```

The order in which the interchanges are made is N to *ihi*+1, then 1 to *ilo*-1.

abnrm: Pointer to a real variable, the one-norm of the balanced matrix is stored here. (The one-norm is the maximum of the sum of absolute values of elements in any column.)

rconde: An initialized vector or a NULL pointer. If not a null pointer, then the reciprocal condition numbers of the eigenvalues are stored here.

rcondv: An initialized vector or a NULL pointer. If not a null pointer, then the reciprocal condition numbers of the right eigenvectors are stored here.

info: This argument is used for two purposes. As an input argument it gives whether an igraph error should be generated if the QR algorithm fails to compute all eigenvalues. If *info* is non-zero, then an error is generated, otherwise only a warning is given. On exit it contains the LAPACK error code. Zero means successful exit. A negative values means that some of the arguments had an illegal value, this always triggers an igraph error. An i positive value means that the

QR algorithm failed to compute all the eigenvalues, and no eigenvectors have been computed; element i+1:N of *valuesreal* and *valuesimag* contain eigenvalues which have converged. This case only generated an igraph error, if *info* was non-zero on entry.

Returns:

Error code.

Time complexity: TODO

Example 32.7. File examples/simple/igraph_lapack_dgeevx.c

ARPACK interface in igraph

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. Please see https://github.com/opencollab/arpack-ng for details.

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. A user-defined function of type igraph_arpack_function_t 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.

In igraph, eigenvalue/eigenvector calculations usually involve the following steps:

1. Initialization of an igraph_arpack_options_t data structure using igraph_arpack_options_init.
2. Setting some options in the initialized igraph_arpack_options_t object.
3. Defining a function of type igraph_arpack_function_t. The input of this function is a vector, and the output should be the output matrix multiplied by the input vector.
4. Calling igraph_arpack_rssolve() (is the matrix is symmetric), or igraph_arpack_rnsolve().

The igraph_arpack_options_t object can be used multiple times.

If we have many eigenvalue problems to solve, then it might worth to create an igraph_arpack_storage_t object, and initialize it via igraph_arpack_storage_init(). This structure contains all memory needed for ARPACK (with the given upper limit regarding to the size of the eigenvalue problem). Then many problems can be solved using the same igraph_arpack_storage_t object, without always reallocating the required memory. The igraph_arpack_storage_t object needs to be destroyed by calling igraph_arpack_storage_destroy() on it, when it is not needed any more.

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

Data structures

igraph_arpack_options_t — Options for ARPACK.

```
typedef struct igraph_arpack_options_t {
    /* INPUT */
    char bmat[1];          /* I-standard problem, G-generalized */
    int n;                 /* Dimension of the eigenproblem */
    char which[2];         /* LA, SA, LM, SM, BE */
    int nev;               /* Number of eigenvalues to be computed */
    igraph_real_t tol;     /* Stopping criterion */
    int ncv;               /* Number of columns in V */
    int ldv;               /* Leading dimension of V */
    int ishift;            /* 0-reverse comm., 1-exact with tridiagonal */
    int mxiter;            /* Maximum number of update iterations to take */
    int nb;                /* Block size on the recurrence, only 1 works */
    int mode;              /* The kind of problem to be solved (1-5)
                               1: A*x=l*x, A symmetric
                               2: A*x=l*M*x, A symm. M pos. def.
                               3: K*x = l*M*x, K symm., M pos. semidef.
                               4: K*x = l*KG*x, K s. pos. semidef. KG s. indef.
                               5: A*x = l*M*x, A symm., M symm. pos. semidef. */
    int start;             /* 0: random, 1: use the supplied vector */
    int lworkl;            /* Size of temporary storage, default is fine */
    igraph_real_t sigma;   /* The shift for modes 3,4,5 */
    igraph_real_t sigmai;  /* The imaginary part of shift for rnsolve */
    /* OUTPUT */
    int info;              /* What happened, see docs */
    int ierr;              /* What happened  in the dseupd call */
    int noiter;            /* The number of iterations taken */
    int nconv;
    int numop;             /* Number of OP*x operations */
    int numopb;            /* Number of B*x operations if BMAT='G' */
    int numreo;            /* Number of steps of re-orthogonalizations */
    /* INTERNAL */
    int iparam[11];
    int ipntr[14];
} igraph_arpack_options_t;
```

This data structure contains the options of the ARPACK eigenvalue solver routines. It must be initialized by calling `igraph_arpack_options_init()` on it. Then it can be used for multiple ARPACK calls, as the ARPACK solvers do not modify it. Input options:

Values:

`bmat`: Character. Whether to solve a standard ('I') ot a generalized problem ('B').

`n`: Dimension of the eigenproblem.

`which`: Specifies which eigenvalues/vectors to compute. Possible values for symmetric 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 en than from the low end.

Possible values for non-symmetric matrices:

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

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

`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.

`nev`: The number of eigenvalues to be computed.

`tol`: 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. Setting this to zero means that `igraph_arpack_rssolve` and `igraph_arpack_rnsolve` will determine a suitable value for `ncv` automatically.

`ldv`: Numberic scalar. It should be set to zero in the current igraph 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.

`mxiter`: 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. A*x=lambda*x, A is symmetric.
2. A*x=lambda*M*x, A is symmetric, M is symmetric positive definite.
3. K*x=lambda*M*x, K is symmetric, M is symmetric positive semi-definite.
4. K*x=lambda*KG*x, K is symmetric positive semi-definite, KG is symmetric indefinite.
5. A*x=lambda*M*x, 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. A*x=lambda*x.
2. A*x=lambda*M*x, M is symmetric positive definite.
3. A*x=lambda*M*x, M is symmetric semi-definite.
4. A*x=lambda*M*x, M is symmetric semi-definite.

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

`start`: Whether to use the supplied starting vector (1), or use a random starting vector (0). The starting vector must be supplied in the first column of the `vectors` argument of the `igraph_arpack_rssolve()` of `igraph_arpack_rnsolve()` call.

Output options:

Values:

`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 other error flags as well, but these are converted to igraph errors, see `igraph_error_type_t`.

`ierr`: Error flag of the second ARPACK call (one eigenvalue computation usually involves two calls to ARPACK). This is always zero, as other error codes are converted to igraph errors.

`noiter`: 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.

Internal options:

Values:

`lworkl`: Do not modify this option.

`sigma`: The shift for the shift-invert mode.

`sigmai`: The imaginary part of the shift, for the non-symmetric or complex shift-invert mode.

`iparam`: Do not modify this option.

`ipntr`: Do not modify this option.

`igraph_arpack_storage_t` — Storage for ARPACK.

```
typedef struct igraph_arpack_storage_t {
    int maxn, maxncv, maxldv;
    igraph_real_t *v;
    igraph_real_t *workl;
    igraph_real_t *workd;
    igraph_real_t *d;
    igraph_real_t *resid;
    igraph_real_t *ax;
    int *select;
    /* The following two are only used for non-symmetric problems: */
    igraph_real_t *di;
```

```
    igraph_real_t *workev;
} igraph_arpack_storage_t;
```

Public members, do not modify them directly, these are considered to be read-only.

Values:

`maxn`: Maximum rank of matrix.

`maxncv`: Maximum NCV.

`maxldv`: Maximum LDV.

These members are considered to be private:

Values:

`workl`: Working memory.

`workd`: Working memory.

`d`: Memory for eigenvalues.

`resid`: Memory for residuals.

`ax`: Working memory.

`select`: Working memory.

`di`: Memory for eigenvalues, non-symmetric case only.

`workev`: Working memory, non-symmetric case only.

`igraph_arpack_function_t` — Type of the ARPACK callback function.

```
typedef igraph_error_t igraph_arpack_function_t(igraph_real_t *to, const igraph_
                                        int n, void *extra);
```

Arguments:

`to`: Pointer to an `igraph_real_t`, the result of the matrix-vector product is expected to be stored here.

`from`: Pointer to an `igraph_real_t`, the input matrix should be multiplied by the vector stored here.

`n`: The length of the vector (which is the same as the order of the input matrix).

`extra`: Extra argument to the matrix-vector calculation function. This is coming from the `igraph_arpack_rssolve()` or `igraph_arpack_rnsolve()` function.

Returns:

Error code. If not `IGRAPH_SUCCESS`, then the ARPACK solver considers this as an error, stops and calls the igraph error handler.

`igraph_arpack_options_init` — Initialize ARPACK options.

```
void igraph_arpack_options_init(igraph_arpack_options_t *o);
```

Initializes ARPACK options, set them to default values. You can always pass the initialized `igraph_arpack_options_t` object to built-in igraph functions without any modification. The built-in igraph functions modify the options to perform their calculation, e.g. `igraph_pagerank()` always searches for the eigenvalue with the largest magnitude, regardless of the supplied value.

If you want to implement your own function involving eigenvalue calculation using ARPACK, however, you will likely need to set up the fields for yourself.

Arguments:

o: The `igraph_arpack_options_t` object to initialize.

Time complexity: O(1).

`igraph_arpack_storage_init` — Initialize ARPACK storage.

```
igraph_error_t igraph_arpack_storage_init(igraph_arpack_storage_t *s, igraph_int
                                  igraph_int_t maxncv, igraph_int_t maxldv,
                                  igraph_bool_t symm);
```

You only need this function if you want to run multiple eigenvalue calculations using ARPACK, and want to spare the memory allocation/deallocation between each two runs. Otherwise it is safe to supply a null pointer as the `storage` argument of both `igraph_arpack_rssolve()` and `igraph_arpack_rnsolve()` to make memory allocated and deallocated automatically.

Don't forget to call the `igraph_arpack_storage_destroy()` function on the storage object if you don't need it any more.

Arguments:

s: The `igraph_arpack_storage_t` object to initialize.

maxn: The maximum order of the matrices.

maxncv: The maximum NCV parameter intended to use.

maxldv: The maximum LDV parameter intended to use.

symm: Whether symmetric or non-symmetric problems will be solved using this `igraph_arpack_storage_t`. (You cannot use the same storage both with symmetric and non-symmetric solvers.)

Returns:

Error code.

Time complexity: O(maxncv*(maxldv+maxn)).

`igraph_arpack_storage_destroy` — Deallocate ARPACK storage.

```
void igraph_arpack_storage_destroy(igraph_arpack_storage_t *s);
```

Arguments:

s: The `igraph_arpack_storage_t` object for which the memory will be deallocated.

Time complexity: operating system dependent.

ARPACK solvers

`igraph_arpack_rssolve` — ARPACK solver for symmetric matrices.

```
igraph_error_t igraph_arpack_rssolve(igraph_arpack_function_t *fun, void *extra,
                           igraph_arpack_options_t *options,
                           igraph_arpack_storage_t *storage,
                           igraph_vector_t *values, igraph_matrix_t *vectors);
```

This is the ARPACK solver for symmetric matrices. Please use `igraph_arpack_rnsolve()` for non-symmetric matrices.

Arguments:

fun: Pointer to an `igraph_arpack_function_t` object, the function that performs the matrix-vector multiplication.

extra: An extra argument to be passed to `fun`.

options: An `igraph_arpack_options_t` object.

storage: An `igraph_arpack_storage_t` object, or a null pointer. In the latter case memory allocation and deallocation is performed automatically. Either this or the *vectors* argument must be non-null if the ARPACK iteration is started from a given starting vector. If both are given *vectors* take precedence.

values: If not a null pointer, then it should be a pointer to an initialized vector. The eigenvalues will be stored here. The vector will be resized as needed.

vectors: If not a null pointer, then it must be a pointer to an initialized matrix. The eigenvectors will be stored in the columns of the matrix. The matrix will be resized as needed. Either this or the *storage* argument must be non-null if the ARPACK iteration is started from a given starting vector. If both are given *vectors* take precedence.

Returns:

Error code.

Time complexity: depends on the matrix-vector multiplication. Usually a small number of iterations is enough, so if the matrix is sparse and the matrix-vector multiplication can be done in O(n) time (the number of vertices), then the eigenvalues are found in O(n) time as well.

`igraph_arpack_rnsolve` — ARPACK solver for non-symmetric matrices.

```
igraph_error_t igraph_arpack_rnsolve(igraph_arpack_function_t *fun, void *extra,
```

```
                    igraph_arpack_options_t *options,
                    igraph_arpack_storage_t *storage,
                    igraph_matrix_t *values, igraph_matrix_t *vectors);
```

Please always consider calling `igraph_arpack_rssolve()` if your matrix is symmetric, it is much faster. `igraph_arpack_rnsolve()` for non-symmetric matrices.

Note that ARPACK is not called for 2x2 matrices as an exact algebraic solution exists in these cases.

Arguments:

fun: Pointer to an `igraph_arpack_function_t` object, the function that performs the matrix-vector multiplication.

extra: An extra argument to be passed to `fun`.

options: An `igraph_arpack_options_t` object.

storage: An `igraph_arpack_storage_t` object, or a null pointer. In the latter case memory allocation and deallocation is performed automatically.

values: If not a null pointer, then it should be a pointer to an initialized matrix. The (possibly complex) eigenvalues will be stored here. The matrix will have two columns, the first column contains the real, the second the imaginary parts of the eigenvalues. The matrix will be resized as needed.

vectors: If not a null pointer, then it must be a pointer to an initialized matrix. The eigenvectors will be stored in the columns of the matrix. The matrix will be resized as needed. Note that real eigenvalues will have real eigenvectors in a single column in this matrix; however, complex eigenvalues come in conjugate pairs and the result matrix will store the eigenvector corresponding to the eigenvalue with *positive* imaginary part only. Since in this case the eigenvector is also complex, it will occupy *two* columns in the eigenvector matrix (the real and the imaginary parts, in this order). Caveat: if the eigenvalue vector returns only the eigenvalue with the *negative* imaginary part for a complex conjugate eigenvalue pair, the result vector will *still* store the eigenvector corresponding to the eigenvalue with the positive imaginary part (since this is how ARPACK works).

Returns:

Error code.

Time complexity: depends on the matrix-vector multiplication. Usually a small number of iterations is enough, so if the matrix is sparse and the matrix-vector multiplication can be done in O(n) time (the number of vertices), then the eigenvalues are found in O(n) time as well.

`igraph_arpack_unpack_complex` — Makes the result of the non-symmetric ARPACK solver more readable.

```
igraph_error_t igraph_arpack_unpack_complex(igraph_matrix_t *vectors, igraph_ma
                                  igraph_int_t nev);
```

This function works on the output of `igraph_arpack_rnsolve` and brushes it up a bit: it only keeps *nev* eigenvalues/vectors and every eigenvector is stored in two columns of the *vectors* matrix.

The output of the non-symmetric ARPACK solver is somewhat hard to parse, as real eigenvectors occupy only one column in the matrix, and the complex conjugate eigenvectors are not stored at all (usually). The other problem is that the solver might return more eigenvalues than requested. The

common use of this function is to call it directly after `igraph_arpack_rnsolve` with its *`vectors`* and *`values`* argument and `options`->nev as *`nev`*. This will add the vectors for eigenvalues with a negative imaginary part and return all vectors as 2 columns, a real and imaginary part.

Arguments:

`vectors`: The eigenvector matrix, as returned by `igraph_arpack_rnsolve`. It will be resized, typically it will be larger.

`values`: The eigenvalue matrix, as returned by `igraph_arpack_rnsolve`. It will be resized, typically extra, unneeded rows (=eigenvalues) will be removed.

`nev`: The number of eigenvalues/vectors to keep. Can be less or equal than the number originally requested from ARPACK.

Returns:

Error code.

Time complexity: linear in the number of elements in the *`vectors`* matrix.

Chapter 33. Non-graph related functions

igraph version number

`igraph_version` — The version of the igraph C library.

```
void igraph_version(const char **version_string,
                    int *major,
                    int *minor,
                    int *patch);
```

Arguments:

`version_string`: Pointer to a string pointer. If not `NULL`, it is set to the igraph version string, e.g. "0.10.13", "1.2.0", or "0.10.13-14-g997f59ad7". It consists of three dot-separated numerical parts and potentially of a dash-separated suffix, used in prerelease versions. This string must not be modified or deallocated.

`major`: If not a `NULL` pointer, then it is set to the major igraph version. E.g. for version "0.10.13" this is 0.

`minor`: If not a `NULL` pointer, then it is set to the minor igraph version. E.g. for version "0.10.13" this is 10.

`patch`: If not a `NULL` pointer, then it is set to the subminor igraph version. E.g. for version "0.10.13" this is 13.

Example 33.1. File `examples/simple/igraph_version.c`

Running mean of a time series

`igraph_running_mean` — Calculates the running mean of a vector.

```
igraph_error_t igraph_running_mean(const igraph_vector_t *data, igraph_vector_t
                        igraph_int_t binwidth);
```

The running mean is defined by the mean of the previous *`binwidth`* values.

Arguments:

`data`: The vector containing the data.

`res`: The vector containing the result. This should be initialized before calling this function and will be resized.

`binwidth`: Integer giving the width of the bin for the running mean calculation.

Returns:

Error code.

Time complexity: O(n), n is the length of the data vector.

Random sampling from very long sequences

`igraph_random_sample` — Generates an increasing random sequence of integers.

```
igraph_error_t igraph_random_sample(igraph_vector_int_t *res, igraph_int_t l, ig
                        igraph_int_t length);
```

This function generates an increasing sequence of random integer numbers from a given interval. The algorithm is taken literally from (Vitter 1987). This method can be used for generating numbers from a *very* large interval. It is primarily created for randomly selecting some edges from the sometimes huge set of possible edges in a large graph.

Reference:

J. S. Vitter. An efficient algorithm for sequential random sampling. ACM Transactions on Mathematical Software, 13(1):58--67, 1987. https://doi.org/10.1145/23002.23003

Arguments:

`res`: Pointer to an initialized vector. This will hold the result. It will be resized to the proper size.

`l`: The lower limit of the generation interval (inclusive). This must be less than or equal to the upper limit, and it must be integral.

`h`: The upper limit of the generation interval (inclusive). This must be greater than or equal to the lower limit, and it must be integral.

`length`: The number of random integers to generate.

Returns:

The error code `IGRAPH_EINVAL` is returned in each of the following cases: (1) The given lower limit is greater than the given upper limit, i.e. `l` > `h`. (2) Assuming that `l` < `h` and N is the sample size, the above error code is returned if N > |`h` - `l`|, i.e. the sample size exceeds the size of the candidate pool.

Time complexity: according to (Vitter 1987), the expected running time is O(length).

Example 33.2. File `examples/simple/igraph_random_sample.c`

Random sampling of spatial points

`igraph_rng_sample_sphere_surface` — Sample points uniformly from the surface of a sphere.

```
igraph_error_t igraph_rng_sample_sphere_surface(
    igraph_rng_t* rng, igraph_int_t dim, igraph_int_t n, igraph_real_t radius,
    igraph_bool_t positive, igraph_matrix_t *res
);
```

The center of the sphere is at the origin.

Arguments:

rng: The random number generator to use.

dim: The dimension of the random vectors.

n: The number of vectors to sample.

radius: Radius of the sphere, it must be positive.

positive: Whether to restrict sampling to the positive orthant.

res: Pointer to an initialized matrix, the result is stored here, each column will be a sampled vector. The matrix is resized, as needed.

Returns:

Error code.

Time complexity: O(n*dim*g), where g is the time complexity of generating a standard normal random number.

See also:

`igraph_rng_sample_sphere_volume()`, `igraph_rng_sample_dirichlet()` for other similar samplers.

`igraph_rng_sample_sphere_volume` — Sample points uniformly from the volume of a sphere.

```
igraph_error_t igraph_rng_sample_sphere_volume(
    igraph_rng_t* rng, igraph_int_t dim, igraph_int_t n, igraph_real_t radius,
    igraph_bool_t positive, igraph_matrix_t *res
);
```

The center of the sphere is at the origin.

Arguments:

rng: The random number generator to use.

dim: The dimension of the random vectors.

n: The number of vectors to sample.

radius: Radius of the sphere, it must be positive.

positive: Whether to restrict sampling to the positive orthant.

res: Pointer to an initialized matrix, the result is stored here, each column will be a sampled vector. The matrix is resized, as needed.

Returns:

Error code.

Time complexity: O(n*dim*g), where g is the time complexity of generating a standard normal random number.

See also:

`igraph_rng_sample_sphere_surface()`, `igraph_rng_sample_dirichlet()` for other similar samplers.

`igraph_rng_sample_dirichlet` — Sample points from a Dirichlet distribution.

```
igraph_error_t igraph_rng_sample_dirichlet(
    igraph_rng_t* rng, igraph_int_t n, const igraph_vector_t *alpha,
    igraph_matrix_t *res
);
```

Arguments:

rng: The random number generator to use.

n: The number of vectors to sample.

alpha: The parameters of the Dirichlet distribution. They must be positive. The length of this vector gives the dimension of the generated samples.

res: Pointer to an initialized matrix, the result is stored here, one sample in each column. It will be resized, as needed.

Returns:

Error code.

Time complexity: O(n * dim * g), where dim is the dimension of the sample vectors, set by the length of alpha, and g is the time complexity of sampling from a Gamma distribution.

See also:

`igraph_rng_sample_sphere_surface()` and `igraph_rng_sample_sphere_volume()` for other methods to sample latent vectors.

Fitting power-law distributions to empirical data

`igraph_plfit_result_t` — Result of fitting a power-law distribution to a vector.

```
typedef struct igraph_plfit_result_t {
    igraph_bool_t continuous;
    igraph_real_t alpha;
    igraph_real_t xmin;
    igraph_real_t L;
    igraph_real_t D;
    const igraph_vector_t* data;
} igraph_plfit_result_t;
```

This data structure contains the result of `igraph_power_law_fit()`, which tries to fit a power-law distribution to a vector of numbers. The structure contains the following members:

Values:

`continuous`: Whether the fitted power-law distribution was continuous or discrete.

`alpha`: The exponent of the fitted power-law distribution.

`xmin`: 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.

`L`: The log-likelihood of the fitted parameters; in other words, the probability of observing the input vector given the parameters.

`D`: The test statistic of a Kolmogorov-Smirnov test that compares the fitted distribution with the input vector. Smaller scores denote better fit.

`p`: The p-value of the Kolmogorov-Smirnov test; `NaN` if it has not been calculated yet. 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.

`data`: The vector containing the original input data. May not be valid any more if the caller already destroyed the vector.

`igraph_power_law_fit` — Fits a power-law distribution to a vector of numbers.

```
igraph_error_t igraph_power_law_fit(
    const igraph_vector_t* data, igraph_plfit_result_t* result,
    igraph_real_t xmin, igraph_bool_t force_continuous
);
```

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 alpha is greater than 1. In many real-world cases, the power-law behaviour kicks in only above a threshold value *xmin.* The goal of this functions is to determine *alpha* if *xmin* is given, or to determine *xmin* and the corresponding value of *alpha.*

The function uses the maximum likelihood principle to determine *alpha* for a given *xmin;* in other words, the function will return the *alpha* value for which the probability of drawing the given sample is the highest. When *xmin* is not given in advance, the algorithm will attempt to find the optimal *xmin* 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 the following reference for details:

Aaron Clauset, Cosma R. Shalizi and Mark E.J. Newman: Power-law distributions in empirical data. SIAM Review 51(4):661-703, 2009. https://doi.org/10.1137/070710111

Arguments:

data: vector containing the samples for which a power-law distribution is to be fitted. Note that you have to provide the *samples*, not the probability density function or the cumulative distribution function. For example, if you wish to fit a power-law to the degrees of a graph, you can use the output of `igraph_degree` directly as an input argument to `igraph_power_law_fit`

result: the result of the fitting algorithm. See `igraph_plfit_result_t` for more details. Note that the p-value of the fit is *not* calculated by default as it is time-consuming; you need to call `igraph_plfit_result_calculate_p_value()` to calculate the p-value itself

xmin: the minimum value in the sample vector where the power-law behaviour is expected to kick in. Samples smaller than `xmin` will be ignored by the algorithm. Pass zero here if you want to include all the samples. If `xmin` is negative, the algorithm will attempt to determine its best value automatically.

force_continuous: assume that the samples in the `data` argument come from a continuous distribution 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.

Returns:

Error code: `IGRAPH_ENOMEM`: not enough memory `IGRAPH_EINVAL`: one of the arguments is invalid `IGRAPH_EOVERFLOW`: overflow during the fitting process `IGRAPH_EUNDERFLOW`: underflow during the fitting process `IGRAPH_FAILURE`: the underlying algorithm signaled a failure without returning a more specific error code

Time complexity: in the continuous case, O(n log(n)) if `xmin` is given. In the discrete case, the time complexity is dominated by the complexity of the underlying L-BFGS algorithm that is used to optimize alpha. If `xmin` is not given, the time complexity is multiplied by the number of unique samples in the input vector (although it should be faster in practice).

Example 33.3. File `examples/simple/igraph_power_law_fit.c`

`igraph_plfit_result_calculate_p_value` — Calculates the p-value of a fitted power-law model.

```
igraph_error_t igraph_plfit_result_calculate_p_value(
    const igraph_plfit_result_t* model, igraph_real_t* result, igraph_real_t pr
);
```

The p-value is calculated by resampling the input data many times in a way that the part below the fitted `x_min` threshold is resampled from the input data itself, while the part above the fitted `x_min` threshold is drawn from the fitted power-law function. A Kolmogorov-Smirnov test is then performed for each resampled dataset and its test statistic is compared with the observed test statistic from the

original dataset. The fraction of resampled datasets that have a *higher* test statistic is the returned p-value.

Note that the precision of the returned p-value 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.

If igraph is compiled with OpenMP support, this function will use parallel OpenMP threads for the resampling. Each OpenMP thread gets its own instance of a random number generator. However, since the scheduling of OpenMP threads is outside our control, we cannot guarantee how many resampling instances the threads are asked to execute, thus it may happen that the random number generators are used differently between runs. If you want to obtain reproducible results, seed igraph's master RNG appropriately, and force the number of OpenMP threads to 1 early in your program, either by calling `omp_set_num_threads(1)` or by setting the value of the `OMP_NUM_THREADS` environment variable to 1.

Arguments:

`model`: The fitted power-law model from the `igraph_power_law_fit()` function

`result`: The calculated p-value is returned here

`precision`: The desired precision of the p-value. Higher values correspond to longer calculation time.

Returns:

Error code.

Comparing floats with a tolerance

`igraph_cmp_epsilon` — Compare two double-precision floats with a tolerance.

```
int igraph_cmp_epsilon(double a, double b, double eps);
```

Determines whether two double-precision floats are "almost equal" to each other with a given level of tolerance on the relative error.

The function supports infinities and NaN values. NaN values are considered not equal to any other value (even another NaN), but the ordering is arbitrary; in other words, we only guarantee that comparing a NaN with any other value will not return zero. Positive infinity is considered to be greater than any finite value with any tolerance. Negative infinity is considered to be smaller than any finite value with any tolerance. Positive infinity is considered to be equal to another positive infinity with any tolerance. Negative infinity is considered to be equal to another negative infinity with any tolerance.

Arguments:

`a`: The first float.

`b`: The second float.

`eps`: The level of tolerance on the relative error. The relative error is defined as `abs(a-b) / (abs(a) + abs(b))`. The two numbers are considered equal if this is less than `eps`. Negative epsilon values are not allowed; the returned value will be undefined in this case. Zero means to do an exact comparison without tolerance.

Returns:

Zero if the two floats are nearly equal to each other within the given level of tolerance, positive number if the first float is larger, negative number if the second float is larger.

`igraph_almost_equals` — Compare two double-precision floats with a tolerance.

```
igraph_bool_t igraph_almost_equals(double a, double b, double eps);
```

Determines whether two double-precision floats are "almost equal" to each other with a given level of tolerance on the relative error.

Arguments:

a: The first float.

b: The second float.

eps: The level of tolerance on the relative error. The relative error is defined as `abs(a-b) / (abs(a) + abs(b))`. The two numbers are considered equal if this is less than `eps`.

Returns:

True if the two floats are nearly equal to each other within the given level of tolerance, false otherwise.

`igraph_complex_almost_equals` — Compare two complex numbers with a tolerance.

```
igraph_bool_t igraph_complex_almost_equals(igraph_complex_t a,
                                           igraph_complex_t b,
                                           igraph_real_t eps);
```

Determines whether two complex numbers are "almost equal" to each other with a given level of tolerance on the relative error.

Arguments:

a: The first complex number.

b: The second complex number.

eps: The level of tolerance on the relative error. The relative error is defined as `abs(a-b) / (abs(a) + abs(b))`. The two numbers are considered equal if this is less than `eps`.

Returns:

True if the two complex numbers are nearly equal to each other within the given level of tolerance, false otherwise.

Chapter 34. Advanced igraph programming

Using igraph in multi-threaded programs

The igraph library is considered thread-safe if it has been compiled with thread-local storage enabled, i.e. the `IGRAPH_ENABLE_TLS` setting was toggled to `ON` and the current platform supports this feature. To check whether an igraph build is thread-safe, use the `IGRAPH_THREAD_SAFE` macro. When linking to external versions of igraph's dependencies, it is the responsibility of the user to check that these dependencies were also compiled to be thread-safe.

`IGRAPH_THREAD_SAFE` — Specifies whether igraph was built in thread-safe mode.

```
#define IGRAPH_THREAD_SAFE
```

This macro is defined to 1 if the current build of the igraph library is built in thread-safe mode, and 0 if it is not. A thread-safe igraph library attempts to use thread-local data structures instead of global ones, but note that this is not (and can not) be guaranteed for third-party libraries that igraph links to.

Thread-safe ARPACK library

Note that igraph is only thread-safe if it was built with the internal ARPACK library, i.e. the one that comes with igraph. The standard ARPACK library is not thread-safe.

Thread-safety of random number generators

The default random number generator that igraph uses is *not* guaranteed to be thread-safe. You need to set a different random number generator instance for every thread that you want to use igraph from. This is especially important if you set the seed of the random number generator to ensure reproducibility; sharing a random number generator between threads would break reproducibility as the order in which the various threads are scheduled is random, and therefore they would still receive random numbers in an unpredictable order from the shared random number generator.

Progress handlers

About progress handlers

It is often useful to report the progress of some long calculation, to allow the user to follow the computation and guess the total running time. A couple of igraph functions support this at the time of writing, hopefully more will support it in the future.

To see the progress of a computation, the user has to install a progress handler, as there is none installed by default. If an igraph function supports progress reporting, then it calls the installed progress handler periodically, and passes a percentage value to it, the percentage of computation already performed. To install a progress handler, you need to call `igraph_set_progress_handler()`. Currently there is a single pre-defined progress handler, called `igraph_progress_handler_stderr()`.

Setting up progress handlers

`igraph_progress_handler_t` — Type of progress handler functions

```
typedef igraph_error_t igraph_progress_handler_t(const char *message, igraph_re
                                                 void *data);
```

This is the type of the igraph progress handler functions. There is currently one such predefined function, `igraph_progress_handler_stderr()`, but the user can write and set up more sophisticated ones.

Arguments:

`message`: A string describing the function or algorithm that is reporting the progress. Current igraph functions always use the name *`message`* argument if reporting from the same function.

`percent`: Numeric, the percentage that was completed by the algorithm or function.

`data`: User-defined data. Current igraph functions that report progress pass a null pointer here. Users can write their own progress handlers and functions with progress reporting, and then pass some meaningfull context here.

Returns:

If the return value of the progress handler is not `IGRAPH_SUCCESS`, then `igraph_progress()` returns the error code from the progress handler intact. The `IGRAPH_PROGRESS()` macro also frees all allocated memory.

`igraph_set_progress_handler` — Install a progress handler, or remove the current handler.

```
igraph_progress_handler_t *
igraph_set_progress_handler(igraph_progress_handler_t new_handler);
```

There is a single simple predefined progress handler: `igraph_progress_handler_stderr()`.

Arguments:

`new_handler`: Pointer to a function of type `igraph_progress_handler_t`, the progress handler function to install. To uninstall the current progress handler, this argument can be a null pointer.

Returns:

Pointer to the previously installed progress handler function.

Time complexity: O(1).

`igraph_progress_handler_stderr` — A simple predefined progress handler.

```
igraph_error_t igraph_progress_handler_stderr(const char *message, igraph_real_
                                      void* data);
```

This simple progress handler first prints *message*, and then the percentage complete value in a short message to standard error.

Arguments:

message: A string describing the function or algorithm that is reporting the progress. Current igraph functions always use the same *message* argument if reporting from the same function.

percent: Numeric, the percentage that was completed by the algorithm or function.

data: User-defined data. Current igraph functions that report progress pass a null pointer here. Users can write their own progress handlers and functions with progress reporting, and then pass some meaningfull context here.

Returns:

This function always returns with `IGRAPH_SUCCESS`.

Time complexity: O(1).

Invoking the progress handler

`IGRAPH_PROGRESS` — Report the progress of a calculation from an igraph function (macro variant).

```
#define IGRAPH_PROGRESS(message, percent, data)
```

The standard way to report progress from an igraph function

Arguments:

message: A string, a textual message that references the calculation under progress.

percent: Numeric scalar, the percentage that is complete.

data: User-defined data, this can be used in user-defined progress handler functions, from user-written igraph functions.

Returns:

If the return value of the progress handler is not `IGRAPH_SUCCESS`, then `igraph_progress()` returns the error code from the progress handler intact. The `IGRAPH_PROGRESS()` macro also frees all allocated memory.

`IGRAPH_PROGRESSF` — Report the progress of a calculation from an igraph function, printf-like (macro variant).

```
#define IGRAPH_PROGRESSF(args)
```

This is the more flexible version of `IGRAPH_PROGRESS()`, having a printf-like syntax. As this macro takes variable number of arguments, they must be all supplied as a single argument, enclosed in parentheses. `igraph_progressf()` is then called with the given arguments.

Arguments:

args: The arguments to pass to `igraph_progressf()`.

Returns:

If the progress handler returns with a value other than `IGRAPH_SUCCESS`, then the function that called this macro returns as well, with the same error code, after cleaning up all allocated memory as needed.

`igraph_progress` — Report the progress of a calculation from an igraph function.

```
igraph_error_t igraph_progress(const char *message, igraph_real_t percent, void
```

Note that the usual way to report progress is the `IGRAPH_PROGRESS` macro, as that takes care of the return value of the progress handler.

Arguments:

message: A string describing the function or algorithm that is reporting the progress. Current igraph functions always use the name *message* argument if reporting from the same function.

percent: Numeric, the percentage that was completed by the algorithm or function.

data: User-defined data. Current igraph functions that report progress pass a null pointer here. Users can write their own progress handlers and functions with progress reporting, and then pass some meaningfull context here.

Returns:

Error code from the progress handler function, or `IGRAPH_SUCCESS` if no progress handler function was registered.

Time complexity: O(1).

`igraph_progressf` — Report the progress of a calculation from an igraph function, printf-like.

```
igraph_error_t igraph_progressf(const char *message, igraph_real_t percent, voi
                      ...);
```

This is a more flexible version of `igraph_progress()`, with a printf-like template string. First the template string is filled with the additional arguments and then `igraph_progress()` is called.

Note that there is an upper limit for the length of the *message* string, currently 1000 characters.

Arguments:

message: A string describing the function or algorithm that is reporting the progress. For this function this is a template string, using the same syntax as the standard `libc` `printf` function.

percent: Numeric, the percentage that was completed by the algorithm or function.

data: User-defined data. Current igraph functions that report progress pass a null pointer here. Users can write their own progress handlers and functions with progress reporting, and then pass some meaningfull context here.

...: Additional argument that were specified in the *message* argument.

Returns:

Error code from the progress handler function, or IGRAPH_SUCCESS if no progress handler function was registered. \return

Writing progress handlers

To write a new progress handler, one needs to create a function of type igraph_progress_handler_t. The new progress handler can then be installed with the igraph_set_progress_handler() function.

One can assume that the first progress handler call from a calculation will be call with zero as the *percentage* argument, and the last call from a function will have 100 as the *percentage* argument. Note, however, that if an error happens in the middle of a computation, then the 100 percent call might be omitted.

Writing igraph functions with progress reporting

If you want to write a function that uses igraph and supports progress reporting, you need to include igraph_progress() calls in your function, usually via the IGRAPH_PROGRESS() macro.

It is good practice to always include a call to igraph_progress() with a zero *percentage* argument, before the computation; and another call with 100 *percentage* value after the computation is completed.

It is also good practice *not* to call igraph_progress() too often, as this would slow down the computation. It might not be worth to support progress reporting in functions with linear or log-linear time complexity, as these are fast, even with a large amount of data. For functions with quadratic or higher time complexity make sure that the time complexity of the progress reporting is constant or at least linear. In practice this means having at most O(n) progress checks and at most 100 igraph_progress() calls.

Multi-threaded programs

In multi-threaded programs, each thread has its own progress handler, if thread-local storage is supported and igraph is thread-safe. See the IGRAPH_THREAD_SAFE macro for checking whether an igraph build is thread-safe.

Status handlers

Status reporting

In addition to the possibility of reporting the progress of an igraph computation via igraph_progress(), it is also possible to report simple status messages from within igraph functions, without having to judge how much of the computation was performed already. For this one needs to install a status handler function.

Status handler functions must be of type igraph_status_handler_t and they can be installed by a call to igraph_set_status_handler(). Currently there is a simple predefined status handler function, called igraph_status_handler_stderr(), but the user can define new ones.

igraph functions report their status via a call to the `IGRAPH_STATUS()` or the `IGRAPH_STATUSF()` macro.

Setting up status handlers

`igraph_status_handler_t` — The type of the igraph status handler functions

```
typedef igraph_error_t igraph_status_handler_t(const char *message, void *data)
```

Arguments:

`message`: The status message.

`data`: Additional context, with user-defined semantics. Existing igraph functions pass a null pointer here.

Returns:

Error code. The current calculation will abort if you return anything else than `IGRAPH_SUCCESS` here.

`igraph_set_status_handler` — Install of uninstall a status handler function.

```
igraph_status_handler_t *
igraph_set_status_handler(igraph_status_handler_t new_handler);
```

To uninstall the currently installed status handler, call this function with a null pointer.

Arguments:

`new_handler`: The status handler function to install.

Returns:

The previously installed status handler function.

Time complexity: O(1).

`igraph_status_handler_stderr` — A simple predefined status handler function.

```
igraph_error_t igraph_status_handler_stderr(const char *message, void *data);
```

A simple status handler function that writes the status message to the standard error.

Arguments:

`message`: The status message.

`data`: Additional context, with user-defined semantics. Existing igraph functions pass a null pointer here.

Returns:

Error code.

Time complexity: O(1).

Invoking the status handler

IGRAPH_STATUS — Report the status of an igraph function.

```
#define IGRAPH_STATUS(message, data)
```

Typically this function is called only a handful of times from an igraph function. E.g. if an algorithm has three major steps, then it is logical to call it three times, to signal the three major steps.

Arguments:

message: The status message.

data: Additional context, with user-defined semantics. Existing igraph functions pass a null pointer here.

Returns:

If the status handler returns with a value other than IGRAPH_SUCCESS, then the function that called this macro returns as well, with the same error code, after cleaning up all allocated memory as needed.

IGRAPH_STATUSF — Report the status from an igraph function

```
#define IGRAPH_STATUSF(args)
```

This is the more flexible version of IGRAPH_STATUS(), having a printf-like syntax. As this macro takes variable number of arguments, they must be all supplied as a single argument, enclosed in parentheses. igraph_statusf() is then called with the given arguments.

Arguments:

args: The arguments to pass to igraph_statusf().

Returns:

If the status handler returns with a value other than IGRAPH_SUCCESS, then the function that called this macro returns as well, with the same error code, after cleaning up all allocated memory as needed.

igraph_status — Reports status from an igraph function.

```
igraph_error_t igraph_status(const char *message, void *data);
```

It calls the installed status handler function, if there is one. Otherwise it does nothing. Note that the standard way to report the status from an igraph function is the IGRAPH_STATUS or IGRAPH_S-

TATUSF macro, as these take care of cleaning up allocated memory from the calling function if the status handler returns with an error code.

Arguments:

message: The status message.

data: Additional context, with user-defined semantics. Existing igraph functions pass a null pointer here.

Returns:

Error code from the status handler function, or IGRAPH_SUCCESS if no status handler function was registered.

Time complexity: O(1).

igraph_statusf — Report status, more flexible printf-like version.

```
igraph_error_t igraph_statusf(const char *message, void *data, ...);
```

This is the more flexible version of igraph_status(), that has a syntax similar to the printf standard C library function. It substitutes the values of the additional arguments into the *message* template string and calls igraph_status().

Arguments:

message: Status message template string, the syntax is the same as for the printf function.

data: Additional context, with user-defined semantics. Existing igraph functions pass a null pointer here.

...: The additional arguments to fill the template given in the *message* argument.

Returns:

Error code from the status handler function, or IGRAPH_SUCCESS if no status handler function was registered.

Chapter 35. Glossary

This glossary defines common terms used throughout the igraph documentation.

- **attribute**: A piece of data associated with a vertex, an edge, or the graph itself. The igraph C library currently supports numeric, string and Boolean attribute values, and provides a means for implementing attribute handlers that support custom types.
- **adjacent**: Two vertices are called **adjacent** if there is an edge connecting them. This term describes a vertex-to-vertex relation.
- **adjacency list**: A data structure that associates a list of neighbours (i.e. adjacent vertices) to each vertex.
- **adjacency matrix**: A representation of a graph as a square matrix. `A_ij` gives the number of edge endpoints connecting from the `i`th vertex to the `j`th vertex. Conventionally, the diagonal of the adjacency matrix of an undirected graph contains *twice* the number of self-loops. All igraph functions follow this convention unless noted otherwise.
- **biadjacency matrix**: Analogous to the adjacency matrix, but used for bipartite graphs. Element `B_ij` gives the number of edges from the `i`th vertex of the first group to the `j`th vertex of the second group.
- **bipartite graph**: A graph whose vertices can be partitioned into two groups in such a way that connections are present only between members of different groups.
- **complete graph**: Also called **full graph** within the context of igraph, a graph in which all pairs of vertices are connected to each other.
- **connected graph**: A connected graph consists of a single component, in which any vertex is reachable from any other. In igraph, the null graph is not considered connected, as it has not one, but zero components.
- **edge**: A **connection** between two vertices, also called a **link**. In igraph, edges are referred to by integer indices called **edge IDs**.
- **finalizer stack**: A global stack used internally by igraph to keep track of currently allocated objects and their destructors, so that they can be automatically destroyed in case of an error.
- **game**: Within igraph, this term is used for stochastic graph generators, i.e. functions that sample from random graph models.
- **graph** or **network**: A set of vertices with connections between them. In igraph, graphs may carry associated data in the form of vertex, edge or graph attributes.
- **incident**: An edge is called **incident** to the vertices that are its endpoints. This term describes a vertex-to-edge relation.
- **incidence list**: A data structure that associates a list of incident edges to each vertex.
- **incidence matrix**: A matrix describing the incidence relation between vertices (rows) and edges (columns).
- **membership vector**: Membership vectors are a means of encoding a partitioning of items, usually vertices, into several groups. The `i`th element of the vector gives an integer identifier of the group the `i`th vertex belongs to. Membership vectors are typically used to describe a vertex clustering obtained through community detection, or by identifying the connected components of a graph.
- **multi-edges** or **parallel edges**: More than one edge connecting the same two vertices. In a directed graph, `a -> b, a -> b` are considered parallel edges, but `a -> b, a <- b` are not.
- **null graph**: A graph with no vertices (and no edges).
- **self-loop**, **self-edge**, or simply **loop**: An edge that connects a vertex to itself.
- **simple graph**: A graph that does not have self-loops or multi-edges.
- **singleton graph**: A graph having a single vertex. This term usually refers to a single vertex with no edges, but note that self-loops may in principle be present.
- **vertex**: Graphs consist of vertices, also called **nodes**, that are connected to each other. In igraph, vertices are referred to by integer indices called **vertex IDs**.

Chapter 36. Licenses for igraph and this manual

THE GNU GENERAL PUBLIC LICENSE

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may

add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.>
Copyright (C) <year> <name of author>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by

the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) year name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program
`Gnomovision' (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

The GNU Free Documentation License

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ

stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section

of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by)

any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in

addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

G.1.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
> Permission is granted to copy, distribute and/or modify this document
> under the terms of the GNU Free Documentation License, Version 1.2
> or any later version published by the Free Software Foundation;
> with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
> A copy of the license is included in the section entitled "GNU
> Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the
> Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

A

add_edge, 27
add_edges, 28
add_vertices, 29
adhesion, 596
adjacency, 283
adjacency_spectral_embedding, 644
adjlist, 286
adjlist_clear, 183
adjlist_destroy, 182
adjlist_get, 182
adjlist_init, 180
adjlist_init_complementer, 181
adjlist_init_empty, 181
adjlist_init_from_inclist, 182
adjlist_simplify, 184
adjlist_size, 183
adjlist_sort, 183
all_minimal_st_separators, 600
all_st_cuts, 588
all_st_mincuts, 589
almost_equals, 717
are_adjacent, 388
arpack_function_t, 705
arpack_options_init, 706
arpack_options_t, 701
arpack_rnsolve, 707
arpack_rssolve, 707
arpack_storage_destroy, 706
arpack_storage_init, 706
arpack_storage_t, 704
arpack_unpack_complex, 708
articulation_points, 438
ASSERT, 46
assortativity, 497
assortativity_degree, 498
assortativity_nominal, 496
astar_heuristic_func_t, 405
asymmetric_preference_game, 319
atlas, 308
attribute_combination, 252
attribute_combination_add, 250
attribute_combination_destroy, 251
attribute_combination_init, 250
attribute_combination_remove, 251
attribute_combination_type_t, 251
attribute_elemtype_t, 245
attribute_record_destroy, 249
attribute_record_init, 246
attribute_record_init_copy, 246
attribute_record_resize, 247
attribute_record_set_default_boolean, 249
attribute_record_set_default_numeric, 248
attribute_record_set_default_string, 249
attribute_record_set_name, 247
attribute_record_set_type, 248
attribute_record_size, 247
attribute_record_t, 246
attribute_table_t, 240
attribute_type_t, 244
automorphism_group, 558
automorphism_group_bliss, 562
average_local_efficiency, 424
average_path_length, 410
avg_nearest_neighbor_degree, 499

B

barabasi_aging_game, 323
barabasi_game, 321
beta_weighted_gabriel_graph, 361
betweenness, 447
betweenness_cutoff, 461
betweenness_subset, 463
bfs, 382
bfshandler_t, 384
bfs_simple, 383
biadjacency, 351
bibcoupling, 473
biconnected_components, 437
bipartite_game_gnm, 347
bipartite_game_gnp, 348
bipartite_iea_game, 350
bipartite_projection, 354
bipartite_projection_size, 353
bitset_and, 199
bitset_capacity, 203
bitset_countl_one, 201
bitset_countl_zero, 200
bitset_countr_one, 201
bitset_countr_zero, 201
bitset_destroy, 195
bitset_fill, 198
bitset_init, 194
bitset_init_copy, 195
bitset_is_all_one, 202
bitset_is_all_zero, 202
bitset_is_any_one, 202
bitset_is_any_zero, 202
bitset_not, 200
bitset_null, 198
bitset_or, 199
bitset_popcount, 200
bitset_reserve, 203
bitset_resize, 204
bitset_size, 203
bitset_update, 204
bitset_xor, 199
BIT_CLEAR, 197
BIT_MASK, 195
BIT_NSLOTS, 198
BIT_SET, 196
BIT_SLOT, 196

BIT_TEST, 197
blas_ddot, 692
blas_dgemm, 693
blas_dgemv, 693
blas_dgemv_array, 694
blas_dnrm2, 692
bliss_info_t, 560
bliss_sh_t, 559
bond_percolation, 440
bridges, 439

C

callaway_traits_game, 328
calloc, 48
canonical_permutation, 558
canonical_permutation_bliss, 563
cattribute_EAB, 263
cattribute_EABV, 264
cattribute_EAB_set, 272
cattribute_EAB_setv, 276
cattribute_EAN, 261
cattribute_EANV, 262
cattribute_EAN_set, 271
cattribute_EAN_setv, 276
cattribute_EAS, 265
cattribute_EASV, 265
cattribute_EAS_set, 272
cattribute_EAS_setv, 277
cattribute_GAB, 255
cattribute_GAB_set, 267
cattribute_GAN, 255
cattribute_GAN_set, 266
cattribute_GAS, 256
cattribute_GAS_set, 267
cattribute_has_attr, 254
cattribute_list, 254
cattribute_remove_all, 279
cattribute_remove_e, 279
cattribute_remove_g, 278
cattribute_remove_v, 278
cattribute_VAB, 258
cattribute_VABV, 259
cattribute_VAB_set, 269
cattribute_VAB_setv, 274
cattribute_VAN, 257
cattribute_VANV, 258
cattribute_VAN_set, 268
cattribute_VAN_setv, 273
cattribute_VAS, 260
cattribute_VASV, 261
cattribute_VAS_set, 270
cattribute_VAS_setv, 275
centralization, 465
centralization_betweenness, 467
centralization_betweenness_tmax, 470
centralization_closeness, 468
centralization_closeness_tmax, 471
centralization_degree, 466
centralization_degree_tmax, 469
centralization_eigenvector_centrality, 468
centralization_eigenvector_centrality_tmax, 472
CHECK, 42
CHECK_CALLBACK, 43
chung_lu_game, 335
circle_beta_skeleton, 360
circulant, 294
cited_type_game, 329
citing_cited_type_game, 330
cliques, 534
cliques_callback, 535
clique_handler_t, 536
clique_number, 541
clique_size_hist, 534
closeness, 444
closeness_cutoff, 459
cmp_epsilon, 716
cocitation, 473
cohesion, 597
cohesive_blocks, 598
coloring_greedy_t, 579
community_eb_get_merges, 621
community_edge_betweenness, 619
community_fastgreedy, 622
community_fluid_communities, 628
community_infomap, 630
community_label_propagation, 628
community_leading_eigenvector, 614
community_leading_eigenvector_callback_t, 616
community_leiden, 624
community_leiden_simple, 626
community_multilevel, 623
community_optimal_modularity, 605
community_spinglass, 610
community_spinglass_single, 612
community_to_membership, 606
community_voronoi, 632
community_walktrap, 618
compare_communities, 608
complementer, 369
complex_almost_equals, 717
compose, 370
connected_components, 433
connect_neighborhood, 371
constraint, 453
contract_vertices, 371
convergence_degree, 458
convex_hull_2d, 363
copy, 18
coreness, 507
correlated_game, 320
correlated_pair_game, 320
count_adjacent_triangles, 550
count_automorphisms, 557
count_automorphisms_bliss, 561
count_isomorphisms_vf2, 565
count_loops, 493

count_multiple, 495
count_multiple_1, 495
count_reachable, 436
count_subisomorphisms_vf2, 570
count_triangles, 550
create, 282
create_bipartite, 346
cycle_graph, 293
cycle_handler_t, 524

D

decompose, 434
degree, 26
degree_1, 27
degree_correlation_vector, 500
degree_sequence_game, 332
DELALL, 280
delaunay_graph, 357
DELEA, 279
DELEAS, 280
delete_edges, 29
delete_vertices, 29
delete_vertices_map, 30
DELGA, 278
DELGAS, 280
DELVA, 279
DELVAS, 280
density, 512
destroy, 19
de_bruijn, 309
dfs, 385
dfshandler_t, 386
diameter, 411
difference, 369
dim_select, 646
disjoint_union, 364
disjoint_union_many, 364
distances, 389
distances_bellman_ford, 393
distances_cutoff, 390
distances_dijkstra, 391
distances_dijkstra_cutoff, 392
distances_floyd_warshall, 395
distances_johnson, 394
diversity, 514
dominator_tree, 585
dot_product_game, 343
dqueue_back, 153
dqueue_clear, 152
dqueue_destroy, 151
dqueue_empty, 152
dqueue_full, 152
dqueue_get, 153
dqueue_head, 153
dqueue_init, 151
dqueue_pop, 154
dqueue_pop_back, 154
dqueue_push, 154
dqueue_size, 152
dyad_census, 548

E

EAB, 263
EABV, 264
EAN, 262
EANV, 263
EAS, 265
EASV, 266
ecc, 487
eccentricity, 413
ecount, 20
ECOUNT_MAX, 31
edge, 20
edgelist_percolation, 441
edges, 21
edge_betweenness, 448
edge_betweenness_cutoff, 462
edge_betweenness_subset, 464
edge_connectivity, 593
edge_disjoint_paths, 595
edge_type_sw_t, 345
eigenvector_centrality, 455
eit_create, 237
eit_destroy, 238
EIT_END, 238
EIT_GET, 239
EIT_NEXT, 238
EIT_RESET, 239
EIT_SIZE, 238
empty, 17
empty_attrs, 18
enter_safelocale, 691
erdos_renyi_game_gnm, 312
erdos_renyi_game_gnp, 313
ERROR, 41
error, 41
ERRORF, 41
errorf, 42
error_handler_abort, 35
error_handler_ignore, 35
error_handler_printignore, 35
error_handler_t, 34
error_t, 35
error_type_t, 35
ess_1, 234
ess_all, 233
ess_none, 233
ess_range, 235
ess_vector, 234
establishment_game, 328
es_1, 229
es_all, 227
es_all_between, 229
es_as_vector, 235
es_copy, 235
es_destroy, 236

es_incident, 228
es_is_all, 236
es_none, 229
es_pairs, 231
es_pairs_small, 231
es_path, 232
es_range, 230
es_size, 236
es_type, 237
es_vector, 230
es_vector_copy, 233
eulerian_cycle, 529
eulerian_path, 530
even_tarjan_reduction, 601
exit_safelocale, 691
expand_path_to_pairs, 31
extended_chordal_ring, 295

F

famous, 306
FATAL, 46
fatal, 46
FATALF, 46
fatalf, 47
fatal_handler_abort, 45
fatal_handler_t, 45
feedback_arc_set, 527
feedback_vertex_set, 528
FINALLY, 43
FINALLY_CLEAN, 44
FINALLY_FREE, 44
find_cycle, 522
forest_fire_game, 331
free, 49
FROM, 21
from_hrg_dendrogram, 641
from_prufer, 299
full, 303
full_bipartite, 346
full_citation, 304
full_multipartite, 304
fundamental_cycles, 530

G

GAB, 256
gabriel_graph, 358
GAN, 255
GAS, 257
generalized_petersen, 310
get_adjacency, 516
get_adjacency_sparse, 517
get_all_eids_between, 24
get_all_shortest_paths, 405
get_all_shortest_paths_dijkstra, 407
get_all_simple_paths, 409
get_biadjacency, 352
get_edgelist, 519
get_eid, 22
get_eids, 23
get_isomorphisms_vf2, 566
get_isomorphisms_vf2_callback, 567
get_k_shortest_paths, 408
get_laplacian, 489
get_laplacian_sparse, 490
get_shortest_path, 398
get_shortest_paths, 396
get_shortest_paths_bellman_ford, 401
get_shortest_paths_dijkstra, 399
get_shortest_path_astar, 404
get_shortest_path_bellman_ford, 403
get_shortest_path_dijkstra, 400
get_stochastic, 518
get_stochastic_sparse, 519
get_subisomorphisms_vf2, 571
get_subisomorphisms_vf2_callback, 572
get_widest_path, 418
get_widest_paths, 419
girth, 412
global_efficiency, 422
gomory_hu_tree, 591
graphlets, 634
graphlets_candidate_basis, 635
graphlets_project, 635
graph_center, 414
graph_count, 576
graph_power, 372
grg_game, 342
growing_random_game, 327

H

harmonic_centrality, 446
harmonic_centrality_cutoff, 460
has_loop, 492
has_multiple, 494
has_mutual, 516
heap_clear, 156
heap_delete_top, 157
heap_destroy, 155
heap_empty, 156
heap_init, 155
heap_init_array, 155
heap_push, 156
heap_reserve, 158
heap_size, 157
heap_top, 157
hexagonal_lattice, 291
hrg_consensus, 639
hrg_create, 641
hrg_dendrogram, 643
hrg_destroy, 638
hrg_fit, 639
hrg_game, 641
hrg_init, 638
hrg_predict, 642
hrg_resize, 639

hrg_sample, 640
hrg_size, 638
hrg_t, 637
hsbm_game, 316
hsbm_list_game, 317
hub_and_authority_scores, 457
hypercube, 288

I

iea_game, 314
igraph_bool_t, 16
IGRAPH_INTEGER_MAX, 16
IGRAPH_INTEGER_MIN, 16
igraph_int_t, 16
igraph_real_t, 16
IGRAPH_UINT_MAX, 16
IGRAPH_UINT_MIN, 16
igraph_uint_t, 16
incident, 25
inclist_clear, 186
inclist_destroy, 185
inclist_get, 185
inclist_init, 184
inclist_size, 185
independence_number, 547
independent_vertex_sets, 545
induced_subgraph, 375
induced_subgraph_edges, 377
induced_subgraph_map, 376
intersection, 367
intersection_many, 368
invalidate_cache, 32
invert_permutation, 577
isoclass, 574
isoclass_create, 576
isoclass_subgraph, 575
isocompat_t, 568
isohandler_t, 568
isomorphic, 556
isomorphic_bliss, 560
isomorphic_vf2, 564
is_acyclic, 525
is_biconnected, 439
is_bigraphical, 443
is_bipartite, 355
is_bipartite_coloring, 580
is_chordal, 509
is_clique, 533
is_complete, 533
is_connected, 433
is_dag, 525
is_directed, 20
is_edge_coloring, 581
is_eulerian, 529
is_forest, 483
is_graphical, 442
is_independent_vertex_set, 544
is_loop, 492
is_matching, 509
is_maximal_matching, 510
is_minimal_separator, 599
is_multiple, 493
is_mutual, 515
is_perfect, 581
is_same_graph, 32
is_separator, 599
is_simple, 491
is_tree, 482
is_vertex_coloring, 580

J

join, 365
joint_degree_distribution, 503
joint_degree_matrix, 504
joint_type_distribution, 502

K

kary_tree, 296
kautz, 309
k_regular_game, 334

L

lapack_dgeev, 698
lapack_dgeevx, 699
lapack_dgesv, 696
lapack_dgetrf, 695
lapack_dgetrs, 695
lapack_dsyevr, 696
laplacian_normalization_t, 491
laplacian_spectral_embedding, 645
largest_cliques, 536
largest_independent_vertex_sets, 546
largest_weighted_cliques, 543
lastcit_game, 326
layout_align, 673
layout_bipartite, 651
layout_circle, 648
layout_davidson_harel, 658
layout_drl, 654
layout_drl_3d, 654
layout_drl_default_t, 653
layout_drl_options_init, 654
layout_drl_options_t, 652
layout_fruchterman_reingold, 655
layout_fruchterman_reingold_3d, 669
layout_gem, 658
layout_graphopt, 650
layout_grid, 649
layout_grid_3d, 669
layout_kamada_kawai, 656
layout_kamada_kawai_3d, 670
layout_lgl, 660
layout_mds, 660
layout_merge_dla, 672
layout_random, 648

layout_random_3d, 668
layout_reingold_tilford, 661
layout_reingold_tilford_circular, 662
layout_sphere, 668
layout_star, 649
layout_sugiyama, 664
layout_umap, 665
layout_umap_3d, 671
layout_umap_compute_weights, 666
lazy_adjlist_clear, 188
lazy_adjlist_destroy, 187
lazy_adjlist_get, 187
lazy_adjlist_has, 187
lazy_adjlist_init, 186
lazy_adjlist_size, 188
lazy_inclist_clear, 190
lazy_inclist_destroy, 189
lazy_inclist_get, 189
lazy_inclist_has, 190
lazy_inclist_init, 188
lazy_inclist_size, 190
lcf, 293
lcf_small, 294
le_community_to_membership, 617
linegraph, 378
list_triangles, 551
local_efficiency, 423
local_scan_0, 428
local_scan_0_them, 429
local_scan_1_ecount, 428
local_scan_1_ecount_them, 430
local_scan_k_ecount, 429
local_scan_k_ecount_them, 430
local_scan_neighborhood_ecount, 431
local_scan_subset_ecount, 431
loops_t, 520
lune_beta_skeleton, 360

M

malloc, 48
MATRIX, 96
matrix_add, 104
matrix_add_cols, 117
matrix_add_constant, 103
matrix_add_rows, 116
matrix_all_almost_e, 107
matrix_all_e, 107
matrix_all_g, 108
matrix_all_ge, 109
matrix_all_l, 108
matrix_all_le, 108
matrix_as_sparsemat, 147
matrix_capacity, 113
matrix_cbind, 110
matrix_colsum, 106
matrix_complex_all_almost_e, 120
matrix_complex_create, 119
matrix_complex_create_polar, 119
matrix_complex_imag, 118
matrix_complex_real, 118
matrix_complex_realimag, 118
matrix_complex_zapsmall, 120
matrix_contains, 115
matrix_copy_to, 98
matrix_destroy, 95
matrix_div_elements, 105
matrix_empty, 112
matrix_fill, 96
matrix_get, 96
matrix_get_col, 100
matrix_get_ptr, 97
matrix_get_row, 99
matrix_init, 94
matrix_init_array, 94
matrix_init_copy, 95
matrix_isnull, 113
matrix_is_symmetric, 114
matrix_max, 110
matrix_maxdifference, 114
matrix_min, 110
matrix_minmax, 111
matrix_mul_elements, 104
matrix_ncol, 114
matrix_nrow, 114
matrix_null, 95
matrix_prod, 105
matrix_rbind, 109
matrix_remove_col, 117
matrix_remove_row, 117
matrix_resize, 116
matrix_resize_min, 116
matrix_rowsum, 106
matrix_scale, 103
matrix_search, 115
matrix_select_cols, 102
matrix_select_rows, 102
matrix_select_rows_cols, 103
matrix_set, 97
matrix_set_col, 101
matrix_set_row, 100
matrix_size, 113
matrix_sub, 104
matrix_sum, 105
matrix_swap, 99
matrix_swap_cols, 101
matrix_swap_rows, 101
matrix_transpose, 106
matrix_update, 99
matrix_view, 97
matrix_view_from_vector, 98
matrix_which_max, 111
matrix_which_min, 111
matrix_which_minmax, 112
matrix_zapsmall, 109
maxdegree, 454
maxflow, 583

maxflow_stats_t, 586
maxflow_value, 584
maximal_cliques, 537
maximal_cliques_callback, 541
maximal_cliques_count, 538
maximal_cliques_file, 539
maximal_cliques_hist, 540
maximal_cliques_subset, 539
maximal_independent_vertex_sets, 546
maximum_bipartite_matching, 511
maximum_cardinality_search, 508
mean_degree, 513
metric_t, 357
mincut, 589
mincut_value, 590
minimum_cycle_basis, 531
minimum_size_separators, 600
minimum_spanning_tree, 480
modularity, 603
modularity_matrix, 604
motifs_handler_t, 554
motifs_randesu, 551
motifs_randesu_callback, 554
motifs_randesu_estimate, 553
motifs_randesu_no, 552
mycielskian, 378
mycielski_graph, 311

N

nearest_neighbor_graph, 358
neighborhood, 426
neighborhood_graphs, 427
neighborhood_size, 425
neighbors, 25
neimode_t, 521

O

OTHER, 22

P

pagerank, 449
pagerank_algo_t, 449
path_graph, 292
path_length_hist, 411
permute_vertices, 577
personalized_pagerank, 451
personalized_pagerank_vs, 452
plfit_result_calculate_p_value, 715
plfit_result_t, 713
power_law_fit, 714
preference_game, 318
product, 373
PROGRESS, 720
progress, 721
PROGRESSF, 720
progressf, 721
progress_handler_stderr, 719
progress_handler_t, 719
pseudo_diameter, 415
psumtree_destroy, 191
psumtree_get, 192
psumtree_init, 191
psumtree_reset, 193
psumtree_search, 192
psumtree_size, 191
psumtree_sum, 192
psumtree_update, 193

R

radius, 414
random_sample, 711
random_spanning_tree, 481
random_walk, 387
reachability, 435
read_graph_dimacs_flow, 680
read_graph_dl, 689
read_graph_edgelist, 676
read_graph_gml, 684
read_graph_graphdb, 682
read_graph_graphml, 683
read_graph_lgl, 679
read_graph_ncol, 677
read_graph_pajek, 686
realize_bipartite_degree_sequence, 302
realize_degree_sequence, 299
realloc, 49
recent_degree_aging_game, 325
recent_degree_game, 324
reciprocity, 514
regular_tree, 297
reindex_membership, 607
relative_neighborhood_graph, 359
reverse_edges, 381
rewire, 335
rewire_directed_edges, 341
rewire_edges, 341
rich_club_sequence, 505
ring, 291
rngtype_glibc2, 213
rngtype_mt19937, 213
rngtype_pcg32, 214
rngtype_pcg64, 214
rng_bits, 207
rng_default, 206
rng_destroy, 207
rng_get_binom, 211
rng_get_bool, 208
rng_get_exp, 210
rng_get_gamma, 211
rng_get_geom, 212
rng_get_integer, 209
rng_get_normal, 210
rng_get_pois, 212
rng_get_unif, 210
rng_get_unif01, 209

rng_init, 206
rng_max, 208
rng_name, 208
rng_sample_dirichlet, 713
rng_sample_sphere_surface, 711
rng_sample_sphere_volume, 712
rng_seed, 207
rng_set_default, 206
rooted_product, 375
roots_for_tree_layout, 663
running_mean, 710

S

sbm_game, 315
SETEAB, 272
SETEABV, 277
SETEAN, 271
SETEANV, 276
SETEAS, 273
SETEASV, 278
SETGAB, 267
SETGAN, 266
SETGAS, 268
setup, 17
SETVAB, 269
SETVABV, 275
SETVAN, 269
SETVANV, 274
SETVAS, 270
SETVASV, 275
set_attribute_table, 244
set_error_handler, 40
set_fatal_handler, 45
set_progress_handler, 719
set_status_handler, 723
set_warning_handler, 38
similarity_dice, 476
similarity_dice_es, 478
similarity_dice_pairs, 477
similarity_inverse_log_weighted, 479
similarity_jaccard, 474
similarity_jaccard_es, 476
similarity_jaccard_pairs, 475
simple_cycles, 522
simple_cycles_callback, 523
simple_interconnected_islands_game, 343
simplify, 379
simplify_and_colorize, 578
sir, 674
sir_destroy, 675
sir_t, 674
site_percolation, 440
small, 282
spanner, 388
sparsemat_add, 133
sparsemat_add_cols, 135
sparsemat_add_rows, 134
sparsemat_arpack_rnsolve, 146
sparsemat_arpack_rssolve, 145
sparsemat_as_matrix, 147
sparsemat_cholsol, 142
sparsemat_colsums, 130
sparsemat_compress, 139
sparsemat_count_nonzero, 129
sparsemat_count_nonzerotol, 129
sparsemat_destroy, 124
sparsemat_droptol, 132
sparsemat_dropzeros, 131
sparsemat_dupl, 139
sparsemat_entry, 130
sparsemat_fkeep, 131
sparsemat_gaxpy, 134
sparsemat_get, 126
sparsemat_getelements, 127
sparsemat_getelements_sorted, 127
sparsemat_index, 124
sparsemat_init, 121
sparsemat_init_copy, 122
sparsemat_init_diag, 122
sparsemat_init_eye, 123
sparsemat_is_cc, 126
sparsemat_is_symmetric, 126
sparsemat_is_triplet, 125
sparsemat_iterator_col, 137
sparsemat_iterator_end, 137
sparsemat_iterator_get, 138
sparsemat_iterator_idx, 138
sparsemat_iterator_init, 136
sparsemat_iterator_next, 138
sparsemat_iterator_reset, 136
sparsemat_iterator_row, 137
sparsemat_lsolve, 140
sparsemat_ltsolve, 141
sparsemat_lu, 143
sparsemat_luresol, 144
sparsemat_lusol, 142
sparsemat_max, 128
sparsemat_min, 128
sparsemat_minmax, 128
sparsemat_multiply, 134
sparsemat_ncol, 125
sparsemat_nonzero_storage, 130
sparsemat_nrow, 124
sparsemat_numeric_destroy, 145
sparsemat_permute, 132
sparsemat_print, 148
sparsemat_qr, 143
sparsemat_qrresol, 144
sparsemat_realloc, 123
sparsemat_resize, 135
sparsemat_rowsums, 129
sparsemat_scale, 132
sparsemat_sort, 136
sparsemat_symblu, 139
sparsemat_symbolic_destroy, 145
sparsemat_symbqr, 140

sparsemat_transpose, 133
sparsemat_type, 125
sparsemat_usolve, 141
sparsemat_utsolve, 141
sparse_adjacency, 286
sparse_weighted_adjacency, 286
spatial_edge_lengths, 362
split_join_distance, 609
square_lattice, 289
stack_clear, 150
stack_destroy, 148
stack_empty, 149
stack_init, 148
stack_pop, 150
stack_push, 150
stack_reserve, 149
stack_size, 149
stack_top, 151
star, 287
static_fitness_game, 337
static_power_law_game, 339
STATUS, 724
status, 724
STATUSF, 724
statusf, 725
status_handler_stderr, 723
status_handler_t, 723
STR, 159
strength, 455
strerror, 37
strvector_append, 162
strvector_capacity, 166
strvector_clear, 164
strvector_destroy, 159
strvector_get, 160
strvector_init, 158
strvector_init_copy, 158
strvector_merge, 163
strvector_push_back, 161
strvector_push_back_len, 161
strvector_remove, 162
strvector_remove_section, 162
strvector_reserve, 165
strvector_resize, 164
strvector_resize_min, 165
strvector_set, 160
strvector_set_len, 160
strvector_size, 165
strvector_swap, 163
strvector_swap_elements, 161
strvector_update, 164
st_edge_connectivity, 592
st_mincut, 586
st_mincut_value, 587
st_vertex_connectivity, 593
subcomponent, 432
subgraph_from_edges, 380
subisomorphic, 557
subisomorphic_lad, 573
subisomorphic_vf2, 569
symmetric_tree, 296

T

THREAD_SAFE, 718
TO, 22
topological_sorting, 526
to_directed, 488
to_prufer, 483
to_undirected, 489
transitive_closure, 436
transitivity_avglocal_undirected, 486
transitivity_barrat, 486
transitivity_local_undirected, 485
transitivity_undirected, 484
tree_from_parent_vector, 298
tree_game, 344
triad_census, 548
triangular_lattice, 290
trussness, 507
turan, 305

U

unfold_tree, 512
union, 366
union_many, 366
UNLIMITED, 31

V

VAB, 259
VABV, 260
VAN, 257
VANV, 258
VAS, 260
VASV, 261
vcount, 19
VCOUNT_MAX, 31
VECTOR, 55
vector_add, 62
vector_add_constant, 61
vector_all_almost_e, 64
vector_all_e, 64
vector_all_g, 65
vector_all_ge, 65
vector_all_l, 64
vector_all_le, 65
vector_append, 58
vector_binsearch, 75
vector_binsearch_slice, 75
vector_capacity, 71
vector_clear, 76
vector_colex_cmp, 68
vector_colex_cmp_untyped, 68
vector_complex_all_almost_e, 81
vector_complex_create, 81
vector_complex_create_polar, 81

vector_complex_imag, 80
vector_complex_real, 80
vector_complex_realimag, 80
vector_complex_zapsmall, 82
vector_contains, 74
vector_contains_sorted, 76
vector_copy_to, 58
vector_destroy, 53
vector_difference_and_intersection_sorted, 85
vector_difference_sorted, 85
vector_div, 63
vector_empty, 71
vector_fill, 54
vector_floor, 63
vector_get, 55
vector_get_ptr, 55
vector_index, 56
vector_index_in_place, 57
vector_init, 52
vector_init_array, 52
vector_init_copy, 52
vector_init_range, 53
vector_insert, 79
vector_intersection_size_sorted, 84
vector_intersect_sorted, 83
vector_isininterval, 72
vector_is_all_finite, 74
vector_is_any_nan, 74
vector_is_equal, 66
vector_is_nan, 73
vector_lex_cmp, 67
vector_lex_cmp_untyped, 67
vector_list_capacity, 171
vector_list_clear, 171
vector_list_destroy, 168
vector_list_discard, 176
vector_list_discard_back, 177
vector_list_discard_fast, 177
vector_list_empty, 170
vector_list_get_ptr, 169
vector_list_init, 167
vector_list_init_copy, 168
vector_list_insert, 174
vector_list_insert_copy, 174
vector_list_insert_new, 175
vector_list_permute, 177
vector_list_pop_back, 174
vector_list_push_back, 172
vector_list_push_back_copy, 173
vector_list_push_back_new, 173
vector_list_remove, 175
vector_list_remove_fast, 176
vector_list_replace, 170
vector_list_reserve, 171
vector_list_resize, 172
vector_list_set, 169
vector_list_size, 170
vector_list_sort, 178
vector_list_sort_ind, 178
vector_list_swap, 179
vector_list_swap_elements, 179
vector_list_tail_ptr, 169
vector_max, 69
vector_maxdifference, 73
vector_min, 69
vector_minmax, 70
vector_mul, 62
vector_null, 53
vector_permute, 61
vector_pop_back, 78
vector_prod, 72
vector_ptr_capacity, 88
vector_ptr_clear, 88
vector_ptr_destroy, 87
vector_ptr_destroy_all, 87
vector_ptr_free_all, 87
vector_ptr_get, 91
vector_ptr_get_item_destructor, 93
vector_ptr_init, 86
vector_ptr_init_copy, 86
vector_ptr_insert, 90
vector_ptr_permute, 92
vector_ptr_pop_back, 90
vector_ptr_push_back, 90
vector_ptr_reserve, 89
vector_ptr_resize, 89
vector_ptr_resize_min, 89
vector_ptr_set, 91
vector_ptr_set_item_destructor, 93
VECTOR_PTR_SET_ITEM_DESTRUCTOR, 93
vector_ptr_size, 88
vector_ptr_sort, 91
vector_ptr_sort_ind, 92
vector_push_back, 78
vector_range, 54
vector_remove, 79
vector_remove_section, 79
vector_reserve, 77
vector_resize, 77
vector_resize_min, 78
vector_reverse, 59
vector_reverse_section, 59
vector_reverse_sort, 83
vector_rotate_left, 60
vector_scale, 61
vector_search, 74
vector_set, 56
vector_shuffle, 60
vector_size, 71
vector_sort, 82
vector_sort_ind, 83
vector_sub, 62
vector_sum, 72
vector_swap, 59
vector_swap_elements, 59
vector_tail, 56

vector_update, 58
vector_view, 57
vector_which_max, 70
vector_which_min, 69
vector_which_minmax, 70
vector_zapsmall, 66
version, 710
vertex_coloring_greedy, 579
vertex_connectivity, 594
vertex_disjoint_paths, 596
vertex_path_from_edge_path, 417
vit_create, 225
vit_destroy, 225
VIT_END, 226
VIT_GET, 227
VIT_NEXT, 226
VIT_RESET, 227
VIT_SIZE, 226
voronoi, 416
vss_1, 223
vss_all, 223
vss_none, 223
vss_range, 224
vss_vector, 224
vs_1, 219
vs_adj, 217
vs_all, 216
vs_copy, 221
vs_destroy, 221
vs_is_all, 222
vs_nonadj, 217
vs_none, 218
vs_range, 221
vs_size, 222
vs_type, 222
vs_vector, 219
vs_vector_copy, 220
vs_vector_small, 220

W

WARNING, 38
warning, 38
WARNINGF, 38
warningf, 39
warning_handler_ignore, 39
warning_handler_print, 39
warning_handler_t, 37
watts_strogatz_game, 340
weighted_adjacency, 284
weighted_biadjacency, 352
weighted_cliques, 542
weighted_clique_number, 543
wheel, 288
widest_path_widths_dijkstra, 420
widest_path_widths_floyd_warshall, 421
write_graph_dimacs_flow, 681
write_graph_dot, 689
write_graph_edgelist, 676
write_graph_gml, 685
write_graph_graphml, 684
write_graph_leda, 690
write_graph_lgl, 680
write_graph_ncol, 678
write_graph_pajek, 688